Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH

Eric Crockett[1], Christian Paquin[2], and Douglas Stebila[3]

[1] *AWS* ericcro@amazon.com
[2] *Microsoft Research* cpaquin@microsoft.com
[3] *University of Waterloo* dstebila@uwaterloo.ca

July 19, 2019

Abstract

Once algorithms for quantum-resistant key exchange and digital signature schemes are selected by standards bodies, adoption of post-quantum cryptography will depend on progress in integrating those algorithms into standards for communication protocols and other parts of the IT infrastructure. In this paper, we explore how two major Internet security protocols, the Transport Layer Security (TLS) and Secure Shell (SSH) protocols, can be adapted to use post-quantum cryptography.

First, we examine various design considerations for integrating post-quantum and hybrid key exchange and authentication into communications protocols generally, and in TLS and SSH specifically. These include issues such as how to negotiate the use of multiple algorithms for hybrid cryptography, how to combine multiple keys, and more. Subsequently, we report on several implementations of post-quantum and hybrid key exchange in TLS 1.2, TLS 1.3, and SSHv2. We also report on work to add hybrid authentication in TLS 1.3 and SSHv2. These integrations are in Amazon s2n and forks of OpenSSL and OpenSSH; the latter two rely on the liboqs library from the Open Quantum Safe project.

1 Introduction

Post-quantum (PQ) cryptographic algorithms have security based on mathematical problems that are widely believed to be difficult for a quantum adversary. The interest in deploying these algorithms is growing due to the desire to hedge against the future possibility of a large-scale quantum computer. NIST is currently in the process of selecting post-quantum algorithms for key exchange and authentication for standardization. Although this is an important step, the adoption of post-quantum cryptography will also depend on the successful transition of communication protocols and applications to use these new algorithms.

Some cryptographic algorithm transitions have happened relatively quickly: for example, AES was released as a FIPS standard in November 2001 [32]; an RFC for its use in TLS was published in June 2002 [13], and included in a December 2002 release of OpenSSL [48].

However, there are many examples of cryptographic algorithms transitions that took a long time. For example, while elliptic curve cryptography (ECC) was invented in the 1980s, the first FIPS standard using ECC was in 2000 [31], the RFC for its use in TLS appeared in 2006 [30], but it was not enabled for forward secrecy by default by Google until late 2011 [27]. As another example, SHA-2 was published as a FIPS standard in 2002 [33], and theoretical weaknesses in SHA-1 were

known in 2005 [51]. Yet web browsers did not stop accepting SHA-1-based certificates until January 2017 [53], only a month before the first collisions in SHA-1 were demonstrated [45, 46].

This highlights the importance of beginning to plan for the transition to post-quantum cryptography early. There are several steps in such a transition for network protocols. First, each network protocol must be evaluated for any constraints that make it challenging to add new algorithms with potentially new characteristics, such as lack of ability to replace or negotiate cryptographic algorithms, or limitations on sizes of keys or packets. Next, specific choices must be made in how to integrate the new algorithm into the protocol: engineering choices, such as how parameters and keys are represented in network packets, and cryptographic choices, such as how keying material is used. Furthermore, these designs must be done in a way that preserves backward compatibility with endpoints (and middle boxes) that have not yet been upgraded, while achieving desirable protocol functionality for upgraded endpoints.

Finally, the transition to post-quantum cryptography includes a twist not seen in previous cryptographic transitions: the use of two (or more) algorithms simultaneously, in what is being called "hybrid" mode. There have been suggestions that some parties may decide to use both traditional (e.g., elliptic curve Diffie–Hellman) and post-quantum algorithms together for a variety of reasons, such as maintaining compliance with industry or government regulations that have not yet been updated while still obtaining post-quantum security, or for early adopters who want the potential of post-quantum security without relying solely on a newer and relatively untested algorithm.

Related work. There have been a variety of documents outlining high-level perspectives on the general transition to post-quantum cryptography, including a whitepaper by the European Telecommunications Standards Institute [12] and technical report by Hoffman [22]. There have been several Internet-Drafts submitted to the IETF describing mechanisms for adding post-quantum or hybrid key exchange in TLS 1.2 [11, 43] and TLS 1.3 [26, 42, 44, 52]; this paper is based in part on some of the ideas in [11, 44]. There have also been Internet-Drafts submitted on post-quantum security for the Internet Key Exchange version 2 (IKEv2) protocol [17, 49].

Various groups have also done experimental demonstrations of post-quantum or hybrid key exchange in TLS 1.2 [2, 8, 9, 10, 37] and TLS 1.3 [28, 38]. This paper includes results based on [2, 36, 37, 38].

