Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 9, 2020

D. Stebila
University of Waterloo
S. Fluhrer
Cisco Systems
S. Gueron
U. Haifa, Amazon Web Services
July 08, 2019

Design issues for hybrid key exchange in TLS 1.3
draft-stebila-tls-hybrid-design-01

Abstract

Hybrid key exchange refers to using multiple key exchange algorithms simultaneously and combining the result with the goal of providing security even if all but one of the component algorithms is broken, and is motivated by transition to post-quantum cryptography. This document categorizes various design considerations for using hybrid key exchange in the Transport Layer Security (TLS) protocol version 1.3 and outlines two concrete instantiations for consideration.

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/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2020.

Copyright Notice

Table of Contents

1. Introduction

This document categorizes various design decisions one could make when implementing hybrid key exchange in TLS 1.3, with the goal of fostering discussion, providing options for short-term prototypes/ experiments, and serving as a basis for eventual standardization. This document also includes two concrete instantiations for consideration, following two different approaches; it is not our intention that both be standardized.

This document does not propose specific post-quantum mechanisms; see Section 1.4 for more on the scope of this document.

Comments are solicited and should be addressed to the TLS working group mailing list at tls@ietf.org and/or the author(s).

1.1. Revision history

- draft-00: Initial version.
- draft-01:
  - Add (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) options.
  - Add Candidate Instantiation 1 (Section 4.1).

* Add Candidate Instantiation 2 (Section 4.2).

1.2. Terminology

For the purposes of this document, it is helpful to be able to divide cryptographic algorithms into two classes:

- "Traditional" algorithms: Algorithms which are widely deployed today, but which may be deprecated in the future. In the context of TLS 1.3 in 2019, examples of traditional key exchange algorithms include elliptic curve Diffie-Hellman using secp256r1 or x25519, or finite-field Diffie-Hellman.

- "Next-generation" (or "next-gen") algorithms: Algorithms which are not yet widely deployed, but which may eventually be widely deployed. An additional facet of these algorithms may be that we have less confidence in their security due to them being relatively new or less studied. This includes "post-quantum" algorithms.

"Hybrid" key exchange, in this context, means the use of two (or more) key exchange mechanisms based on different cryptographic assumptions (for example, one traditional algorithm and one next-gen algorithm), with the purpose of the final session key being secure as long as at least one of the component key exchange mechanisms remains unbroken. We use the term "component" algorithms to refer to the algorithms that are being combined in a hybrid key exchange.

The primary motivation of this document is preparing for post-quantum algorithms. However, it is possible that public key cryptography based on alternative mathematical constructions will be required independent of the advent of a quantum computer, for example because of a cryptanalytic breakthrough. As such we opt for the more generic term "next-generation" algorithms rather than exclusively "post-quantum" algorithms.

1.3. Motivation for use of hybrid key exchange

Ideally, one would not use hybrid key exchange: one would have confidence in a single algorithm and parameterization that will stand the test of time. However, this may not be the case in the face of quantum computers and cryptanalytic advances more generally.

Many (but not all) of the post-quantum algorithms currently under consideration are relatively new; they have not been subject to the same depth of study as RSA and finite-field / elliptic curve Diffie-Hellman, and thus we do not necessarily have as much confidence in

their fundamental security, or the concrete security level of specific parameterizations.

Early adopters eager for post-quantum security may want to use hybrid key exchange to have the potential of post-quantum security from a less-well-studied algorithm while still retaining at least the security currently offered by traditional algorithms. (They may even need to retain traditional algorithms due to regulatory constraints, for example FIPS compliance.)

Moreover, it is possible that even by the end of the NIST Post-Quantum Cryptography Standardization Project, and for a period of time thereafter, conservative users may not have full confidence in some algorithms.

As such, there may be users for whom hybrid key exchange is an appropriate step prior to an eventual transition to next-generation algorithms.

1.4. Scope

This document focuses on hybrid ephemeral key exchange in TLS 1.3 [TLS13]. It intentionally does not address:

- Selecting which next-generation algorithms to use in TLS 1.3, nor algorithm identifiers nor encoding mechanisms for next-generation algorithms. (The outcomes of the NIST Post-Quantum Cryptography Standardization Project [NIST] will inform this choice.)

- Authentication using next-generation algorithms. (If a cryptographic assumption is broken due to the advent of a quantum computer or some other cryptanalytic breakthrough, confidentiality of information can be broken retroactively by any adversary who has passively recorded handshakes and encrypted communications. But session authentication cannot be retroactively broken.)

1.5. Goals

The primary goal of a hybrid key exchange mechanism is to facilitate the establishment of a shared secret which remains secure as long as as one of the component key exchange mechanisms remains unbroken.

In addition to the primary cryptographic goal, there may be several additional goals in the context of TLS 1.3:

- *Backwards compatibility:* Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid key exchange standard is developed for TLS, should remain compatible with endpoints and

middle-boxes that are not hybrid-aware. The three scenarios to consider are:

1. Hybrid-aware client, hybrid-aware server: These parties should establish a hybrid shared secret.

2. Hybrid-aware client, non-hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware client is willing to downgrade to traditional-only).

3. Non-hybrid-aware client, hybrid-aware server: These parties should establish a traditional shared secret (assuming the hybrid-aware server is willing to downgrade to traditional-only).

Ideally backwards compatibility should be achieved without extra round trips and without sending duplicate information; see below.

- *High performance:* Use of hybrid key exchange should not be prohibitively expensive in terms of computational performance. In general this will depend on the performance characteristics of the specific cryptographic algorithms used, and as such is outside the scope of this document. See [BCNS15], [CECPQ1], [FRODO] for preliminary results about performance characteristics.

- *Low latency:* Use of hybrid key exchange should not substantially increase the latency experienced to establish a connection. Factors affecting this may include the following.

  * The computational performance characteristics of the specific algorithms used. See above.

  * The size of messages to be transmitted. Public key / ciphertext sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantially. See [BCNS15], [FRODO] for preliminary results in a laboratory setting, and [LANGLEY] for preliminary results on more realistic networks.

  * Additional round trips added to the protocol. See below.

- *No extra round trips:* Attempting to negotiate hybrid key exchange should not lead to extra round trips in any of the three hybrid-aware/non-hybrid-aware scenarios listed above.

- *No duplicate information:* Attempting to negotiate hybrid key exchange should not mean having to send multiple public keys of the same type.

1.6. Related work

Quantum computing and post-quantum cryptography in general are outside the scope of this document. For a general introduction to quantum computing, see a standard textbook such as [NIELSEN]. For an overview of post-quantum cryptography as of 2009, see [BERNSTEIN]. For the current status of the NIST Post-Quantum Cryptography Standardization Project, see [NIST]. For additional perspectives on the general transition from classical to post-quantum cryptography, see for example [ETSI] and [HOFFMAN], among others.

There have been several Internet-Drafts describing mechanisms for embedding post-quantum and/or hybrid key exchange in TLS:

- Internet-Drafts for TLS 1.2: [WHYTE12]

- Internet-Drafts for TLS 1.3: [KIEFER], [SCHANCK], [WHYTE13]

There have been several prototype implementations for post-quantum and/or hybrid key exchange in TLS:

- Experimental implementations in TLS 1.2: [BCNS15], [CECPQ1], [FRODO], [OQS-102]

- Experimental implementations in TLS 1.3: [CECPQ2], [OQS-111]

These experimental implementations have taken an ad hoc approach and not attempted to implement one of the drafts listed above.

Unrelated to post-quantum but still related to the issue of combining multiple types of keying material in TLS is the use of pre-shared keys, especially the recent TLS working group document on including an external pre-shared key [EXTERN-PSK].

Considering other IETF standards, there is work on post-quantum preshared keys in IKEv2 [IKE-PSK] and a framework for hybrid key exchange in IKEv2 [IKE-HYBRID]. The XMSS hash-based signature scheme has been published as an informational RFC by the IRTF [XMSS].

