Transport Layer Security
Internet-Draft
Intended status: Standards Track
Expires: 24 August 2026

J. Wagner
Y. Wang
UNC Charlotte
V. Smyslov
ELVIS-PLUS
Y. Nir
Dell Technologies
20 February 2026

# Larger Data in TLS 1.3 Handshake
# draft-wagner-tls-keysharepqc-08

## Abstract

This memo discusses possible modifications of TLS 1.3, aimed to allow trasferring data larger than 64 Kbytes in handshake messages. One possible application for this feature is to allow using post-quantum Key Encapsulation Method that have large public key or ciphertext size (like Classic McEliece).

## About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://jwagrunner.github.io/internet-draft/draft-wagner-tls-keysharepqc.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-wagner-tls-keysharepqc/.

Discussion of this document takes place on the Transport Layer Security mailing list (mailto:tls@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/tls/. Subscribe at https://www.ietf.org/mailman/listinfo/tls/.

Source for this draft and an issue tracker can be found at https://github.com/jwagrunner/internet-draft.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

## Copyright Notice



## Table of Contents

## 1. Introduction

The Transport Layer Security (TLS) Protocol Version 1.3 ([RFC8446]) is widely used to protect network traffic. To establish secure connection client and server first perform a "handshake" during which they negotiate cipher suites, compute shared session key and perform one-side or mutual authentication. TLS 1.3 handshake protocol consists of several messages, and while the size of each handshake message can be up to 2^24 bytes the size of some individual data blocks inside these messages is limited to 2^16 bytes. This limitation makes it impossible to transfer larger data blocks in TLS 1.3 handshake.

One possible application for larger data in TLS 1.3 handshake is post-quantum Key Encapsulation Mechanisms (KEM). Large public key algorithms, including the code-based cryptographic algorithm family Classic McEliece (see [I-D.josefsson-mceliece], [NIST], [DJB25], [RJM78], and [OQS24]), cannot be easily implemented in TLS 1.3 due to the current key share limitations of 65,535 Bytes. It is important to consider such uses of algorithms given that Classic McEliece is a Round 4 algorithm submitted in the National Institute of Standards and Technology (NIST) standardization process (see [PQC25]). Thus, enabling the use of Classic McEliece algorithms to be used in TLS 1.3 key exchanges and also presenting them as an alternative option to replace classical algorithms for future protection against the threat of attackers in possession of powerful quantum computers that will break classical encryption.

This document discusses the possible ways how the TLS 1.3 handshake can accommodate data larger than 64 Kbytes with an immediate goal to be able to run large public key KEMs, but not limited to.

## 2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

## 3. Possible Solutions to the Problem

### 3.1. New Key Share Extension

Based on the key share extension from [RFC8446] is introduced a new key share extension in this document, "key_share_pqc". This is reflected in this document and is represented as KeyShareEntryPQC below, based on the existing KeyShareEntry from [RFC8446]. However, this is modified along with the existing KeyShareEntry structure to test if the key exchange algorithm chosen in a TLS 1.3 connection belongs from the Classic McEliece family, and if it is, then KeyShareEntryPQC is constructed. If the opposite is true, where the key exchange algorithm is not from the Classic McEliece family, then KeyShareEntry is constructed. Note that the "key_exchange" fields are expanded in KeyShareEntryPQC to accommodate a large public key that is greater than 65,535 Bytes:

```
struct {
  NamedGroup group;
  select (KeyShareEntry.group) {
  case classicmceliece348864:       Empty;
  case classicmceliece460896:       Empty;
  case classicmceliece6688128:      Empty;
  case classicmceliece6960119:      Empty;
  case classicmceliece8192128:      Empty;
  case x25519classicmceliece348864: Empty;
  case rlcel5:                      Empty;
  case other large PQ algorithm1:   Empty;
  case other large PQ algorithm2:   Empty;
  case etc.:                        Empty;
  default:                          opaque key_exchange<1..2^16-1>;
  }
} KeyShareEntry;

struct {
  NamedGroup group;
  select (KeyShareEntryPQC.group) {
  case classicmceliece348864:       opaque key_exchange<1..2^24-1>;
  case classicmceliece460896:       opaque key_exchange<1..2^24-1>;
  case classicmceliece6688128:      opaque key_exchange<1..2^24-1>;
  case classicmceliece6960119:      opaque key_exchange<1..2^24-1>;
  case classicmceliece8192128:      opaque key_exchange<1..2^24-1>;
  case x25519classicmceliece348864: opaque key_exchange<1..2^24-1>;
  case rlcel5:                      opaque key_exchange<1..2^24-1>;
  case other large PQ algorithm1:   opaque key_exchange<1..2^24-1>;
  case other large PQ algorithm2:   opaque key_exchange<1..2^24-1>;
  case etc.:                        opaque key_exchange<1..2^24-1>;
  default:                          Empty;
  }
} KeyShareEntryPQC;
```

Note: PQ (Post-Quantum) where "other large PQ algorithm1" and "other large PQ algorithm2" and "etc." above indicates that one or more future post-quantum algorithms with large public key sizes can be added by just defining a constant for each of these post-quantum algorithms.