Contributions. In this paper, we report on case studies exploring how two major Internet security protocols, Transport Layer Security (TLS) and Secure Shell (SSH), can be adapted to use post-quantum cryptography, both for confidentiality (via post-quantum key exchange) and authentication (via post-quantum digital signatures). Each of our case studies includes an evaluation of design options in the context of the protocol, selection of one or two instantiations of those design options and an implementation thereof, accompanied by observations and lessons learned from the implementation. As of this writing, the case studies include results on more than half of the KEMs and signature scheme families submitted to the NIST Round 2 submission.[1]

[1] In the long run, the OQS team aims to have results on all KEMs and signature schemes in Round 2. As of this writing, liboqs includes 9 of 17 KEM families (BIKE, FrodoKEM, Kyber, LEDAcrypt, NewHope, NTRU, one member of the NTS-KEM family, Saber, and SIKE) and 6 of 9 signature families (Dilithium, MQDSS, Picnic, qTesla, Rainbow, and SPHINCS+). It is missing the KEMs Classic McEliece, HQC, LAC, NTRU Prime, some of the NTS-KEM family, ROLLO, Round5, RQC, and Three Bears, and the signature schemes FALCON, GeMSS, and LUOV. The results on BIKE and qTesla are based on the Round 1 submissions. The choice of algorithms currently was driven by a variety of non-scientific factors, including schemes that this paper's authors were connected to and schemes with implementations in the PQClean project or that were readily adaptable to the frameworks used.

Table 1: Test results for key exchange using post-quantum and hybrid key encapsulation mechanisms in TLS and SSH implementations

	s2n (TLS 1.2)	**OpenSSL 1.0.2** (TLS 1.2)	**OpenSSL 1.1.1** (TLS 1.3)	**OpenSSH**
BIKE1-L1 (round 1)	– ●	● ●	● ●	● ●
BIKE1-L3 (round 1)	– –	● ●	● ●	● ●
BIKE1-L5 (round 1)	– –	● ●	● ●	● ●
BIKE2-L1 (round 1)	– –	● ●	● ●	● ●
BIKE2-L3 (round 1)	– –	● ●	● ●	● ●
BIKE2-L5 (round 1)	– –	● ●	● ●	● ●
BIKE3-L1 (round 1)	– –	● ●	● ●	● ●
BIKE3-L3 (round 1)	– –	● ●	● ●	● ●
BIKE3-L5 (round 1)	– –	● ●	● ●	● ●
FrodoKEM-640-AES	– –	● ●	● ●	● ●
FrodoKEM-640-SHAKE	– –	● ●	● ●	● ●
FrodoKEM-976-AES	– –	● ●	● ●	● ●
FrodoKEM-976-SHAKE	– –	● ●	● ●	● ●
FrodoKEM-1344-AES	– –	◓ ◓	◓ ◓	● ●
FrodoKEM-1344-SHAKE	– –	◓ ◓	◓ ◓	● ●
Kyber512	– –	● ●	● ●	● ●
Kyber768	– –	● ●	● ●	● ●
Kyber1024	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-12†	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-32†	– –	● ●	● ●	● ●
LEDAcrypt-KEM-LT-52†	– –	● ●	● ●	● ●
NewHope-512-CCA	– –	● ●	● ●	● ●
NewHope-1024-CCA	– –	● ●	● ●	● ●
NTRU-HPS-2048-509	– –	● ●	● ●	● ●
NTRU-HPS-2048-677	– –	● ●	● ●	● ●
NTRU-HPS-4096-821	– –	● ●	● ●	● ●
NTRU-HRSS-701	– –	● ●	● ●	● ●
NTS-KEM(12,64)†	– –	○ ○	○ ○	○ ○
LightSaber-KEM	– –	● ●	● ●	● ●
Saber-KEM	– –	● ●	● ●	● ●
FireSaber-KEM	– –	● ●	● ●	● ●
SIKEp503 (round 1)	– ●	– –	– –	– –
SIKEp434	– –	● ●	● ●	● ●
SIKEp503	– –	● ●	● ●	● ●
SIKEp610	– –	● ●	● ●	● ●
SIKEp751	– –	● ●	● ●	● ●

Legend: In each cell, the first symbol is for post-quantum-only key exchange, the second symbol is for post-quantum + ECDH key exchange; see relevant sections for the elliptic curve used. Post-quantum algorithms are NIST Round 2 specifications unless otherwise indicated; † denotes algorithms that are on testing branches of our libraries as of the time of writing. ● denotes success; ◓ denotes a failure that could be fixed by changing a parameter in the implementation; ○ denotes a failure that has not been resolved; – denotes the combination was not tested.