In the academic literature, [EVEN] initiated the study of combining multiple symmetric encryption schemes; [ZHANG], [DODIS], and [HARNIK] examined combining multiple public key encryption schemes, and [HARNIK] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while

preserving security properties. [GIACON] and [BINDEL] examined combining multiple key encapsulation mechanisms.

2. Overview

We identify four distinct axes along which one can make choices when integrating hybrid key exchange into TLS 1.3:

1. How to negotiate the use of hybridization in general and component algorithms specifically?

2. How many component algorithms can be combined?

3. How should multiple key shares (public keys / ciphertexts) be conveyed?

4. How should multiple shared secrets be combined?

The remainder of this document outlines various options we have identified for each of these choices. Immediately below we provide a summary list. Options are labelled with a short code in parentheses to provide easy cross-referencing.

1. (Neg) (Section 3.1) How to negotiate the use of hybridization in general and component algorithms specifically?

    * (Neg-Ind) (Section 3.1.2) Negotiating component algorithms individually

        + (Neg-Ind-1) (Section 3.1.2.1) Traditional algorithms in "ClientHello" "supported_groups" extension, next-gen algorithms in another extension

        + (Neg-Ind-2) (Section 3.1.2.2) Both types of algorithms in "supported_groups" with external mapping to tradition/next-gen.

        + (Neg-Ind-3) (Section 3.1.2.3) Both types of algorithms in "supported_groups" separated by a delimiter.

    * (Neg-Comb) (Section 3.1.3) Negotiating component algorithms as a combination

        + (Neg-Comb-1) (Section 3.1.3.1) Standardize "NamedGroup" identifiers for each desired combination.

    + (Neg-Comb-2) (Section 3.1.3.2) Use placeholder identifiers in "supported_groups" with an extension defining the combination corresponding to each placeholder.

    + (Neg-Comb-3) (Section 3.1.3.3) List combinations by inserting grouping delimiters into "supported_groups" list.

2. (Num) (Section 3.2) How many component algorithms can be combined?

    * (Num-2) (Section 3.2.1) Two.

    * (Num-2+) (Section 3.2.2) Two or more.

3. (Shares) (Section 3.3) How should multiple key shares (public keys / ciphertexts) be conveyed?

    * (Shares-Concat) (Section 3.3.1) Concatenate each combination of key shares.

    * (Shares-Multiple) (Section 3.3.2) Send individual key shares for each algorithm.

    * (Shares-Ext-Additional) (Section 3.3.3) Use an extension to convey key shares for component algorithms.

4. (Comb) (Section 3.4) How should multiple shared secrets be combined?

    * (Comb-Concat) (Section 3.4.1) Concatenate the shared secrets then use directly in the TLS 1.3 key schedule.

    * (Comb-KDF-1) (Section 3.4.2) and (Comb-KDF-2) (Section 3.4.3) KDF the shared secrets together, then use the output in the TLS 1.3 key schedule.

    * (Comb-XOR) (Section 3.4.4) XOR the shared secrets then use directly in the TLS 1.3 key schedule.

    * (Comb-Chain) (Section 3.4.5) Extend the TLS 1.3 key schedule so that there is a stage of the key schedule for each shared secret.

    * (Comb-AltInput) (Section 3.4.6) Use the second shared secret in an alternate (otherwise unused) input in the TLS 1.3 key schedule.

3. Design options

3.1. (Neg) How to negotiate hybridization and component algorithms?

3.1.1. Key exchange negotiation in TLS 1.3

Recall that in TLS 1.3, the key exchange mechanism is negotiated via the "supported_groups" extension. The "NamedGroup" enum is a list of standardized groups for Diffie-Hellman key exchange, such as "secp256r1", "x25519", and "ffdhe2048".

The client, in its "ClientHello" message, lists its supported mechanisms in the "supported_groups" extension. The client also optionally includes the public key of one or more of these groups in the "key_share" extension as a guess of which mechanisms the server might accept in hopes of reducing the number of round trips.

If the server is willing to use one of the client's requested mechanisms, it responds with a "key_share" extension containing its public key for the desired mechanism.

If the server is not willing to use any of the client's requested mechanisms, the server responds with a "HelloRetryRequest" message that includes an extension indicating its preferred mechanism.

3.1.2. (Neg-Ind) Negotiating component algorithms individually

In these three approaches, the parties negotiate which traditional algorithm and which next-gen algorithm to use independently. The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm.

3.1.2.1. (Neg-Ind-1)

The client advertises two lists to the server: one list containing its supported traditional mechanisms (e.g. via the existing "ClientHello" "supported_groups" extension), and a second list containing its supported next-generation mechanisms (e.g., via an additional "ClientHello" extension). A server could then select one algorithm from the traditional list, and one algorithm from the next-generation list. (This is the approach in [SCHANCK].)

3.1.2.2. (Neg-Ind-2)

The client advertises a single list to the server which contains both its traditional and next-generation mechanisms (e.g., all in the existing "ClientHello" "supported_groups" extension), but with some external table provides a standardized mapping of those mechanisms as

either "traditional" or "next-generation". A server could then select two algorithms from this list, one from each category.

3.1.2.3. (Neg-Ind-3)

The client advertises a single list to the server delimited into sublists: one for its traditional mechanisms and one for its next-generation mechanisms, all in the existing "ClientHello" "supported_groups" extension, with a special code point serving as a delimiter between the two lists. For example, "supported_groups = secp256r1, x25519, delimiter, nextgen1, nextgen4".

3.1.3. (Neg-Comb) Negotiating component algorithms as a combination

In these three approaches, combinations of key exchange mechanisms appear as a single monolithic block; the parties negotiate which of several combinations they wish to use.

3.1.3.1. (Neg-Comb-1)

The "NamedGroup" enum is extended to include algorithm identifiers for each *combination* of algorithms desired by the working group. There is no "internal structure" to the algorithm identifiers for each combination, they are simply new code points assigned arbitrarily. The client includes any desired combinations in its "ClientHello" "supported_groups" list, and the server picks one of these. This is the approach in [KIEFER] and [OQS-111].

3.1.3.2. (Neg-Comb-2)

The "NamedGroup" enum is extended to include algorithm identifiers for each next-gen algorithm. Some additional field/extension is used to convey which combinations the parties wish to use. For example, in [WHYTE13], there are distinguished "NamedGroup" called "hybrid_marker 0", "hybrid_marker 1", "hybrid_marker 2", etc. This is complemented by a "HybridExtension" which contains mappings for each numbered "hybrid_marker" to the set of component key exchange algorithms (2 or more) for that proposed combination.

3.1.3.3. (Neg-Comb-3)

The client lists combinations in "supported_groups" list, using a special delimiter to indicate combinations. For example, "supported_groups = combo_delimiter, secp256r1, nextgen1, combo_delimiter, secp256r1, nextgen4, standalone_delimiter, secp256r1, x25519" would indicate that the client's highest preference is the combination secp256r1+nextgen1, the next highest preference is the combination secp2561+nextgen4, then the single

algorithm secp256r1, then the single algorithm x25519. A hybrid-aware server would be able to parse these; a hybrid-unaware server would see "unknown, secp256r1, unknown, unknown, secp256r1, unknown, unknown, secp256r1, x25519", which it would be able to process, although there is the potential that every "projection" of a hybrid list that is tolerable to a client does not result in list that is tolerable to the client.

3.1.4. Benefits and drawbacks

Combinatorial explosion. (Neg-Comb-1) (Section 3.1.3.1) requires new identifiers to be defined for each desired combination. The other 4 options in this section do not.

Extensions. (Neg-Ind-1) (Section 3.1.2.1) and (Neg-Comb-2) (Section 3.1.3.2) require new extensions to be defined. The other options in this section do not.

New logic. All options in this section except (Neg-Comb-1) (Section 3.1.3.1) require new logic to process negotiation.

