Quantum Resistant Public Key Encryption Scheme RLCE and OpenSSL

Yongge Wang

UNC Charlotte

September 12, 2023

Outline

Quantum Computers and Post Quantum Security

- Quantum Adiabatic/Annealing Computation: D-Wave 7440 qubits quantum computers
- Circuit based quantum computers
  - Trapped ion based quantum computers: NIST, UMD, and Innsbruck (Austria)
  - Superconducting quantum computers: D-Wave, Google, IBM, Intel, Rigetti (a start up: ex-IBM-employees), etc.
  - Photonics based quantum computers: Aaronson's Boson Sampling quantum computer model uses photons
  - Topological QC with non-abelian anyons: Microsoft

Post Quantum Security

- Peter Shor's algorithm to factor a large integer – All web services will be hacked and gone!
- Grover's algorithm to speed up database search. For an unsorted database of $2^{10} = 1024$ entries, classical search requires 500 steps on average. Quantum computer requires $2^5 = 32$ steps. For a DB of $2^{20} = 1048576$ entries, QC only needs 1024 steps to search (1000 times improvement)

Post Quantum Security

- A NIST published report from April 2016 acknowledges the possibility of quantum technology to render the commonly used RSA algorithm insecure by 2030
- In December 2016 NIST initiated a standardization process by announcing a call for proposals.

Post Quantum Cryptography: lattice based

- **Shortest Vector Problem (SVP)**: Given an arbitrary basis B of a lattice $\mathcal{L}(B)$, find a shortest nonzero lattice vector.
- **Short Integer Solution (SIS)**: Given m uniformly random vectors $\mathbf{a}_i \in Z_q^n$, forming the columns of a matrix $A \in Z_q^{n \times m}$, find a nonzero integer vector $z \in Z^m$ of norm $\|z\| \leq \beta$ with

$$A\mathbf{z} = \mathbf{0} \in Z_q^n$$

- **Learning with Error (LWE)**: Let $s \in Z_q^n$ be a secret. Given m independent samples $\mathbf{a}_i \in Z_q^n$ and (discretized) Gaussian error distribution $e_i \in Z_q$, output $\mathbf{a}_i \cdot \mathbf{s} + e_i$. Find $\mathbf{s}$.

Ring-SIS

- Lattice based encryption scheme has large key size (comparable with code based schemes)
- a trick: use ideal lattices and Ring-SIS: the generator matrices are cyclic (or the base is cyclic etc.)
- Ring-LWE and Ring-SIS hard? We are not sure yet!

McEliece Scheme

McEliece Scheme (1978)

`Mc.KeySetup`: An $(n, k, 2t+1)$ linear Goppa code $\mathcal{C}$ with $k \times n$ generator matrix G_s. Public key: $G = SG_sP$. Private key: G_s Where S is random and P is permutation.

`Mc.Enc`$(G, \mathbf{m}, \mathbf{e})$. For a message $\mathbf{m} \in \{0,1\}^k$, choose a random vector $\mathbf{e} \in \{0,1\}^n$ of weight t. The cipher text $\mathbf{c} = \mathbf{m}G + \mathbf{e}$

`Mc.Dec`$(S, G_s, P, \mathbf{c})$. For a received ciphertext $\mathbf{c}$, first compute $\mathbf{c}' = \mathbf{c}P^{-1} = \mathbf{m}SG$. Next use an error-correction algorithm to recover $\mathbf{m}' = \mathbf{m}S$ and compute the message $\mathbf{m}$ as $\mathbf{m} = \mathbf{m}'S^{-1}$.

Security

- Broken ones: Niederreiter's scheme with Generalized Reed-Solomon Code Broken
- Broken ones: Wild Goppa code based McEliece, GRS-McEliece with random columns
- Unbroken ones: Original McEliece, MDPC/LDPC McEliece, Wang's RLCE

RLCE Key setup

`RLCE.KeySetup`. Let G_s be a $k \times n$ generator matrix for an $[n, k, d]$ linear code $\mathcal{C}$correcting at least t errors. Let $G_s P_1 = [\mathbf{g}_0, \cdots, \mathbf{g}_{n-1}]$ for a random permutation P_1