Another Note: An additional algorithm is included in the above, "rlcel5", since it also has a large public key beyond the 65,535 Byte limit. See Section 7 for more information discussing this RLCE algorithm.

This is then applied to the existing KeyShareClientHello structure, which originates from [RFC8446], that now contains an additional field for KeyShareEntryPQC:

```
struct {
   KeyShareEntry client_shares<0..2^16-1>;
   KeyShareEntryPQC client_shares<0..2^24-1>;
} KeyShareClientHello;
```

Since the KeyShareClientHello needs to be expanded to accommodate for the KeyShareEntryPQC struct, the same applies to the existing Extension struct, originated as well from [RFC8446] but "extension_data" is now expanded:

```
struct {
  ExtensionType extension_type;
  opaque extension_data<0..2^24-1>;
} Extension;
```

Since there is a new key share extension to accommodate keys larger than the 65,535 Byte limit (KeyShareEntryPQC), this is reflected in the existing ExtensionType structure from [RFC8446] where this is the new type that holds a value of TBD, "key_share_pqc":

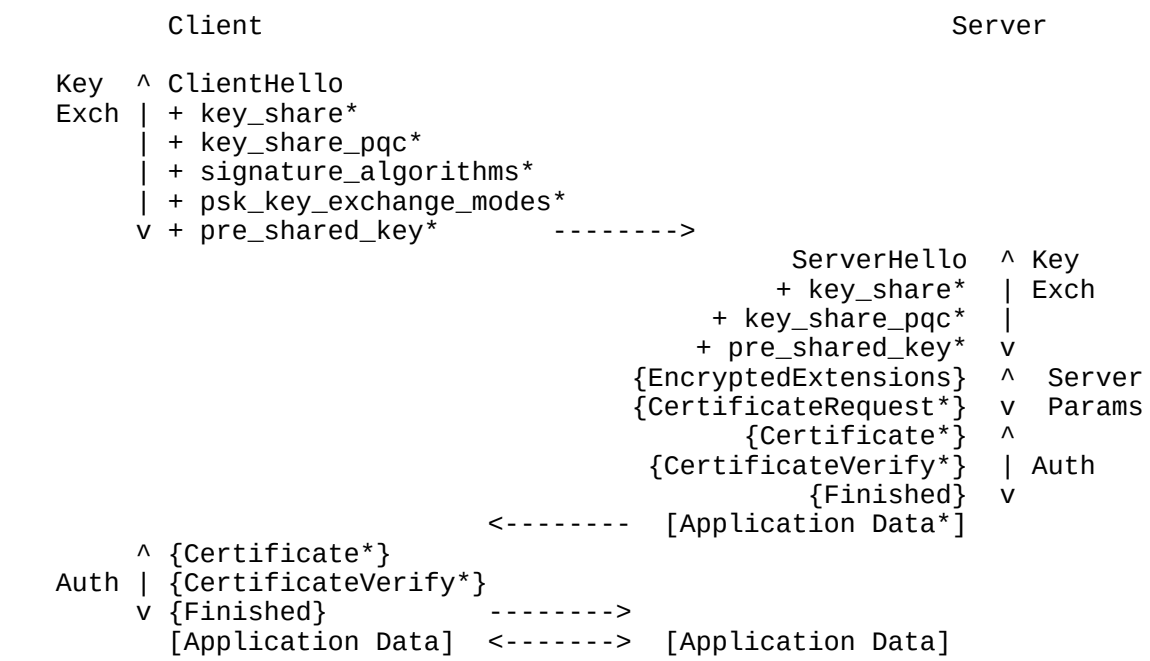
```
   enum {
       server_name(0),                             /* RFC 6066 */
       max_fragment_length(1),                     /* RFC 6066 */
       status_request(5),                          /* RFC 6066 */
       supported_groups(10),                       /* RFC 8422, 7919 */
       signature_algorithms(13),                   /* 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(TBD),
       (65535)
   } ExtensionType;
```

Since the "extension_data" field will be much larger for a KeyShareClientHello that contains a large public key that is greater than the previously defined 65,535 Byte limit, an example being a Classic McEliece public key, the server must be able to handle this circumstance when receiving the ClientHello message. One way is to compare the value for a packet that contains extensions including a large public key from the ClientHello message to a macro constant (for example, "CLIENT_HELLO_MIN_EXT_LENGTH" as defined in this introduced TLS implementation in this paper, see [SRVR1650] and [SRVR1211]) and if this packet value is longer than this constant, the server will change the way it normally handles all of the extensions. This constant could be easily modified in the aforementioned TLS Open Secure Socket Layer (OpenSSL) implementation. The process of how the server collects the extensions from a ClientHello message must also be modified, as the server must be able to process the new key share extension differently than the other extensions, should the server see this inside a ClientHello message. For example, see [EXT652].