Matching security levels. (Neg-Ind-1) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), and (Neg-Comb-2) (Section 3.1.3.2) allow algorithms of different claimed security level from their corresponding lists to be combined. For example, this could result in combining ECDH secp256r1 (classical security level 128) with NewHope-1024 (classical security level 256). Implementations dissatisfied with a mismatched security levels must either accept this mismatch or attempt to renegotiate. (Neg-Ind-1) (Section 3.1.2.1), (Neg-Ind-2) (Section 3.1.2.2), and (Neg-Ind-3) (Section 3.1.2.3) give control over the combination to the server; (Neg-Comb-2) (Section 3.1.3.2) gives control over the combination to the client. (Neg-Comb-1) (Section 3.1.3.1) only allows standardized combinations, which could be set by TLS working group to have matching security (provided security estimates do not evolve separately).

Backwards-compability. TLS 1.3-compliant hybrid-unaware servers should ignore unreocgnized elements in "supported_groups" (Neg-Ind-2) (Section 3.1.2.2), (Neg-Ind-3) (Section 3.1.2.3), (Neg-Comb-1) (Section 3.1.3.1), (Neg-Comb-2) (Section 3.1.3.2) and unrecognized "ClientHello" extensions (Neg-Ind-1) (Section 3.1.2.1), (Neg-Comb-2) (Section 3.1.3.2). In (Neg-Ind-3) (Section 3.1.2.3) and (Neg-Comb-3) (Section 3.1.3.3), a server that is hybrid-unaware will ignore the delimiters in "supported_groups", and thus might try to negotiate an algorithm individually that is only meant to be used in combination; depending on how such an implementation is coded, it may also

encounter bugs when the same element appears multiple times in the list.

3.2. (Num) How many component algorithms to combine?

3.2.1. (Num-2) Two

Exactly two algorithms can be combined together in hybrid key exchange. This is the approach taken in [KIEFER] and [SCHANCK].

3.2.2. (Num-2+) Two or more

Two or more algorithms can be combined together in hybrid key exchange. This is the approach taken in [WHYTE13].

3.2.3. Benefits and Drawbacks

Restricting the number of component algorithms that can be hybridized to two substantially reduces the generality required. On the other hand, some adopters may want to further reduce risk by employing multiple next-gen algorithms built on different cryptographic assumptions.

3.3. (Shares) How to convey key shares?

In ECDH ephmeral key exchange, the client sends its ephmeral public key in the "key_share" extension of the "ClientHello" message, and the server sends its ephmeral public key in the "key_share" extension of the "ServerHello" message.

For a general key encapsulation mechanism used for ephemeral key exchange, we imagine that that client generates a fresh KEM public key / secret pair for each connection, sends it to the client, and the server responds with a KEM ciphertext. For simplicity and consistency with TLS 1.3 terminology, we will refer to both of these types of objects as "key shares".

In hybrid key exchange, we have to decide how to convey the client's two (or more) key shares, and the server's two (or more) key shares.

3.3.1. (Shares-Concat) Concatenate key shares

The client concatenates the bytes representing its two key shares and uses this directly as the "key_exchange" value in a "KeyShareEntry" in its "key_share" extension. The server does the same thing. Note that the "key_exchange" value can be an octet string of length at most 2^16-1. This is the approach taken in [KIEFER], [OQS-111], and [WHYTE13].

3.3.2. (Shares-Multiple) Send multiple key shares

The client sends multiple key shares directly in the "client_shares" vectors of the "ClientHello" "key_share" extension. The server does the same. (Note that while the existing "KeyShareClientHello" struct allows for multiple key share entries, the existing "KeyShareServerHello" only permits a single key share entry, so some modification would be required to use this approach for the server to send multiple key shares.)

3.3.3. (Shares-Ext-Additional) Extension carrying additional key shares

The client sends the key share for its traditional algorithm in the original "key_share" extension of the "ClientHello" message, and the key share for its next-gen algorithm in some additional extension in the "ClientHello" message. The server does the same thing. This is the approach taken in [SCHANCK].

3.3.4. Benefits and Drawbacks

Backwards compatibility. (Shares-Multiple) (Section 3.3.2) is fully backwards compatible with non-hybrid-aware servers. (Shares-Ext-Additional) (Section 3.3.3) is backwards compatible with non-hybrid-aware servers provided they ignore unrecognized extensions. (Shares-Concat) (Section 3.3.1) is backwards-compatible with non-hybrid aware servers, but may result in duplication / additional round trips (see below).

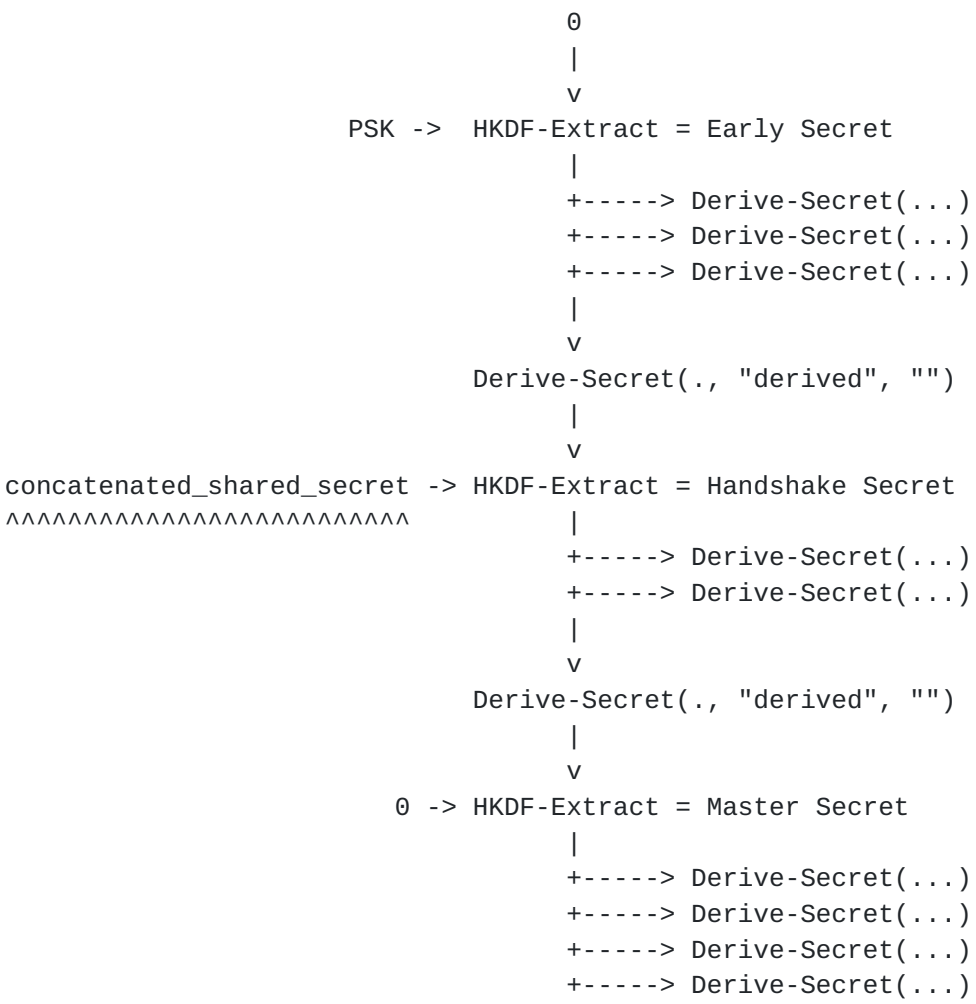
Duplication versus additional round trips. If a client wants to offer multiple key shares for multiple combinations in order to avoid retry requests, then the client may ended up sending a key share for one algorithm multiple times when using (Shares-Ext-Additional) (Section 3.3.3) and (Shares-Concat) (Section 3.3.1). (For example, if the client wants to send an ECDH-secp256r1 + McEliece123 key share, and an ECDH-secp256r1 + NewHope1024 key share, then the same ECDH public key may be sent twice. If the client also wants to offer a traditional ECDH-only key share for non-hybrid-aware implementations and avoid retry requests, then that same ECDH public key may be sent another time.) (Shares-Multiple) (Section 3.3.2) does not result in duplicate key shares.