1. Let $G_1 = [\mathbf{g}_0, \cdots, \mathbf{g}_{n-w}, \mathbf{r}_0, \cdots, \mathbf{g}_{n-1}, \mathbf{r}_{w-1}]$ be a $k \times (n+w)$ matrix where $\mathbf{r}_i \in GF(q)^k$ are random
2. Let $A_i \in GF(q)^{2\times 2}$ be random 2×2 matrices. Let $A = \text{diag}[I_{n-w}, A_0, \cdots, A_{w-1}]$ be an $(n+w) \times (n+w)$ non-singular matrix.
3. The public key: $k \times (n+w)$ matrix $G = SG_1AP_2$ and the private key: (S, G_s, P_1, P_2, A) where S is random $k \times k$ matrix and P_2 is a permutation.

RLCE Encryption/Decryption

RLCE.Enc$(G, \mathbf{m}, \mathbf{e})$. For a message $\mathbf{m} \in GF(q)^k$, choose $\mathbf{e} \in GF(q)^{n+w}$ of weight at most t. The cipher: $\mathbf{c} = \mathbf{m}G + \mathbf{e}$.

RLCE.Dec$(S, G_s, P_1, P_2, A, \mathbf{c})$. For a cipher text $\mathbf{c}$, compute

$$\mathbf{c}P_2^{-1}A^{-1} = \mathbf{m}SG_1 + \mathbf{e}P_2^{-1}A^{-1} = [c'_0, \ldots, c'_{n+w-1}].$$

Let $\mathbf{c}' = [c'_0, c'_1, \cdots, c'_{n-w}, c'_{n-w+2}, \cdots, c'_{n+w-2}] \in GF(q)^n$. Then $\mathbf{c}'P_1^{-1} = \mathbf{m}SG_s + \mathbf{e}'$ for some $\mathbf{e}' \in GF(q)^n$ of weight at most t. Using an efficient decoding algorithm, one can recover $\mathbf{m}SG_s$ from $\mathbf{c}'P_1^{-1}$. Let D be a $k \times k$ inverse matrix of SG'_s where G'_s is the first k columns of G_s. Then $\mathbf{m} = \mathbf{c}_1 D$ where $\mathbf{c}_1$ is the first k elements of $\mathbf{m}SG_s$.

Recommended parameters

RLCE

κ_c, κ_q	sk	cipher	pk
128, 80	310116	988	188001
192,110	747393	1545	450761
256,144	1773271	2640	1232001

McEliece

ID	κ_c, κ_q	sk	cipher	pk
mceliece348864[f]	**128, 80**	6452	128	261120
mceliece460896[f]	**192,110**	13568	188	524160
mceliece6688128[f]	**256,144**	13892	240	1044992
mceliece6960119[f]	**256,144**	13908	226	1047319
mceliece8192128[f]	**256,144**	14080	240	1357824

RLCE and RSA performance (milliseconds)

κ_C	RSA modulus	key setup		encryption		decryption	
		RSA	RLCE	RSA	RLCE	RSA	RLCE
128	3072	433.607	151.834	0.135540	0.360	6.576281	1.345
192	7680	9346.846	637.988	0.672769	0.776	75.075443	2.676
256	15360	80790.751	1587.330	2.498523	1.745	560.225740	9.383

Information-set decoding (ISD)

- Information-set decoding (ISD) is one of the most important message recovery attacks on McEliece encryption schemes.
- For the RLCE encryption scheme, the ISD attack is based on the number of columns in the public key G instead of the number of columns in the private key G_s.
- The cost of ISD attack on an $[n, k, t; w]$-RLCE scheme is equivalent to the cost of ISD attack on an $[n + w, k; t]$-McEliece scheme.

Naive ISD

- Uniformly selects k columns from the public key and checks whether it is invertible.
- If it is invertible, one multiplies the inverse with the corresponding ciphertext values in these coordinates that correspond to the k columns of the public key.
- If these coordinates contain no errors in the ciphertext, one recovers the plain text.

Quantum ISD

- For a function $f : \{0,1\}^l \to \{0,1\}$ with the property that there is an $x_0 \in \{0,1\}^l$ such that $f(x_0) = 1$ and $f(x) = 0$ for all $x \neq x_0$, Grover's algorithm finds the value x_0 using $\frac{\pi}{4}\sqrt{2^l}$ Grover iterations and $O(l)$ qubits.
- Grover's algorithm converts the function f to a reversible circuit C_f and calculates

$$|x\rangle \xrightarrow{C_f} (-1)^{f(x)}|x\rangle$$

in each of the Grover iterations. Thus the total steps for Grover's algorithm is bounded by $\frac{\pi|C_f|}{4}\sqrt{2^l}$.