The ServerHello message is modified as well where the KeyShareServerHello structure originates from [RFC8446]:

```
struct {
    KeyShareEntry server_share;
    KeyShareEntryPQC server_sharePQC;
} KeyShareServerHello;
```

This new "key_share_pqc" extension is therefore can be implemented in the full TLS handshake, where Figure 1 from [RFC8446] is modified to be the following:




+ Indicates noteworthy extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages/extensions that are not always sent.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 1: Full TLS Handshake with "key_share_pqc" extension

#### 3.1.1. Modification to PskKeyExchangeMode structure

There are two key establishments that are considered when examining the structure of PskKeyExchangeMode from [RFC8446]. Since there is no Diffie Hellman algorithm in use with a pre-shared key (PSK) when considering the use of a Classic McEliece algorithm for key exchange, then there must be another key exchange mode to utilize in this case. Therefore, this is reflected in the existing [RFC8446] PskKeyExchangeMode structure below where "psk_pqc_ke(2)" is added:

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

When selecting a Classic McEliece algorithm and using an external PSK or a resumption PSK, "02" will then be listed for the "psk_key_exchange_modes" extension along with the new "key_share_pqc" extension in the ClientHello message. At the end of this ClientHello message is printed the "00 29" extension (pre-shared key extension), where the PSK identity should be printed and is mapped to the binder that should proceed it in this pre-shared key extension. The ServerHello message will also contain the new "key_share_pqc" extension, and will as well contain the pre-shared key extension, where it should contain "00 00" at the end which represents the server selecting the PSK identity of 0 (for example: the Selected Identity of 0 shown in the pre-shared key extension in a ServerHello message in this Wireshark example: [RASHOK20]). Overall, this is a new key exchange selecting a Classic McEliece algorithm using a PSK, whether its external or resumption, and this can be demonstrated in the TLS Implementation below.

As stated above, resumption PSK with a Classic McEliece algorithm chosen as a key exchange algorithm involves the use of the new "key_share_pqc" extension for both the ClientHello and ServerHello messages. Thus, the Resumption and PSK Message Flow diagram (which originates from Figure 3 of [RFC8446]) is derived for this situation and has been tested with the TLS Implementation mentioned in this document:

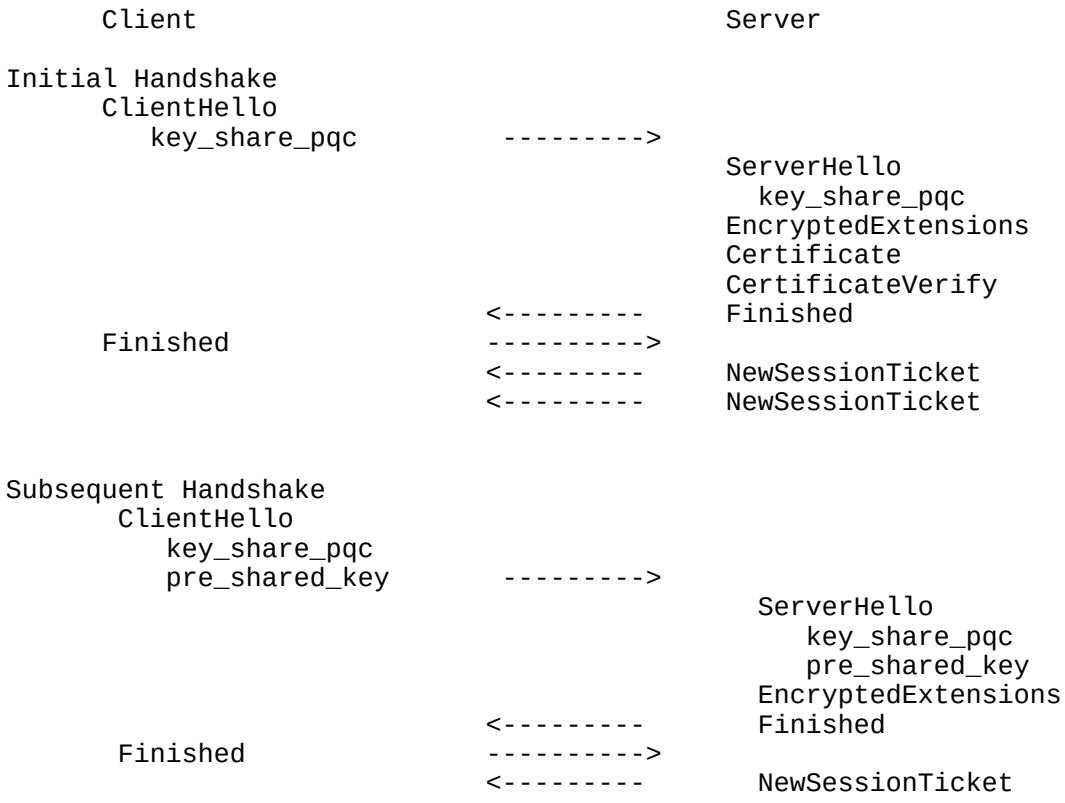



Figure 2: Resumption with "key_share_pqc" extension

### 3.1.2. Hello Retry Request using New Key Share Extension

In a Hello Retry Request scenario, the first ClientHello message will have two algorithms listed in its "supported_groups" extension, where the numerical identifier (NID) for the algorithm that is no longer recognized by the server as an acceptable algorithm will first be listed in this extension, followed by the NID for a Classic McEliece algorithm. In this same ClientHello message is where "02" will be listed in the "psk_key_exchange_modes" extension, and the original "key_share" extension (value 51) is also shown with its public key for the unacceptable algorithm.

When the server responds with the HelloRetryRequest message, the random is the same special value for SHA-256 as indicated in Section 4.1.3 of [RFC8446], and has the same exact fields ("legacy_version", "random", "legacy_session_id_echo", "cipher_suite", "legacy_compression_method", and "extensions") as in the ServerHello structure indicated in [RFC8446] (see section 4.1.3).

The extensions field not only consists of the "supported_versions" extension, but also the new "key_share_pqc" extension where the server offers the client the Classic McEliece algorithm NID it shares with the client.

When the client sends a second ClientHello in response to the HelloRetryRequest, this will be the same message as the first ClientHello with one exception: the original "key_share" extension is replaced with the new "key_share_pqc" extension which contains the large public key of a Classic McEliece algorithm. Then the ServerHello message will then respond containing the new "key_share_pqc" extension.

Therefore, this Hello Retry Request scenario is reflected in Figure 3 below, which is a modification of Figure 2 in [RFC8446], and this can be demonstrated in the TLS Implementation mentioned in this documentation:

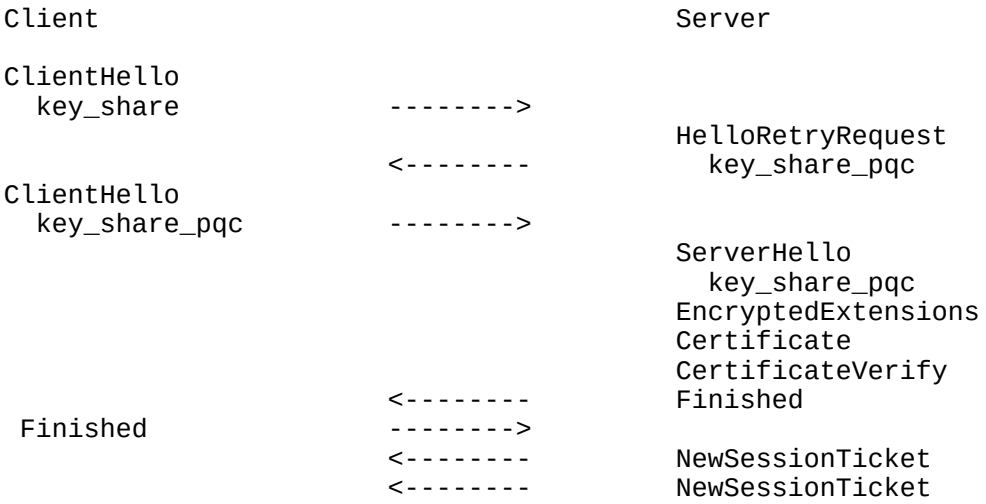



Figure 3: Handshake with HelloRetryRequest with "key_share_pqc" extension

Note: When the client processes the HelloRetryRequest message, it must mark the new "key_share_pqc" extension as an unsolicited extension, which would be an additional exception to the rule noted in [RFC8446] regarding extension responses MUST NOT be sent if the corresponding extension requests were not sent by a remote endpoint (see section 4.2 in [RFC8446]).

The following structure would remain intact from [RFC8446], since support would already be provided for a Classic McEliece algorithm being in NamedGroup (see Section 4):

```
struct {
    NamedGroup selected_group;
} KeyShareHelloRetryRequest;
```

When a Hello Retry Request involves a PSK in use with a Classic McEliece algorithm, both the first and second ClientHello messages (the second one being sent after a HelloRetryRequest message) will contain the exact same content except the first ClientHello will have the original "key_share" extension and the second ClientHello will have the new "key_share_pqc" extension. Another exception includes different binders in both ClientHello messages' pre-shared key extensions. This pre-shared key extension appears as the last extension in both ClientHello messages as well in the ServerHello message.