Our specific case studies are as follows:

- **TLS 1.2**: Post-quantum and hybrid key exchange, in OpenSSL 1.0.2s and Amazon s2n.
- **TLS 1.3**: Post-quantum and hybrid key exchange, and post-quantum and hybrid authentication, in OpenSSL 1.1.1c.
- **SSH 2**: Post-quantum and hybrid key exchange, and post-quantum and hybrid authentication, in OpenSSH 7.9.

The OpenSSL and OpenSSH implementations rely on the liboqs library from the Open Quantum Safe project, which is a C library that provides implementations of post-quantum KEMs and signatures schemes in a common interface based on implementations from NIST submission packages; some of the implementations in liboqs are based on implementations in the PQClean project [25]. The s2n implementation relies on implementations directly from NIST submission packages.

Tables 1 and 2 list the KEM and signature schemes tested in the case studies we examine, and whether each scheme's use was successful in these prototypes. The failures encountered were in general due to large message sizes (public keys / ciphertexts for KEMs, public keys / signatures for signature schemes):

- Some of the failures involved sizes that were bigger than the protocol specification allowed; these might be fixable by changing the protocol specification, such as increasing 2-byte length fields to 3-byte length fields; but this increases the risk of incompatibilities with existing implementations. These are denoted by ○ in Tables 1 and 2.
- Some of the failures involved sizes that were within protocol specification tolerances, but where the implementation in question had internal buffers or parameters set smaller than the maximum size permitted by the specification. In these cases, we were able to increase the implementations' buffers or parameters and get the algorithm working. These are denoted by ◓ in Tables 1 and 2.

Details about the failures and lessons learned appear in the corresponding sections of the paper.

Limitations and future work. The implementations reported in this paper at the time of writing had not incorporated some NIST Round 2 submissions into their frameworks, so the results are incomplete. Beyond adding the remaining Round 2 submissions, the next steps for our work would be on benchmarking the performance of network protocols when using post-quantum algorithms.

2 Hybrid modes

TLS and SSH are designed with algorithm agility in mind, so they permit parties to support multiple cryptographic algorithms within each category of functionality (key exchange, public key authentication, symmetric cipher, hash function, etc.); such a combination is called a "ciphersuite" in the context of TLS, and we will use that terminology in general.

Both TLS and SSH include a mechanism for negotiating which ciphersuite to use, either all-at-once or in an à la carte manner. In principle, negotiation allows for parties which support different subsets of algorithms to select a mutually agreeable ciphersuite, provided they have at least one overlapping supported algorithm in each category. In practice, negotiation works fairly well: while there have been problems with incompatibilities between implementations, these have tended to arise elsewhere in the protocol, not directly in the cryptographic algorithms supported.

However, TLS and SSH are designed to actually negotiate and subsequently use only a single algorithm in each category of the ciphersuite: while they can select from multiple key exchange mechanisms, they have to pick only one to use. To support hybrid modes of key exchange or

Table 2: Test results for authentication using post-quantum and hybrid signatures in TLS and SSH implementations

	OpenSSL 1.1.1 (TLS 1.3)	**OpenSSH**
Dilithium-2	● ●	● ●
Dilithium-3	● ●	● ●
Dilithium-4	● ●	● ●
MQDSS-31-48	◓ ◓	● ●
MQDSS-31-64	◓ ◓	● ●
Picnic-L1-FS	◓ ◓	● ●
Picnic-L1-UR	◓ ◓	● ●
Picnic-L3-FS	○ ○	● ●
Picnic-L3-UR	○ ○	● ●
Picnic-L5-FS	○ ○	● ●
Picnic-L5-UR	○ ○	● ●
Picnic2-L1-FS	● ●	● ●
Picnic2-L3-FS	◓ ◓	● ●
Picnic2-L5-FS	◓ ◓	● ●
qTesla-I (round 1)	● ●	● ●
qTesla-III-size (round 1)	● ●	● ●
qTesla-III-speed (round 1)	● ●	● ●
Rainbow-Ia-Classic†	◓ ◓	◓ ◓
Rainbow-Ia-Cyclic†	● ●	● ●
Rainbow-Ia-Cyclic-Compressed†	● ●	● ●
Rainbow-IIIc-Classic†	◓ ◓	○ ○
Rainbow-IIIc-Cyclic†	◓ ◓	○ ○
Rainbow-IIIc-Cyclic-Compressed†	◓ ◓	○ ○
Rainbow-Vc-Classic†	◓ ◓	○ ○
Rainbow-Vc-Cyclic†	◓ ◓	○ ○
Rainbow-Vc-Cyclic-Compressed†	◓ ◓	○ ○
SPHINCS+-{Haraka,SHA256,SHAKE256}-128f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-128s-{robust,simple}	● ●	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-192f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-192s-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-256f-{robust,simple}	◓ ◓	● ●
SPHINCS+-{Haraka,SHA256,SHAKE256}-256s-{robust,simple}	◓ ◓	● ●