Quantum ISD against RLCE

Thus Grover's quantum algorithm requires approximately

$$7\left((n+w)k+k^{2.807}+k^2\right)(\log_2 q)^{1.585}\sqrt{\frac{\binom{n+w}{k}}{\binom{n+w-t}{k}}}$$

steps for the simple ISD algorithm against RLCE encryption scheme.

ISD for systematic RLCE schemes

- One uniformly selects $k = k_1 + k_2$ columns from the public key where k_1 columns are from the first k columns of the public key.
- Assume that first k_1 columns have no error. Simplify the computation process for ISD

Insecure ciphertexts for systematic RLCE schemes

- For a systematic RLCE, if a small number of errors were added to the first k components of the ciphertext, one may be able to exhaustively search these errors.
- Let

$$\gamma_l = \max_{l \le i \le t} \left\{ \frac{\binom{k-l}{k-i}}{q^i \binom{k}{i}} \right\}$$

The RLCE produces an insecure ciphertext in case that the ciphertext contains at most l errors within the first k components of the ciphertext and $\gamma_l > 2^{-\kappa_c}$ where κ_c is the security parameter.

Sidelnikov-Shestakov's attack

- If $w \geq n - k$, not enough equations for Sidelnikov-Shestakov's attack
- If $w < n - k$, one need to guess some values to establish enough equations. The guess space is normally too big to be successful.

Known non-randomized column attack

- What happens if the positions of non-randomized $n - w$ GRS columns are known to the adversary?
- Possibility one: guess the remaining w columns of the GRS generator matrix. Search space too big
- Use Sidelnikov-Shestakov attack to calculate a private key for the punctured $[n - w, k]$ GRS_k code consisting of the non-randomized GRS columns and then list-decode the punctured $[n - w, k]$ GRS_k code.

Filtration attacks

- For two codes $\mathcal{C}_1$ and $\mathcal{C}_2$ of length n, the star product code $\mathcal{C}_1 * \mathcal{C}_2$ is the vector space spanned by $\mathbf{a} * \mathbf{b}$ for all pairs $(\mathbf{a}, \mathbf{b}) \in \mathcal{C}_1 \times \mathcal{C}_2$ where $\mathbf{a} * \mathbf{b} = [a_0 b_0, a_1 b_1, \cdots, a_{n-1} b_{n-1}]$.
- For the square code $\mathcal{C}^2 = \mathcal{C} * \mathcal{C}$ of $\mathcal{C}$, we have $\dim \mathcal{C}^2 \leq \min \left\{ n, \binom{\dim \mathcal{C}+1}{2} \right\}$.
- For an $[n, k]$ GRS code $\mathcal{C}$, let $\mathbf{a}, \mathbf{b} \in \mathrm{GRS}_k(\mathbf{x}, \mathbf{y})$ where $\mathbf{a} = (y_0 p_1(x_0), \cdots, y_{n-1} p_1(x_{n-1}))$ and $\mathbf{b} = (y_0 p_2(x_0), \cdots, y_{n-1} p_2(x_{n-1}))$. Then $\mathbf{a} * \mathbf{b} = (y_0^2 p_1(x_0) p_2(x_0), \cdots, y_{n-1}^2 p_1(x_{n-1}) p_2(x_{n-1}))$. Thus $\mathrm{GRS}_k(\mathbf{x}, \mathbf{y})^2 \subseteq \mathrm{GRS}_{2k-1}(\mathbf{x}, \mathbf{y} * \mathbf{y})$ where we assume $2k - 1 \leq n$.