#### 3.1.3. Other Use Case (RLCE Algorithm)

The Random Linear Code-based Encryption (RLCE) algorithm group (see [RLCE17]) is another code-based cryptographic scheme (NIST Round 1 [NIST1]). "rlcel5" is a RLCE algorithm from this group (where the public key size is 1,232,001 Bytes) that can be used in the new key share extension, and can be demonstrated for use for TLS key exchange in the TLS Implementation mentioned in this document.

#### 3.1.4. Hybrid Combination "x25519classicmceliece348864"

"x25519classicmceliece348864" is a hybrid mechanism introduced in this document that combines both classicmceliece348864 and x25519 [RFC7748] in TLS key exchanges. The experiment TLS implementation presented in this document, which uses the fork [JWYWPROV] of the oqs-provider [OQSPROV], is one example of using x25519classicmceliece348864 in a hybrid key exchange; when x25519classicmceliece348864 is chosen in this circumstance, it uses the "concatenating" method mentioned in [I-D.ietf-tls-hybrid-design] in the new key_share_pqc extension. In the ClientHello message, this new key share extension contains both the Classic McEliece public key and X25519 key concatenated together. In the ServerHello message, this new key share extension then contains the classicmceliece348864 ciphertext and X25519 key concatenated together.

#### 3.1.5. TLS Implementation

A TLS implementation exists that tests the use of a new key share extension for both the ClientHello and ServerHello messages that is implemented for OpenSSL, and also where the mentioned Classic McEliece algorithms can be chosen for key exchange when initiating TLS connections. It can be accessed here: [JWYW25].

#### 3.1.6. Summary of Changes from RFC 8446

A new structure is introduced of KeyShareEntryPQC along with modifications of existing structures including KeyShareEntry, NamedGroup, Extension, ExtensionType, KeyShareClientHello, and KeyShareServerHello. Adding a new ExtensionType of "key_share_pqc" allows for the addition of this new structure of KeyShareEntryPQC, which is based on the existing KeyShareEntry, but "key_exchange" has been expanded and select statements are added to both structures which depend on the KeyShareEntry.group or KeyShareEntryPQC.group being called in a TLS connection for key exchange. This new KeyShareEntryPQC will now also appear in existing structures of KeyShareClientHello and KeyShareServerHello. Thus, the "extension_data" is expanded in the existing Extension structure.

### 3.2. Post-handshake Key Exchange with Extended Key Update

Extended Key Update [I-D.ietf-tls-extended-key-update] is a TLS 1.3 extension that allows to perform post-handshake key exchange in order to update session keys. This mechanism defines new TLS 1.3 handshake message type - ExtendedKeyUpdate. Since TLS 1.3 handshake messages can be up to 2^24 bytes long, this allows to transfer large key shares using this message.

Currently, the functionality of Extended Key Update is limited to only allow using exactly the same key exchange mechanism as was negotiated and used during the handshake. However, the mechanism can be extended to also allow performing a different key exchange mechanism, that could be additionally negotiated during the handshake. In this case a modified Extended Key Update must be run immediately after the initial handshake and before any application data sent over the connection. Thus, the resulting key exchange will always be non-composite hybrid key exchange, similar to what IKEv2 does (see [RFC9370]).