3.4. (Comb) How to use keys?

Each component key exchange algorithm establishes a shared secret. These shared secrets must be combined in some way that achieves the "hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken.

3.4.1. (Comb-Concat) Concatenate keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the concatenated shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:



This is the approach used in [KIEFER], [OQS-111], and [WHYTE13].

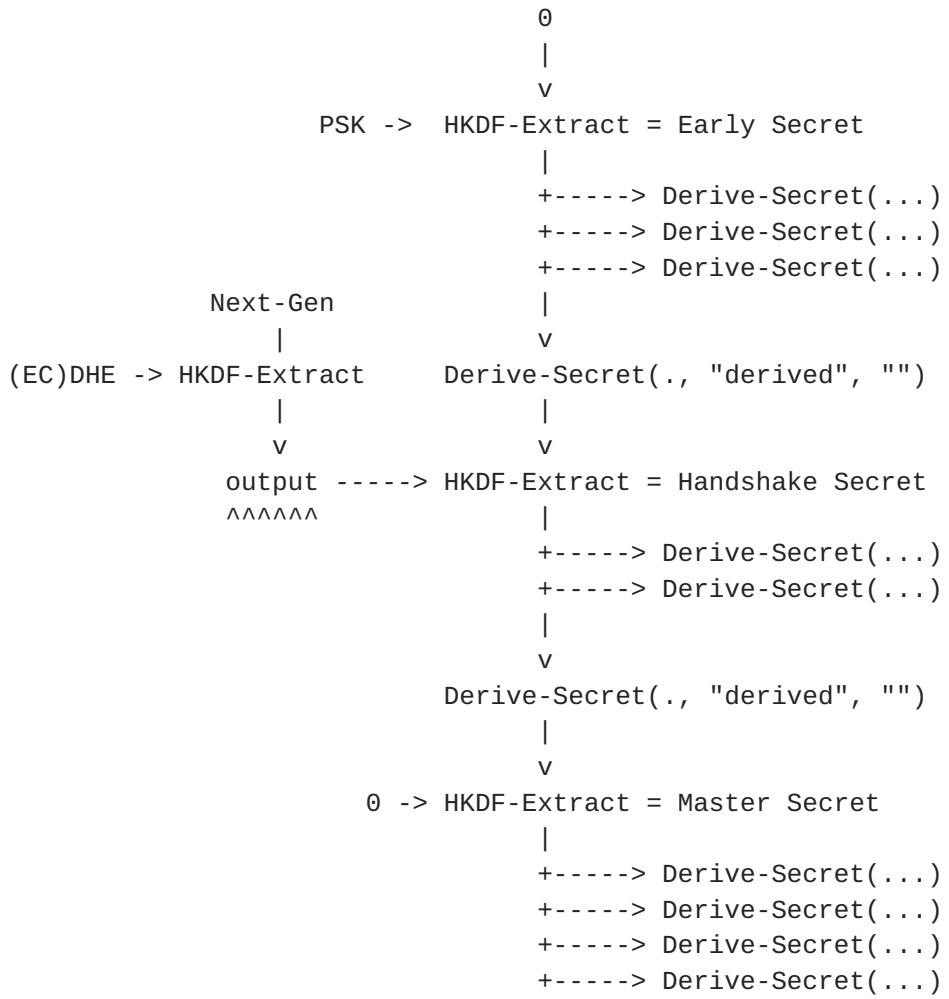
[GIACON] analyzes the security of applying a KDF to concatenated KEM shared secrets, but their analysis does not exactly apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

[BINDEL] analyzes the security of the (Comb-Concat) approach as abstracted in their "dualPRF" combiner. They show that, if the component KEMs are IND-CPA-secure (or IND-CCA-secure), then the values output by "Derive-Secret" are IND-CPA-secure (respectively,

IND-CCA-secure). An important aspect of their analysis is that each ciphertext is input to the final PRF calls; this holds for TLS 1.3 since the "Derive-Secret" calls that derive output keys (application traffic secrets, and exporter and resumption master secrets) include the transcript hash as input.

3.4.2. (Comb-KDF-1) KDF keys

Each party feeds the shared secrets established by each component algorithm in an agreed-upon order into a KDF, then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean first applying "HKDF-Extract" to the shared secrets, then using the output in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule:

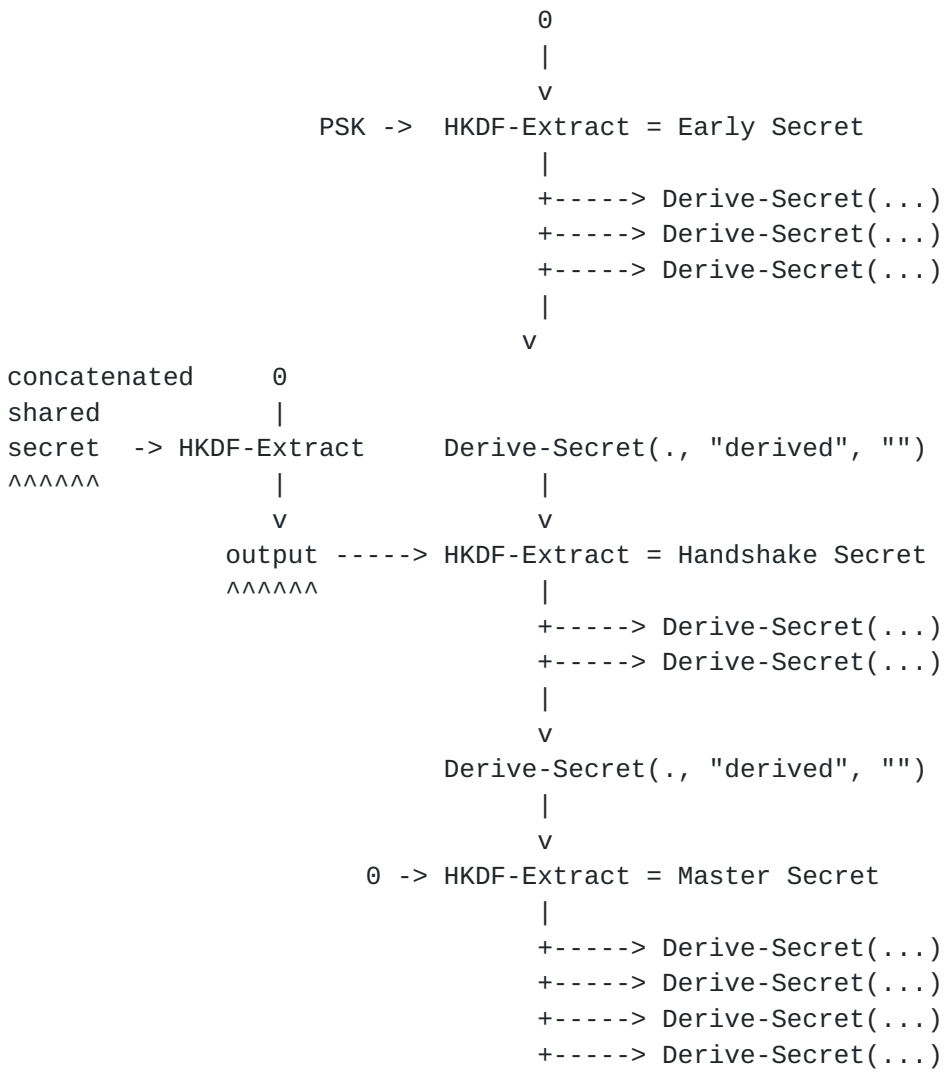


3.4.3. (Comb-KDF-2) KDF keys

Each party concatenates the shared secrets established by each component algorithm in an agreed-upon order then feeds that into a KDF, then feeds the result through the TLS key schedule.