Filtration attacks against GRS-RLCE

- G is public key for an (n, k, d, t, w) GRS-RLCE scheme.
- Let $\mathcal{C}$ be the code generated by the rows of G.
- Let $\mathcal{D}_1$ be the code with a generator matrix D_1 obtained from G by replacing the randomized $2w$ columns with all-zero columns and let $\mathcal{D}_2$ be the code with a generator matrix D_2 obtained from G by replacing the $n - w$ non-randomized columns with zero columns.
- Since $\mathcal{C} \subset \mathcal{D}_1 + \mathcal{D}_2$ and the pair $(\mathcal{D}_1, \mathcal{D}_2)$ is an orthogonal pair, we have $\mathcal{C}^2 \subset \mathcal{D}_1^2 + \mathcal{D}_2^2$. It follows that

$$2k - 1 \leq \dim \mathcal{C}^2 \leq \min\{2k - 1, n - w\} + 2w \qquad (1)$$

where we assume that $2w \leq k^2$.

Filtration attacks against GRS-RLCE: $k \geq n - w$

- Assume that the $2w$ randomized columns in $\mathcal{D}_2$ behave like random columns in the filtration attacks
- We have $\dim \mathcal{C}^2 = \mathcal{D}_1^2 + \mathcal{D}_2^2 = n - w + \mathcal{D}_2^2 = n + w$.
- For any code $\mathcal{C}'$ of length n' that is obtained from $\mathcal{C}$ using code puncturing and code shortening, we have $\dim \mathcal{C}'^2 = n'$.
- Thus filtration techniques could not be used to recover any non-randomized columns in D_1.

Filtration attacks against GRS-RLCE: $k < 2k \leq n - w$

- let $\mathcal{C}_i$ be the punctured $\mathcal{C}$ code at position i. We distinguish the following two cases:
  - Column i of G is a randomized column: the expected dimension for $\mathcal{C}_i^2$ is $2k + 2w - 2$.
  - Column i of G is a non-randomized column: the expected dimension for $\mathcal{C}_i^2$ is $2k + 2w - 1$.
- This shows that if $n - w \geq 2k$, then the filtration techniques could be used to identify the randomized columns within the public key G. Thus it is recommended to have $n - w < 2k$ for RLCE scheme.

Filtration attacks against GRS-RLCE: $k < n - w < 2k$

- Shorten $l = l_1 + l_2 < k - 1$ columns from G where l_1 columns are from D_1 and l_2 columns are from D_2. The shortened code has dimension

$$d_{l,l_1} = \min\{(k-l)^2, \min\{2(k-l_1)-1, n-w-l_1, (k-l)^2\}+ \min\{2w-l_2, (k-l)^2\}\}.$$

- A necessary condition for the filtration attack

$$d_{l,l_1} = 2(k-l_1) - 1 + \min\left\{2w - l_2, (k-l)^2\right\}. \qquad (2)$$

- Given l, the probability for the filtration attack is bounded by

$$\frac{\sum_{l_1=\max\{0,l-2w\}}^{l} \lambda(d_{l,l_1}) \binom{n-w}{l_1} \binom{2w}{l-l_1}}{\binom{n+w}{l}}$$

where $\lambda(d_{l,l_1}) = 1$ if (2) holds and $\lambda(d_{l,l_1}) = 0$ otherwise.

Filtration attacks with brute-force

- The adversary may carry out a filtration attack by exhaustively searching some GRS columns.
- That is, the adversary randomly selects $u \leq w$ pairs of columns from the public key G with the hope that u columns of the underlying GRS code generator matrix could be reconstructed using exhaustive search.
- One can obtain u columns of GRS code generator matrix from the public key with a probability $\frac{\binom{w}{u}}{q^{2u}\binom{n+w}{2u}}$.
- Assume that one has correctly guessed u columns of the GRS code generator matrix and $k < n - w + u$. One can continue the standard filtration attack
- The probability is quite small generally

libOQS and OpenSSL

- libOQS: https://openquantumsafe.org/liboqs/
- libOQS in OpenSSL
- it was not able to integrate McEliece into OpenSSL
- TLS 1.3: https://www.rfc-editor.org/rfc/rfc8446

PQC in OpenSSL – RFC 8446

```
struct {
      ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
      Random random;
      opaque legacy_session_id<0..32>;
      CipherSuite cipher_suites<2..2^16-2>;
      opaque legacy_compression_methods<1..2^8-1>;
      Extension extensions<8..2^16-1>;
} ClientHello;
```

- The challenge: the extension is at most 2^{16} bytes. That is, at most 65,536 bytes (65KB).
- if public key is larger than 65KB, then it will just not work!