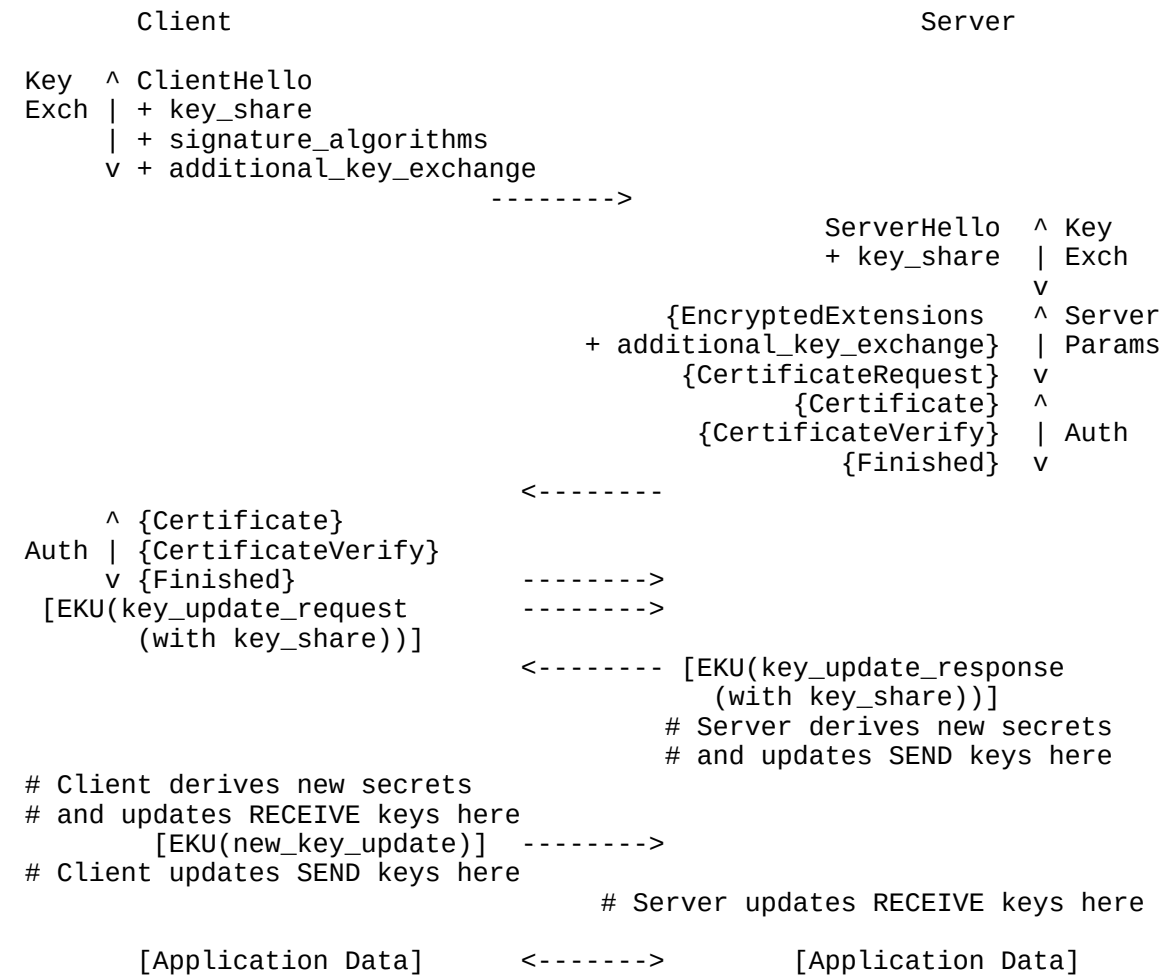



Figure 4: Additional Key Exchange with Extended Key Update

### 3.3. New AuxHandshakeData Handshake Message

If there is a need to send large pieces of data that do not fit into the existing TLS 1.3 handshake messages during the handshake (e.g. in the case of PQ KEM with large public keys, like Classic McEliece) then the client indicates this with new extension of type aux_data (Figure 7) in the ClientHello message. This extension contains no data.

If the server supports this extension, it replies with the HelloRetryRequest that also includes the aux_data extension.

Once HelloRetryRequest has been received, the client repeats ClientHello and immediately after that sends a new message AuxHandshakeData (Figure 5) which actually contains large data. The server then responds with ServerHello that also is immediately followed by AuxHandshakeData message, preceding all other other handshake messages (e.g. EncryptedExtensions, etc.).

The data in the AuxHandshakeData message is organized as an array of AuxHandshakeDataEntry (Figure 4) structures. When a large piece of data should be used in the protocol, it is referenced from the corresponding item in the ClientHello or ServerHello by its index in the AuxHandshakeData message.

For example, with large public keys for some PQ KEMs (like Classic McEliece) the key_share representation would be LargeKeyShareRepresentation (Figure 6), which contains the type of representation and, depending on that type, either the key share itself (e.g. for Classic McEliece ciphertext, which is small) or the index of AuxHandshakeDataEntry data elements in the AuxHandshakeData message, which will contain the large key share (e.g. a Classic McEliece public key).