Legend: In each cell, the first symbol is for post-quantum-only authentication, the second symbol is for hybrid post-quantum + traditional authentication; see relevant sections for the traditional authentication method used. Post-quantum algorithms are NIST Round 2 specifications unless otherwise indicated; † denotes algorithms that are on testing branches of our libraries as of the time of writing. ● denotes success; ◓ denotes a failure that could be fixed by changing a parameter in the implementation; ○ denotes a failure because a fix would violate the protocol specification or for another as-yet unidentified reason.

authentication, the protocol must be modified to indicate how to negotiate a combination of algorithms to use in hybrid mode, and how to combine them cryptographically.

In this section we explore high-level goals and design considerations for hybrid modes of key exchange and authentication. In subsequent sections, we discuss options specific to particular versions of TLS and SSH.

2.1 Goals for hybrid modes

The primary goal of a hybrid mode is to ensure that the desired security property holds as long as one of the component schemes remains unbroken. For key exchange, this means that the session key should remain secure (and thus application data confidential) as long as one of the component key exchange mechanisms is unbroken. For authentication, this means that the protocol should provide secure authentication as long as one of the digital signatures schemes is unbroken at the time of session establishment.

In addition to the primary cryptographic goals, there may be several additional goals for hybrid modes in real-world network protocols. These include:

Backwards compatibility. Clients and servers who are "hybrid-aware", i.e., compliant with whatever hybrid mode is added to TLS or SSH, 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 negotiate and use hybrid modes.
2. Hybrid-aware client, non-hybrid-aware server: These parties should negotiate a traditional (non-PQ) ciphersuite (if 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 (non-PQ) ciphersuite (if 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 modes 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 the hybridization should not substantially affect performance. Preliminary results about such performance include [8, 9, 10].

Low latency. Use of hybrid modes 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 / signature sizes for post-quantum algorithms range from hundreds of bytes to over one hundred kilobytes, so this impact can be substantial. See [8, 9] for preliminary results in a laboratory setting, and [29] for preliminary results on more realistic networks.
- Additional round trips added to the protocol. See below.

No extra round trips. Attempting to negotiate hybrid modes 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 modes should not mean having to send multiple cryptographic values for the same algorithm.

2.2 Design considerations for hybrid modes

In general, we identify four distinct axes along which one can make choices when adding support for hybrid modes. These are:

1. How to negotiate the use of hybridization in general, and component algorithms and parameters specifically?
2. How many component algorithms can be combined?
3. How should cryptographic data from multiple algorithms (public keys / ciphertexts / signatures) be conveyed?
4. How should cryptographic data from multiple algorithms (e.g., shared secrets) be combined?

Some of the answers to these questions are specific to details of a particular network protocol, while others are independent of protocol specifics.

2.2.1 Negotiation

Many network protocols, including TLS and SSH, negotiate which algorithms to use with the following basic approach: one party sends an ordered list of supported algorithms, and the other party responds either with a single selection from that list, or with their own ordered list of supported algorithms (and then the protocol specifies how to mutually select a single element from two ordered lists). If no mutually supported algorithm can be found, an error is raised or communicated.

For hybrid modes, the goal is to negotiate two or more algorithms and use both of them. The main choice to make when negotiating algorithms for hybrid modes is whether each algorithm in the hybrid mode should be negotiated separately or as a single combined algorithm. A second choice is whether to separately or jointly negotiate parameterizations of a cryptographic algorithm. For example, in key exchange, a ciphersuite could either specify that the key exchange will be "ECDH", with parameters specified separately, or the ciphersuite could include both the algorithm and parameters, e.g., "ECDH+nistp256".

If each algorithm is to be negotiated separately, then the protocol's message formats and logic will need to be modified to allow negotiation of multiple component algorithms. The designer of the negotiation mechanism must also choose whether to separately negotiate component algorithms of different types – for example, "select one of the following traditional algorithms, select one of the following post-quantum algorithms" – or informally (at least within the protocol syntax) distinguish between algorithm types – for example, "select two of the following algorithms" where it is up to the implementation to pick one traditional and one post-quantum. Care must be taken to ensure that individually negotiated algorithms having matching security levels. There is also the potential that additional message formats for conveying a second list to negotiate may affect backwards compatibility with old implementations.