PQC in OpenSSL – RFC 8446

- the PQC revision should work for (1) and (3) of TLS 1.3
  1. (EC)DHE: replace DH with RLCE/McELience
  2. PSK-only
  3. PSK with (EC)DHE: replace DHE with RLCE/McELience
- Implementation discussions: a client/sever can use key_share_PQC (52) or psk_key_exchange_modes_PQC (53) to send KEM ciphertexts. For short key PQC schemes, it must be included in key_share (51) or psk_key_exchange_modes (45). If long key PQC (McEliece/RLCE) is used, it must use ExtensionType 52 and 53.

PQC in OpenSSL – RFC 8446 revised

```
struct {
        ExtensionType extension_type;
        opaque extension_data<0..2^16-1>;
    } Extension;
    enum {
        server_name(0),                             /* RFC 6066 */
        max_fragment_length(1),                     /* RFC 6066 */
        status_request(5),                          /* RFC 6066 */
        supported_groups(10),                       /* RFC 8422, 7919 */
        signature_algorithms(13),                   /* RFC 8446 */
        use_srtp(14),                               /* RFC 5764 */
        heartbeat(15),                              /* RFC 6520 */
        application_layer_protocol_negotiation(16), /* RFC 7301 */
        signed_certificate_timestamp(18),           /* RFC 6962 */
        client_certificate_type(19),                /* RFC 7250 */
        server_certificate_type(20),                /* RFC 7250 */
        padding(21),                                /* RFC 7685 */
        pre_shared_key(41),                         /* RFC 8446 */
        early_data(42),                             /* RFC 8446 */
        supported_versions(43),                     /* RFC 8446 */
        cookie(44),                                 /* RFC 8446 */
        psk_key_exchange_modes(45),                 /* RFC 8446 */
        certificate_authorities(47),                /* RFC 8446 */
        oid_filters(48),                            /* RFC 8446 */
        post_handshake_auth(49),                    /* RFC 8446 */
        signature_algorithms_cert(50),              /* RFC 8446 */
        key_share(51),                              /* RFC 8446 */
        key_share_PQC (52),
        psk_key_exchange_modes_PQC (53),
        (65535)
```

PQC in OpenSSL – RFC 8446 revised

```
uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];     /* Crypto suite selector */
struct {
          ExtensionType extension_type;
          select (Extension.extension_type) {
              case 52 or 53:  opaque extension_data<8..2^22-1>;
              case  default:  opaque extension_data<8..2^16-1>;
          };
} Extension

struct {
          ProtocolVersion legacy_version=0x0303; /*TLS v1.2*/
          Random random;
          opaque legacy_session_id<0..32>;
          CipherSuite cipher_suites<2..2^16-2>;
          opaque legacy_compression_methods<1..2^8-1>;
          Extension extensions<8..2^22-1>;
} ClientHello;

struct {
          ProtocolVersion legacy_version = 0x0303;     /* TLS v1.2 */
          Random random;
          opaque legacy_session_id_echo<0..32>;
          CipherSuite cipher_suite;
          uint8 legacy_compression_method = 0;
          Extension extensions<6..2^22-1>;
} ServerHello;
```

PQC in OpenSSL – RFC 8446 revised

```
enum {
        /* Elliptic Curve Groups (ECDHE) */
        secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
        x25519(0x001D), x448(0x001E),

        /* Finite Field Groups (DHE) */
        ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
        ffdhe6144(0x0103), ffdhe8192(0x0104),

        /* Reserved Code Points */
        ffdhe_private_use(0x01FC..0x01FF),
        ecdhe_private_use(0xFE00..0xFEFF),

        /* PQC */
        rlcel1 (0x024D), rlcel3 (0x024E), rlcel5 (0x024F),
        mceliece1 (0x025D),  mceliece3 (0x025E),  mceliece5 (0x025F),

        (0xFFFF)
} NamedGroup;
```