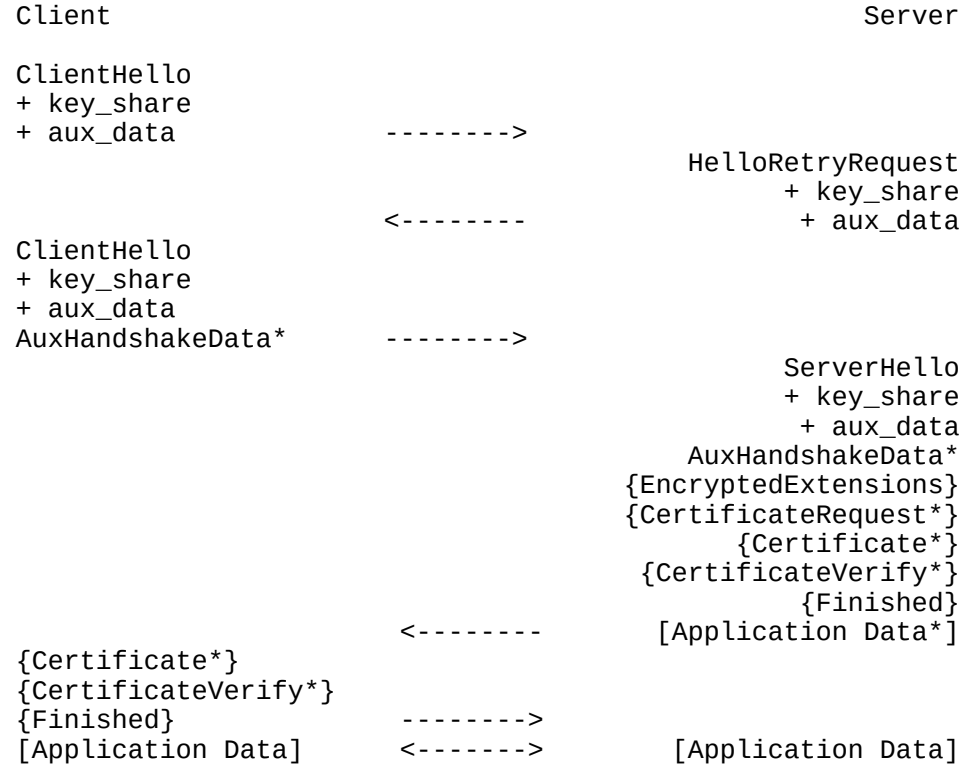


Figure 5: Using the AuxHandshakeData Message in TLS Handshake

```
enum {
    ...
    aux_handshake_data(TBA),
    (65535)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (Handshake.msg_type) {
        ...
        case aux_handshake_data:    AuxHandshakeData;
    };
} Handshake;
```

Figure 6: Definition of AuxHandshakeData

```
  struct {
      opaque  data<0..2^24-1>;
  } AuxHandshakeDataEntry;

  struct {
      AuxHandshakeDataEntry   aux_data<0..2^24-1>;
  } AuxHandshakeData;

struct {
    uint8 form;
    select (LargeKeyShareRepresentation.type) {
        case 0:     uint16      aux_data_entry_index;
        default:    opaque      key_exchange<0..2^16-1>;
    };
} LargeKeyShareRepresentation;

enum {
        ...
        aux_data(TBD),
        (65535)
    } ExtensionType;

struct {
} AuxData;
```

Figure 7: Format of AuxHandshakeData

## 4. Analyzing of the Proposed Solutions

This document proposes three possible solutions for transferring large amounts of data during the TLS 1.3 handshake.

### 1. New Key Share Extension

This is a straightforward solution to the problem, it changes the format of the ClientHello and the ServerHello messages in a non-backward compatible way.

Advantages:

* It is the most efficient solution in terms of round trips - the number of round trips needed to establish the TLS connection does not increase.

Disadvantages:

* It can only be used in environments when clients know beforehand that servers they contact support this extension.

* It deals only with key shares, thus it is not a generic solution to transferring large data in handshake.

* Since the format of the ClientHello is changed, it is unclear how this extension will interact with Encrypted ClientHello extension.

* It is not clear how middleboxes will handle modified ClientHello and ServerHello

### 2. Modified Extended Key Update

Advantages:

* This solution keeps the current TLS 1.3 handshake intact, thus making it friendly to middleboxes.

Disadvantages:

* The number of round trips needed before application data can be sent increases.

* It complicates the TLS state machine - application data should not be sent once the initial handshake is complete, instead it can only be sent after the modified Extended Key Update immediately following the initial handshake completes.

* It deals only with key shares, thus it is not a generic solution to transferring large data in a handshake.

* It is unclear how this would interact with regular Extended Key Update extension either only initial key exchange algorithm is used for rekey using Extended Key Update, or this extension neede to be modified to be able to perform several successive key exchanges (similar to [RFC9370]).

3. New AuxHandshakeData Handshake Message

Advantages:

* This solution keeps the current ClientHello and ServerHello messages intact, but adds a new handshake message following them. It seems that this is more friendly to middleboxes than modifying the format of CH and SH, but this is not for sure.

* This is a generic solution, allowing to transfer large data of any kind in a TLS handshake.

* Since the ClientHello format remains the same, it seems that this solution can be used with ECH (requires more investigations).

Disadvantages:

* The solution relies on HelloRetryRequest, thus the number of round trips needed to complete a handshake increases.

Acknowledgements

Thank you D. J. Bernstein and Simon Josefsson as they advised to have at least one reference for the description of Classic McEliece. Thank you also to Eliot Lear for his feedback on other fields regarding the next algorithm needed.

Thank you as well to Martin Thomson and David Schinazi, as their Internet Draft template was used to generate this document, before the authors' information was added. The authors also want to thank the contributors of the kramdown-rfc GitHub repository, as their examples helped with the format of the figures, references, and authors' information presented in this document. Thank you also to Joyce Reynolds and Robert Braden, as their Internet Draft [JR04] was helpful as a guide on how to write the citations in this document (i.e., using citation brackets with author's initials, year, etc.).