Compared with (Comb-KDF-1) (Section 3.4.2), this method concatenates the (2 or more) shared secrets prior to input to the KDF, whereas (Comb-KDF-1) puts the (exactly 2) shared secrets in the two different input slots to the KDF.

Compared with (Comb-Concat) (Section 3.4.1), this method has an extract KDF application. While this adds computational overhead, this may provide a cleaner abstraction of the hybridization mechanism for the purposes of formal security analysis.



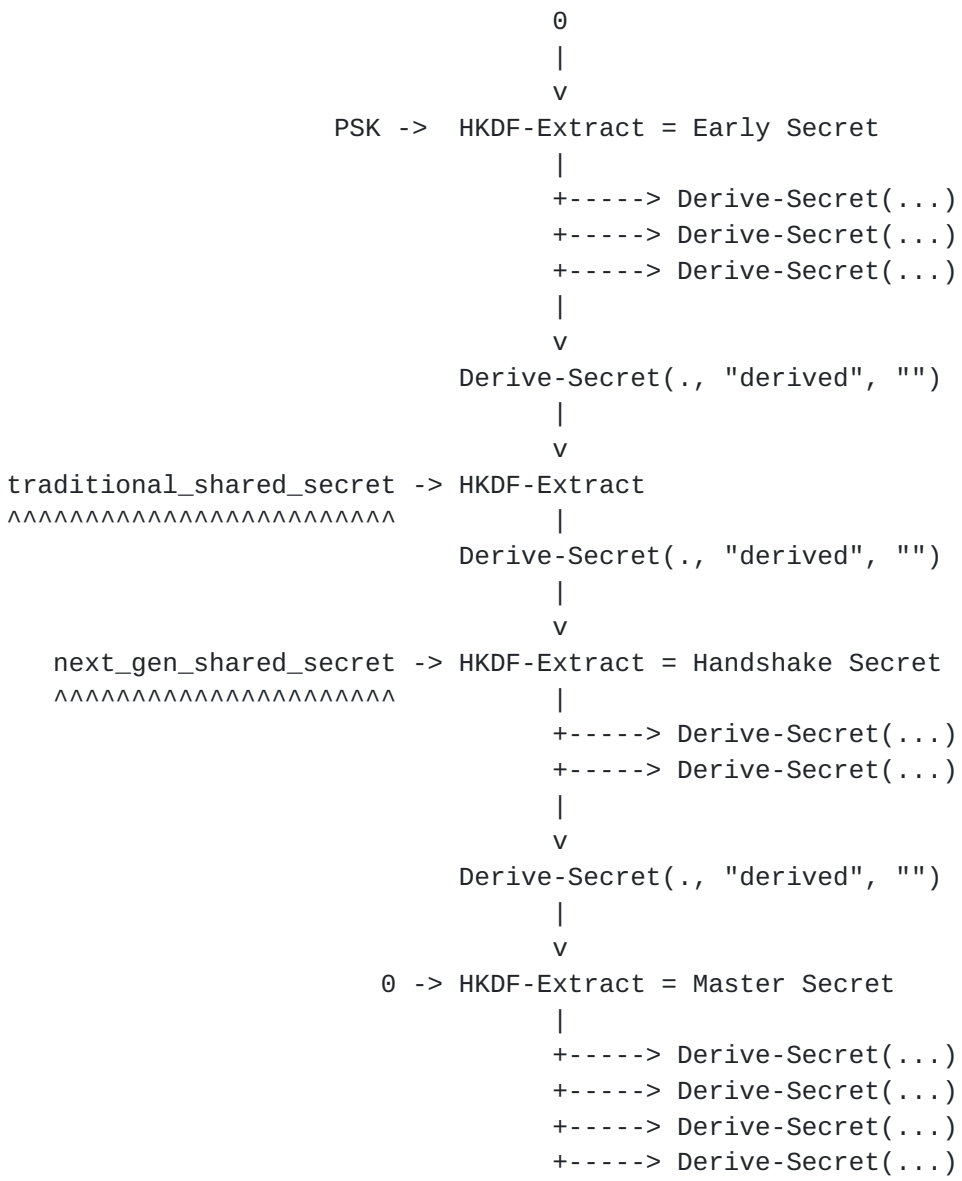
3.4.4. (Comb-XOR) XOR keys

Each party XORs the shared secrets established by each component algorithm (possibly after padding secrets of different lengths), then feeds that through the TLS key schedule. In the context of TLS 1.3, this would mean using the XORed shared secret in place of the (EC)DHE input to the second call to "HKDF-Extract" in the TLS 1.3 key schedule.

[GIACON] analyzes the security of applying a KDF to the XORed KEM shared secrets, but their analysis does not quite apply here since the transcript of ciphertexts is included in the KDF application (though it should follow relatively straightforwardly).

3.4.5. (Comb-Chain) Chain of KDF applications for each key

Each party applies a chain of key derivation functions to the shared secrets established by each component algorithm in an agreed-upon order; roughly speaking: "F(k1 || F(k2))". In the context of TLS 1.3, this would mean extending the key schedule to have one round of the key schedule applied for each component algorithm's shared secret:

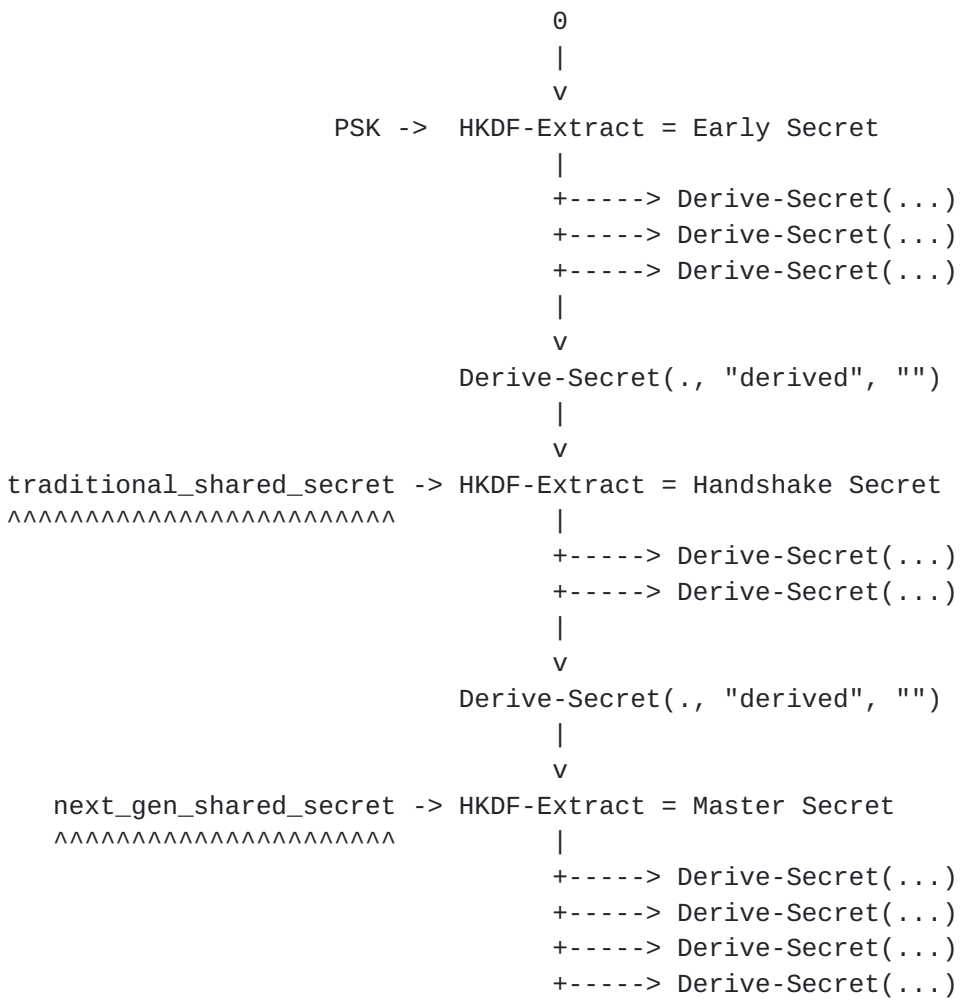


This is the approach used in [SCHANCK].