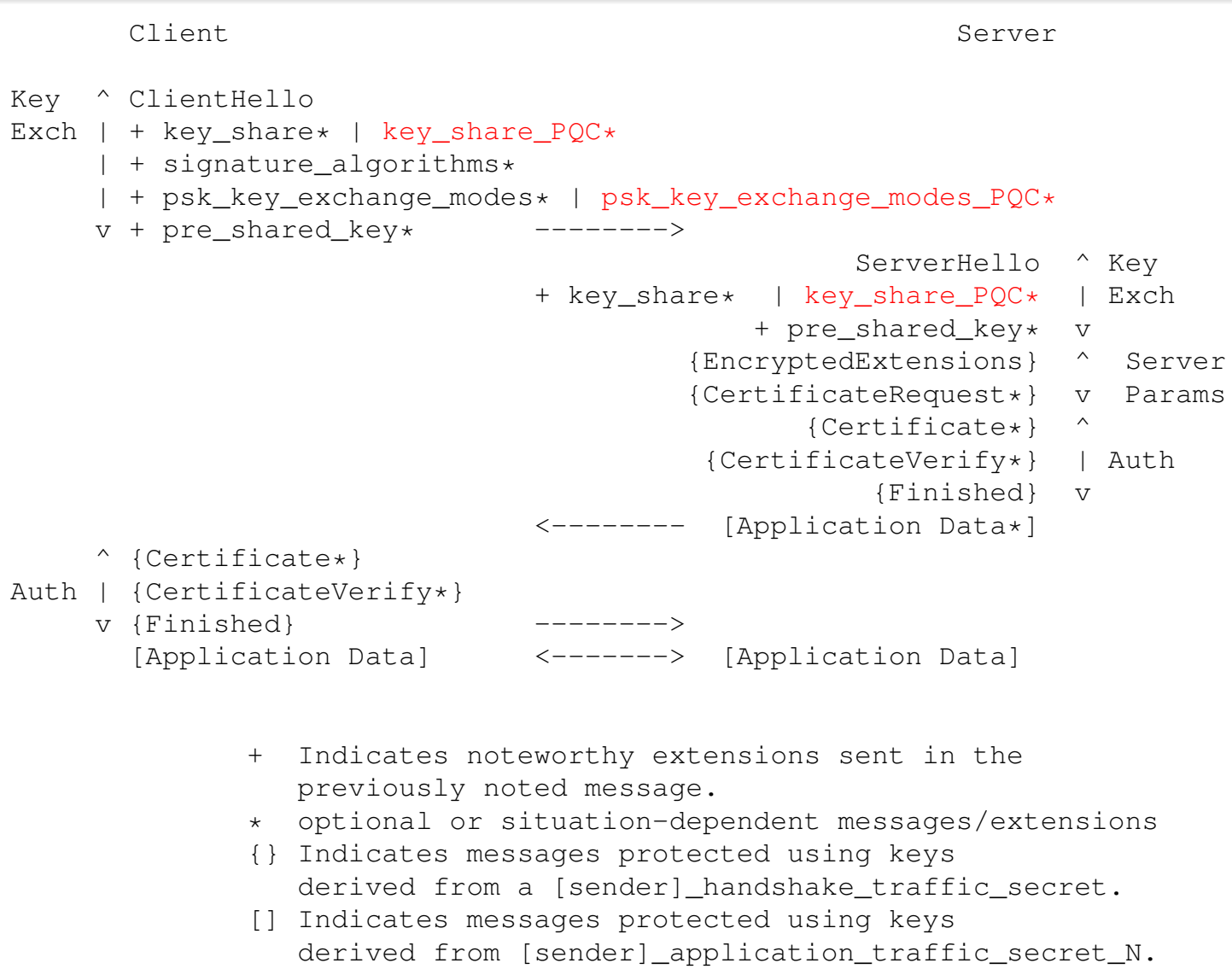
PQC in OpenSSL – RFC 8446 revised

```
struct {
    NamedGroup group;
    select (KeyShareEntry.group) {
       case rlcel1 | rlcel3 | rlcel5 :          opaque key_exchange<1..2^22-1>;
       case mceliece1 | mceliece3 | mceliece5 : opaque key_exchange<1..2^22-1>;
       default:                                 opaque key_exchange<1..2^16-1>;
    }
} KeyShareEntry;

struct {
     KeyShareEntry client_shares<0..2^22-1>;
}  KeyShareClientHello;


struct {
      KeyShareEntry server_share<0..2^22-1>;
} KeyShareServerHello;
```