Using Extended Key Update mechanism for transferring large key shares was proposed by John Mattsson.

## References

### Normative References

[I-D.josefsson-mceliece]
Josefsson, S., "Classic McEliece", Work in Progress, Internet-Draft, draft-josefsson-mceliece-03, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-josefsson-mceliece-03>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

### Informative References

[DJB25] Bernstein, D., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Sendrier, N., Szefer, J., Tjhai, C., Tomlinson, M., and W. Wang, "Classic McEliece: Implementation", 2024, <https://classic.mceliece.org/impl.html>.

[EXT652] Wagner, J., "ssl/statem/extensions.c#L652C9-L663C9", 2024, <https://github.com/jwagrunner/openssl/blob/master/ssl/statem/extensions.c#L652C9-L663C9>.

[I-D.ietf-tls-extended-key-update]
Tschofenig, H., T端xen, M., Reddy.K, T., Fries, S., and Y. Rosomakho, "Extended Key Update for Transport Layer Security (TLS) 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-extended-key-update-09, 18 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-extended-key-update-09>.

[I-D.ietf-tls-hybrid-design]
Stebila, D., Fluhrer, S., and S. Gueron, "Hybrid key exchange in TLS 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-hybrid-design-16, 7 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-hybrid-design-16>.

[JR04] Reynolds, J. and R. Braden, "Instructions to Request for Comments (RFC) Authors", 2004, <https://www.rfc-editor.org/old/instructions2authors.txt>.

[JWYW25] Wagner, J. and Y. Wang, "openssl", 2025, <https://github.com/jwagrunner/openssl>.

[JWYWPROV] Wagner, J. and Y. Wang, "oqs-provider", 2025, <https://github.com/jwagrunner/oqs-provider>.

[NIST] Bernstein, D., Chou, T., Cid, C., Gilcher, J., Lange, T., Maram, V., von Maurich, I., Misoczki, R., Niederhagen, R., Persichetti, E., Peters, C., Sendrier, N., Szefer, J., Tjhai, C., Tomlinson, M., and W. Wang, "Classic McEliece", 2025, <https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions>.

[NIST1] Wang, Y., "RLCE-KEM", 2025, <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-1-submissions>.

[OQS24] Open Quantum Safe, "liboqs / Algorithms / Classic McEliece", 2024, <https://openquantumsafe.org/liboqs/algorithms/kem/classic_mceliece>.

[OQSPROV] Project, O. Q. S., "OQS Provider for OpenSSL 3", July 2023, <https://github.com/open-quantum-safe/oqs-provider/>.

[PQC25] NIST, "Post-Quantum Cryptography: Round 4 Submissions", 2025, <https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions>.

[RASHOK20] rashok, "How to do TLS 1.3 PSK using openssl?", 2020, <https://stackoverflow.com/questions/58719595/how-to-do-tls-1-3-psk-using-openssl>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[RFC9370] Tjhai, CJ., Tomlinson, M., Bartlett, G., Fluhrer, S., Van Geest, D., Garcia-Morchon, O., and V. Smyslov, "Multiple Key Exchanges in the Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 9370, DOI 10.17487/RFC9370, May 2023, <https://www.rfc-editor.org/rfc/rfc9370>.

[RJM78] McEliece, R., "A Public-Key Cryptosystem Based On Algebraic Coding Theory", 1978, <https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF>.

[RLCE17] Wang, Y., "Quantum Resistant Public Key Encryption Scheme RLCE and IND-CCA2 Security for McEliece Schemes", 2017, <https://eprint.iacr.org/2017/206.pdf>.

[SRVR1211] Wagner, J., "ssl/statem/statem_srvr.c#L1211", 2024, <https://github.com/jwagrunner/openssl/blob/master/ssl/statem/statem_srvr.c#L1211>.

[SRVR1650] Wagner, J., "ssl/statem/statem_srvr.c#L1650", 2024, <https://github.com/jwagrunner/openssl/blob/master/ssl/statem/statem_srvr.c#L1650>.

## Authors' Addresses

Jonathan Wagner
UNC Charlotte
9201 University City Blvd
Charlotte, NC, 28223
United States of America
Email: jwagne31@charlotte.edu

Yongge Wang
UNC Charlotte
9201 University City Blvd
Charlotte, NC, 28223
United States of America
Email: yongge.wang@charlotte.edu

Valery Smyslov
ELVIS-PLUS
PO Box 81

Moscow (Zelenograd)
124460
Russian Federation
Email: svan@elvis.ru

Yoav Nir
Dell Technologies
9 Andrei Sakharov St
Haifa 3190500
Israel
Email: ynir.ietf@gmail.com