[BINDEL] analyzes the security of this approach as abstracted in their nested dual-PRF "N" combiner, showing a similar result as for the dualPRF combiner that it preserves IND-CPA (or IND-CCA) security. Again their analysis depends on each ciphertext being input to the final PRF ("Derive-Secret") calls, which holds for TLS 1.3.

3.4.6. (Comb-AltInput) Second shared secret in an alternate KDF input

In the context of TLS 1.3, the next-generation shared secret is used in place of a currently unused input in the TLS 1.3 key schedule, namely replacing the "0" "IKM" input to the final "HKDF-Extract":



This approach is not taken in any of the known post-quantum/hybrid TLS drafts. However, it bears some similarities to the approach for using external PSKs in [EXTERN-PSK].

3.4.7. Benefits and Drawbacks

New logic. While (Comb-Concat) (Section 3.4.1), (Comb-KDF-1) (Section 3.4.2), and (Comb-KDF-2) (Section 3.4.3) require new logic to compute the concatenated shared secret, this value can then be used by the TLS 1.3 key schedule without changes to the key schedule logic. In contrast, (Comb-Chain) (Section 3.4.5) requires the TLS 1.3 key schedule to be extended for each extra component algorithm.

Philosophical. The TLS 1.3 key schedule already applies a new stage for different types of keying material (PSK versus (EC)DHE), so (Comb-Chain) (Section 3.4.5) continues that approach.

Efficiency. (Comb-KDF-1) (Section 3.4.2), (Comb-KDF-2) (Section 3.4.3), and (Comb-Chain) (Section 3.4.5) increase the number

of KDF applications for each component algorithm, whereas (Comb-Concat) (Section 3.4.1) and (Comb-AltInput) (Section 3.4.6) keep the number of KDF applications the same (though with potentially longer inputs).

Extensibility. (Comb-AltInput) (Section 3.4.6) changes the use of an existing input, which might conflict with other future changes to the use of the input.

More than 2 component algorithms. The techniques in (Comb-Concat) (Section 3.4.1) and (Comb-Chain) (Section 3.4.5) can naturally accommodate more than 2 component shared secrets since there is no distinction to how each shared secret is treated. (Comb-AltInput) (Section 3.4.6) would have to make some distinct, since the 2 component shared secrets are used in different ways; for example, the first shared secret is used as the "IKM" input in the 2nd "HKDF-Extract" call, and all subsequent shared secrets are concatenated to be used as the "IKM" input in the 3rd "HKDF-Extract" call.

3.4.8. Open questions

At this point, it is unclear which, if any, of the above methods preserve FIPS compliance: i.e., if one shared secret is from a FIPS-compliant method (e.g., ECDH), and another shared secret is from a non-approved method (e.g., post-quantum), is the result still considered FIPS compliant? Guidance from NIST on this question would be helpful. Specifically, are any of these approaches acceptable under either [NIST-SP-800-56C] or [NIST-SP-800-135]?

4. Candidate instantiations

In this section, we describe two candidate instantiations of hybrid key exchange in TLS 1.3, based on the design considerations framework above. It is not our intention that both of these instantations be standardized; we are providing two for discussion and for comparing and contrasting the two approaches.

4.1. Candidate Instantiation 1

Candidate Instantiation 1 allows for two or more component algorithms to be combined (Num-2+) (Section 3.2.2), and negotiates the combination using markers in the "NamedGroup" list as pointers to an extension listing the algorithms comprising each possible combination (Neg-Comb-2) (Section 3.1.3.2) following the approach of [WHYTE13]. The client conveys its multiple key shares individually in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Multiple) (Section 3.3.2). The server conveys its multiple key shares concatenated together in its "KeyShareServerHello" struct

(Shares-Concat) (Section 3.3.1). The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).

4.1.1. ClientHello extension supported_groups

Following [WHYTE13] section 3.1, the "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing markers for hybrid combinations:

```
enum {
    /* existing named groups */
    secp256r1 (23),
    ...,

    /* new code points eventually defined for post-quantum algorithms */
    ...,

    /* new code points reserved for hybrid markers */
    hybrid_marker00 (0xFD00),
    hybrid_marker01 (0xFD01),
    ...
    hybrid_markerFF (0xFDFF),

    /* existing reserved code points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;
```

"hybrid_marker" code points do not a priori represent any fixed combination. Instead, during each session establishment, the client defines what it wants each "hybrid_marker" code point to represent using the following extension.

4.1.2. ClientHello extension hybrid_extension

Following [WHYTE13] section 3.2.4, a new "ClientHello" "hybrid_extension" extension is defined. It is defined as follows:

```
struct {
    NamedGroup hybrid_marker;
    NamedGroup components<2..10>;
} HybridMapping;

struct {
    HybridMapping map<0..255>;
} HybridExtension;
```

The "HybridExtension" contains 0 or more "HybridMapping"s. Each "HybridMapping" corresponds to one of the "hybrid_marker" included in the "supported_groups" extension, and lists the component algorithms that are meant to comprise the this hybrid combination, which can be any of the existing named groups (elliptic curve or finite field), new code points eventually defined for post-quantum algorithms, or reserved code points for private use.

4.1.3. ClientHello extension key_share

No syntactical modifications are made to the "KeyShareEntry" or "KeyShareClientHello" data structures.

Semantically, the client does not send a "KeyShareEntry" corresponding to any of the "hybrid_marker" code points. Instead, the client sends "KeyShareEntry" for each of the component algorithms listed in the "HybridMapping"s.

For example, if the list of "supported_groups" is "secp256r1", "x25519", "hybrid_marker00", and "hybrid_marker01", where "hybrid_marker00" comprises "secp256r1" with a fictional post-quantum algorithm "PQ1", and "hybrid_marker01" comprises "x25519" with "PQ1", then the client could send three "KeyShareEntry" components: one for "secp256r1", one for "x25519", and one for "PQ1".

4.1.4. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string.

To complete the negotiation of a hybrid algorithm, the server responds with the "NamedGroup" value being the "hybrid_marker" code point correspond to the combination that the server was willing to agree to.

The "key_exchange" string is the octet representation of the following struct:

```
struct {
    KeyShareEntry key_share<2..10>;
} HybridKeyShare;
```

where there is one "key_share" entry for each of the components of this hybrid combination.

Note that the "key_exchange" string has a maximum length of 2^16-1 octets, which may be insufficient for some post-quantum algorithms or

for some hybridizations of multiple post-quantum algorithms. It remains an open question as to whether this length can be increased without breaking existing TLS 1.3 implementations.

4.1.5. Key schedule

The component algorithm shared secrets are combined by concatenating them, then applying a key derivation function, the output of which is then used in the TLS 1.3 key schedule in place of the (EC)DHE shared secret. The component shared secrets are concatenated in the order that they appear in the "components" vector of the "HybridMapping" extension above.

We provide two options for concatenating the shared secrets, and would like feedback from the working group in which to proceed with.

Each component algorithm's "shared_secret" is defined by the algorithm itself, for example the DHE or ECDHE shared secrets as defined in Section 7.4 of [TLS13], or as defined by post-quantum methods once standardized in their own documents.