PQC in OpenSSL – RFC 8446 revised

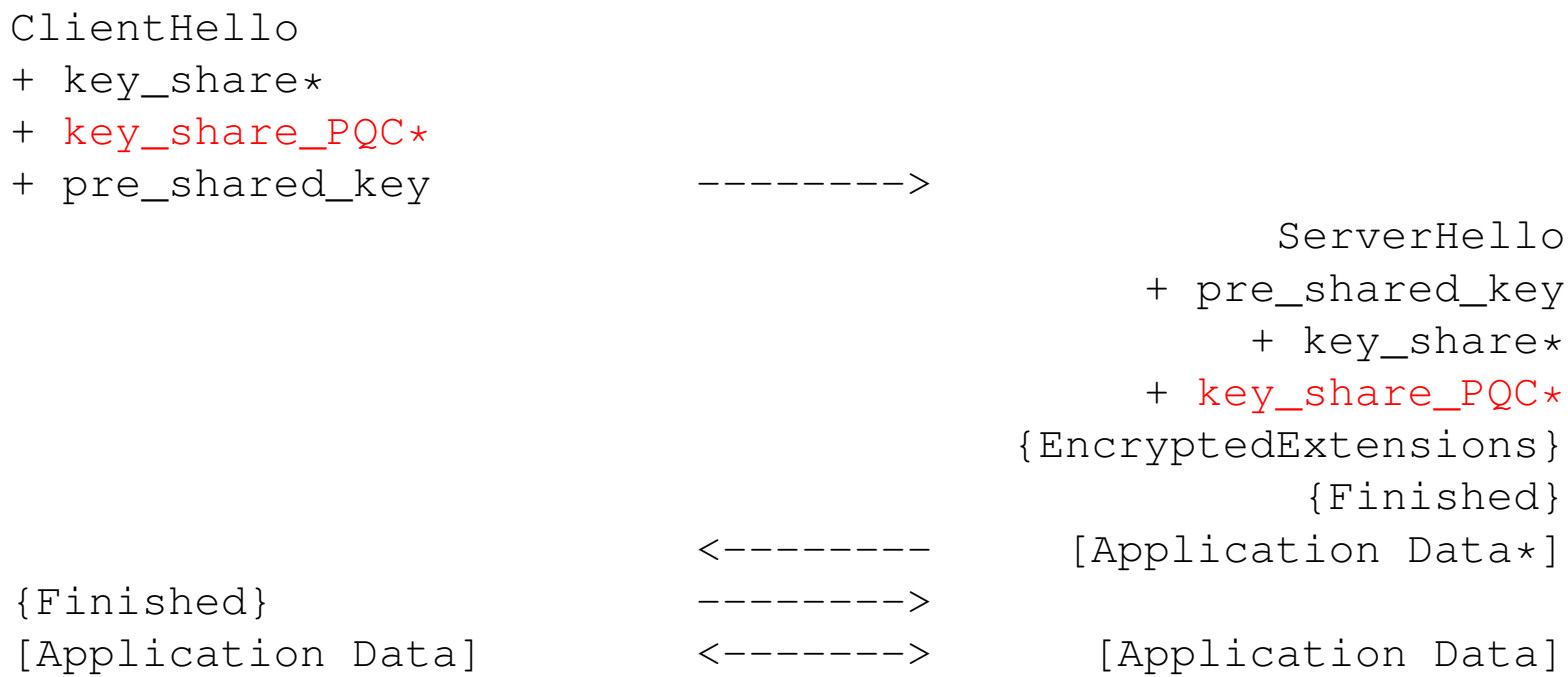
```
enum {
    psk_ke(0), psk_dhe_ke(1), psk_dhe_ke_pqc(2), (255)
} PskKeyExchangeMode;

struct {
    PskKeyExchangeMode ke_modes<1..255>;
} PskKeyExchangeModes;
```

RFC 8446 revised: basic full TLS handshake



RFC 8446 revised: Message Flow with Incorrect DHE Share

RFC 8446 revised: Resumption using PSK mode



RFC 8446 revised: 0-RTT Data

Experiments: integrate RLCE into libOQS

- added RLCE to libOQS
- revised openSSL with the proposed revisions
- testing works with all servers.
- available at: `https://github.com/yonggewang/openssl`
- available at: `https://github.com/yonggewang/rlce`

Questions

Questions?