Option 1: Using data structures. Option 1 uses a full-fledged TLS 1.3 data structure to represent the list of component shared secrets. As a result, lengths of each shared secret are unambiguously encoded.

```
struct SharedSecret {
    opaque shared_secret<0..2^16-1>;
}

struct {
    SharedSecret component<2..10>;
} HybridSharedSecret;
```

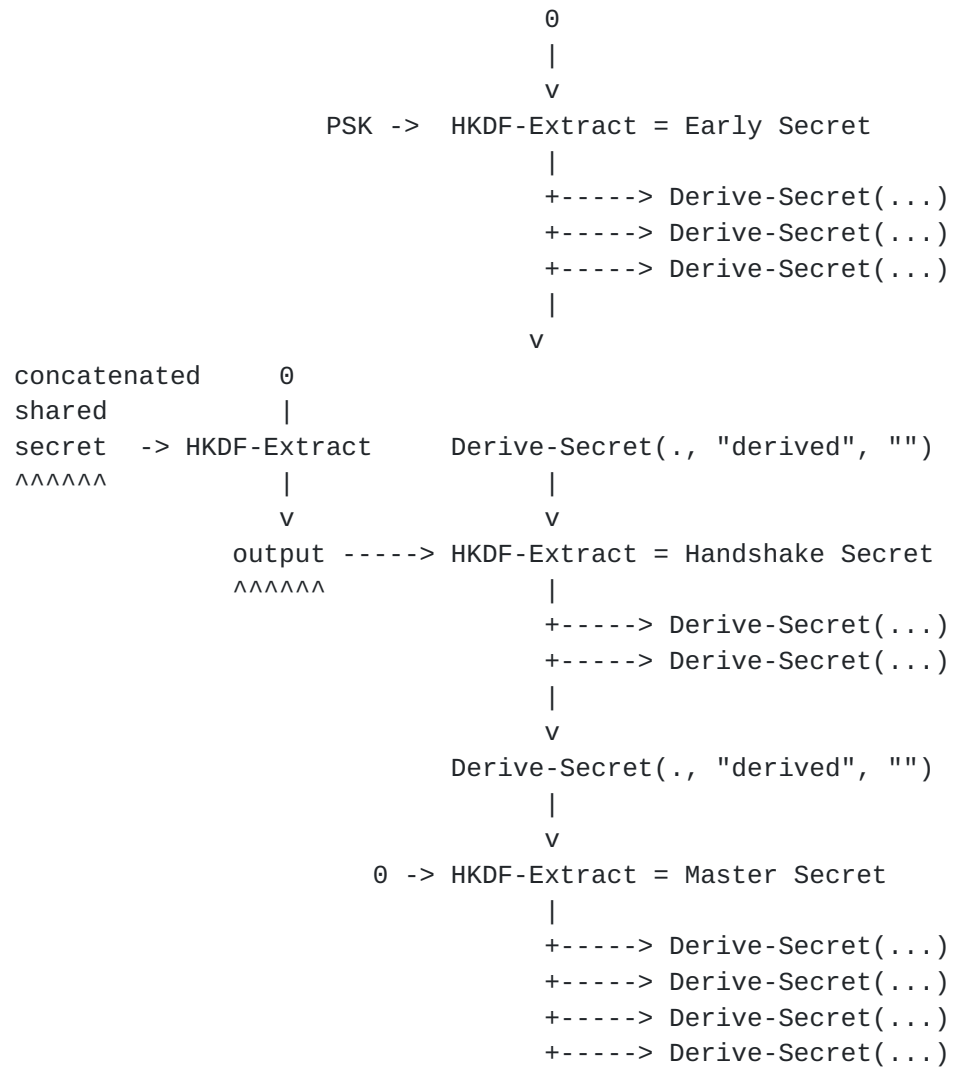
The "concatenated_shared_secret" is then the octet representation of the "HybridSharedSecret " struct.

Option 2: Direct concatenation. Option 2 directly concatenates the shared secrets. Option 2 should only be considered if the shared secret for each algorithm is guarantees to be of a fixed length, which would imply that, once the component algorithms are fixed, concatenation is bijective.

```
concatenated_shared_secret = shared_secret0 | shared_secret1 | ...
```

In either option, the "concatenated_shared_secret" octet string is used as the IKM argument of HKDF-Extract, with the zero-length string as the salt argument. THe output of HKDF-Extract is used as the IKM

argument for HKDF-Extract's calculation of the handshake secret, as shown below.



4.2. Candidate Instantiation 2

Candidate Instantiation 2 allows for exactly two component algorithms to be combined (Num-2) (Section 3.2.1), and uses code points standardized for each permissible combination. The client concatenates its multiple key shares together as a distinct entry in the "client_shares" vector of the "ClientHello" "key_share" extension (Shares-Concat) (Section 3.3.1). The server does the same. The shared secrets are combined by concatenating them then feeding them through a KDF, then feeding the result into the TLS 1.3 key schedule (Comb-KDF-2) (Section 3.4.3).

4.2.1. ClientHello extension supported_groups

The "NamedGroup" enum used by the client to populate the "supported_groups" extension is extended to include new code points representing each desired combination.

For example,

```
enum {
    /* existing named groups */
    secp256r1 (23),
    x25519 (0x001D),
    ...,

    /* new code points eventually defined for post-quantum algorithms */
    PQ1 (0x????),
    PQ2 (0x????),
    ...,

    /* new code points defined for hybrid combinations */
    secp256r1_PQ1 (0x????),
    secp256r1_PQ2 (0x????),
    x25519_PQ1 (0x????),
    x25519_PQ2 (0x????),

    /* existing reserved code points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;
```

4.2.2. ClientHello extension KeyShareClientHello

The client sends a "KeyShareClientHello" struct containing multiple "KeyShareEntry" values, some of which may correspond to some of the hybrid combination code points it listed in the "supported_groups" extension above.

The "KeyShareEntry" for a hybrid combination code point contains an opaque "key_exchange" string which is the octet representation of the following struct:

```
struct {
    KeyShareEntry key_share<2..10>;
} HybridKeyShare;
```

where there is one "key_share" entry for each of the components of this hybrid combination.

Note that this approach may result in duplication of key shares being sent; for example, a client wanting to support either the combination "secp256r1_PQ1" or "x25519_PQ1" would send two "PQ1" key shares.

4.2.3. ServerHello extension KeyShareServerHello

The server responds with a "KeyShareServerHello" struct containing a single "KeyShareEntry", which contains a single "NamedGroup" value and an opaque "key_exchange" string. The "key_exchange" string is the octet representation of the "HybridKeyShare" struct defined above.

4.2.4. Key schedule

The key schedule is computed as in Candidate Instantiation 1 above.

4.3. Comparing Candidate Instantiation 1 and 2

CI2 requires much less change to negotiation routines - each hybrid combination is just a new key exchange method, and the concatenation of key shares and shared secrets can be handled internally to that method. This comes at the cost, however, of combinatorial explosion of code points: one code point needs to be standardized for each desired combination. We have also limited the number of hybrid algorithms to 2 in CI2 to somewhat limit the explosion of code points needing to be defined. Concatenating client key shares also risks sending duplicate key shares, increasing communication sizes.

CI1 requires more change to negotiation routines, since it introduces new data structures and has an indirect mapping between hybrid combinations and key shares. Benefits from this approach include avoiding sending duplicate key shares and not needing to standardize every possible supported combination. Implementers, however, must do the work of deciding which combinations of algorithms are meaningful / tolerable / desirable from a security perspective, potentially complicating interoperability.

5. IANA Considerations

If Candidate Instantiation 1 is selected, the TLS Supported Groups registry will have to be updated to include code points for hybrid markers.

6. Security Considerations

The majority of this document is about security considerations. As noted especially in Section 3.4, the shared secrets computed in the hybrid key exchange should be computed in a way that achieves the

"hybrid" property: the resulting secret is secure as long as at least one of the component key exchange algorithms is unbroken. While many natural approaches seem to achieve this, there can be subtleties (see for example the introduction of [GIACON]).

The rest of this section highlights a few unresolved questions related to security.

6.1. Active security

One security consideration that is not yet resolved is whether key encapsulation mechanisms used in TLS 1.3 must be secure against active attacks (IND-CCA), or whether security against passive attacks (IND-CPA) suffices. Existing security proofs of TLS 1.3 (such as [DFGS15], [DOWLING]) are formulated specifically around Diffie-Hellman and use an "actively secure" Diffie-Hellman assumption (PRF Oracle Diffie-Hellman (PRF-ODH)) rather than a "passively secure" DH assumption (e.g. decisional Diffie-Hellman (DDH)), but do not claim that the actively secure notion is required. In the context of TLS 1.2, [KPW13] show that, at least in one formalization, a passively secure assumption like DDH is insufficient (even when signatures are used for mutual authentication). Resolving this issue for TLS 1.3 is an open question.

6.2. Resumption

TLS 1.3 allows for session resumption via a pre-shared key. When a pre-shared key is used during session establishment, an ephemeral key exchange can also be used to enhance forward secrecy. If the original key exchange was hybrid, should an ephemeral key exchange in a resumption of that original key exchange be required to use the same hybrid algorithms?

6.3. Failures

Some post-quantum key exchange algorithms have non-trivial failure rates: two honest parties may fail to agree on the same shared secret with non-negligible probability. Does a non-negligible failure rate affect the security of TLS? How should such a failure be treated operationally? What is an acceptable failure rate?

7. Acknowledgements

These ideas have grown from discussions with many colleagues, including Christopher Wood, Matt Campagna, and authors of the various hybrid Internet-Drafts and implementations cited in this document. The immediate impetus for this document came from discussions with

attendees at the Workshop on Post-Quantum Software in Mountain View, California, in January 2019.

Martin Thomson suggested the (Comb-KDF-1) (Section 3.4.2) approach.

8. References

8.1. Normative References

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

8.2. Informative References

[BCNS15] Bos, J., Costello, C., Naehrig, M., and D. Stebila, "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem", 2015 IEEE Symposium on Security and Privacy, DOI 10.1109/sp.2015.40, May 2015.

[BERNSTEIN] "Post-Quantum Cryptography", Springer Berlin Heidelberg book, DOI 10.1007/978-3-540-88702-7, 2009.

[BINDEL] Bindel, N., Brendel, J., Fischlin, M., Goncalves, B., and D. Stebila, "Hybrid Key Encapsulation Mechanisms and Authenticated Key Exchange", Post-Quantum Cryptography (PQCrypto) , 2019, <https://eprint.iacr.org/2018/903>.

[CECPQ1] Braithwaite, M., "Experimenting with Post-Quantum Cryptography", July 2016, <https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html>.

[CECPQ2] Langley, A., "CECPQ2", December 2018, <https://www.imperialviolet.org/2018/12/12/cecpq2.html>.

[DFGS15] Dowling, B., Fischlin, M., Guenther, F., and D. Stebila, "A Cryptographic Analysis of the TLS 1.3 Handshake Protocol Candidates", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15, DOI 10.1145/2810103.2813653, 2015.

[DODIS] Dodis, Y. and J. Katz, "Chosen-Ciphertext Security of Multiple Encryption", Theory of Cryptography pp. 188-209, DOI 10.1007/978-3-540-30576-7_11, 2005.

[DOWLING] Dowling, B., "Provable Security of Internet Protocols", Queensland University of Technology dissertation, DOI 10.5204/thesis.eprints.108960, n.d..

[ETSI] Campagna, M., Ed. and . others, "Quantum safe cryptography and security: An introduction, benefits, enablers and challengers", ETSI White Paper No. 8 , June 2015, <https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf>.

[EVEN] Even, S. and O. Goldreich, "On the Power of Cascade Ciphers", Advances in Cryptology pp. 43-50, DOI 10.1007/978-1-4684-4730-9_4, 1984.

[EXTERN-PSK] Housley, R., "TLS 1.3 Extension for Certificate-based Authentication with an External Pre-Shared Key", draft-ietf-tls-tls13-cert-with-extern-psk-02 (work in progress), May 2019.

[FRODO] Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., and D. Stebila, "Frodo", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16, DOI 10.1145/2976749.2978425, 2016.

[GIACON] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", Public-Key Cryptography - PKC 2018 pp. 190-218, DOI 10.1007/978-3-319-76578-5_7, 2018.

[HARNIK] Harnik, D., Kilian, J., Naor, M., Reingold, O., and A. Rosen, "On Robust Combiners for Oblivious Transfer and Other Primitives", Lecture Notes in Computer Science pp. 96-113, DOI 10.1007/11426639_6, 2005.

[HOFFMAN] Hoffman, P., "The Transition from Classical to Post-Quantum Cryptography", draft-hoffman-c2pq-05 (work in progress), May 2019.

[IKE-HYBRID] Tjhai, C., Tomlinson, M., grbartle@cisco.com, g., Fluhrer, S., Geest, D., Garcia-Morchon, O., and V. Smyslov, "Framework to Integrate Post-quantum Key Exchanges into Internet Key Exchange Protocol Version 2 (IKEv2)", draft-tjhai-ipsecme-hybrid-qske-ikev2-03 (work in progress), January 2019.

[IKE-PSK] Fluhrer, S., McGrew, D., Kampanakis, P., and V. Smyslov, "Postquantum Preshared Keys for IKEv2", draft-ietf-ipsecme-qr-ikev2-08 (work in progress), March 2019.

[KIEFER] Kiefer, F. and K. Kwiatkowski, "Hybrid ECDHE-SIDH Key Exchange for TLS", draft-kiefer-tls-ecdhe-sidh-00 (work in progress), November 2018.

[KPW13] Krawczyk, H., Paterson, K., and H. Wee, "On the Security of the TLS Protocol: A Systematic Analysis", Advances in Cryptology - CRYPTO 2013 pp. 429-448, DOI 10.1007/978-3-642-40041-4_24, 2013.

[LANGLEY] Langley, A., "Post-quantum confidentiality for TLS", April 2018, <https://www.imperialviolet.org/2018/04/11/pqconftls.html>.

[NIELSEN] Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information", Cambridge University Press , 2000.

[NIST] National Institute of Standards and Technology (NIST), "Post-Quantum Cryptography", n.d., <https://www.nist.gov/pqcrypto>.

[NIST-SP-800-135] National Institute of Standards and Technology (NIST), "Recommendation for Existing Application-Specific Key Derivation Functions", December 2011, <https://doi.org/10.6028/NIST.SP.800-135r1>.

[NIST-SP-800-56C] National Institute of Standards and Technology (NIST), "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", April 2018, <https://doi.org/10.6028/NIST.SP.800-56Cr1>.

[OQS-102] Open Quantum Safe Project, "OQS-OpenSSL-1-0-2_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_0_2-stable>.

[OQS-111] Open Quantum Safe Project, "OQS-OpenSSL-1-1-1_stable", November 2018, <https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable>.

[SCHANCK] Schanck, J. and D. Stebila, "A Transport Layer Security (TLS) Extension For Establishing An Additional Shared Secret", draft-schanck-tls-additional-keyshare-00 (work in progress), April 2017.

[WHYTE12] Schanck, J., Whyte, W., and Z. Zhang, "Quantum-Safe Hybrid (QSH) Ciphersuite for Transport Layer Security (TLS) version 1.2", draft-whyte-qsh-tls12-02 (work in progress), July 2016.

[WHYTE13] Whyte, W., Zhang, Z., Fluhrer, S., and O. Garcia-Morchon, "Quantum-Safe Hybrid (QSH) Key Exchange for Transport Layer Security (TLS) version 1.3", draft-whyte-qsh-tls13-06 (work in progress), October 2017.

[XMSS] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/info/rfc8391>.

[ZHANG] Zhang, R., Hanaoka, G., Shikata, J., and H. Imai, "On the Security of Multiple Encryption or CCA-security+CCA-security=CCA-security?", Public Key Cryptography - PKC 2004 pp. 360-374, DOI 10.1007/978-3-540-24632-9_26, 2004.

Authors' Addresses

Douglas Stebila
University of Waterloo

Email: dstebila@uwaterloo.ca

Scott Fluhrer
Cisco Systems

Email: sfluhrer@cisco.com

Shay Gueron
University of Haifa and Amazon Web Services

Email: shay.gueron@gmail.com