In contrast, negotiating an overall combination of algorithms can be more easily accomplished by defining new identifiers that simply represent a pair of algorithms. This requires no new protocol logic or message format modifications during negotiation (although later cryptographic computations must still be updated to use both algorithms). However, drawbacks of this approach could be that a quadratic number of algorithm identifiers must be defined, one for each combination, and that some protocols may end up sending duplicate values (see TLS 1.3 key exchange below).

2.2.2 Number of component algorithms

The key decision to make here is whether the number of algorithms that can be combined in a single hybrid mode is fixed or variable, and if fixed, how many. There appears to be no consensus on this

matter: some Internet-Drafts for hybrid key exchange in TLS have fixed to two algorithms [11,26,42], others have a variable number of two or more [43,52].

2.2.3 How to convey cryptographic data

If hybrid modes are to be used, then the parties must convey cryptographic data (public keys, ciphertexts, signatures) for multiple cryptosystems within the protocol. Many protocols, including TLS and SSH, are designed with some extensibility, but not in arbitrary locations: in TLS, for example, the `ClientHello` and `ServerHello` messages explicitly support extensions, but other messages do not.

To convey cryptographic data from multiple algorithms, one could either try to extend the messages in which the cryptographic data is sent to provide additional locations for cryptographic values, or one could concatenate the data from multiple algorithms into a single value and place that in the existing message structure. In general the latter is simpler (since it requires no changes to the protocol format or logic), and potentially has fewer backwards compatibility concerns, but it can lead to duplication; see the example of key exchange in TLS 1.3 in Section 3.2.

2.2.4 How to combine cryptographic data

Decisions must also be made about how to cryptographically combine two or more algorithms in a way that provides the intended hybrid security property: the combination is secure as long as one of the component algorithms remains secure. For example: when combining session keys from two key exchange algorithms, should we XOR them? concatenate them? concatenate then feed into a key derivation function? Here protocol designers should make decisions informed by the literature, which we review briefly.

In terms of confidentiality properties, Even and Goldreich [16] initiated the study of combining multiple symmetric encryption schemes; [14,21,55] examined combining multiple public key encryption schemes, and Harnik et al. [21] coined the term "robust combiner" to refer to a compiler that constructs a hybrid scheme from individual schemes while preserving security properties. More recently, Giacon et al. [18] and Bindel et al. [6] examined combining multiple key encapsulation mechanisms.

For digital signatures, Bindel et al. [7] consider combiners for digital signature schemes.

3 Key exchange case studies

In this section we report on design considerations, instantiations, and lessons learned from adding post-quantum and hybrid key exchange to the TLS and SSH protocols.

3.1 Key exchange in TLS 1.2

This section describes two approaches to implementing hybrid key exchange in TLS 1.2. Although it is not clear that post-quantum TLS 1.2 will be broadly adopted in light of TLS 1.3, there are several reasons why it is interesting to implement hybrid key exchange in TLS 1.2. First, both the general interest in hybrid and the implementation efforts described below predate the standardization of TLS 1.3. Second, TLS 1.2 allows the community to evaluate hybrid key exchange on its own merits, without being complicated by orthogonal issues related to TLS 1.3 (such as infrastructure problems, implementation bugs, message size limits, etc). Examining TLS 1.2 allows us to isolate problems due to the intricacies of post-quantum cryptography from issues of using a relatively new network

protocol. Finally, even though TLS 1.3 has been standardized for nearly a year, TLS 1.2 remains dominant.[2] Although the adoption rate for TLS 1.3 is increasing, we expect that TLS 1.2 will be in use far into the future.

3.1.1 Design considerations

Besides the implementations described below [2, 37], there have been several experimental implementations of post-quantum and/or hybrid key exchange in TLS 1.2 [8, 9, 10] and proposed Internet-Drafts [11, 43].

Negotiation. For hybrid key exchange, the first choice to make, as noted in Section 2.2, is whether to negotiate component algorithms individually or together. The second choice is whether to negotiate parameterizations separately or jointly. [43] negotiates the two hybrid algorithms separately: a new `TLS_QSH` ciphersuite is defined, then the particular classical and post-quantum algorithms are individually negotiated; the post-quantum algorithm parameterizations are negotiated jointly, i.e., a single algorithm identifier `ntru_eess439`. In contrast, [2, 11] defines new hybrid ciphersuites with pairs of algorithms (e.g., `ECDH_BIKE`, `ECDH_SIKE`), with parameters for the post-quantum algorithm negotiated separately. Finally, some implementations [8, 9, 10] take a third approach, in which they choose to negotiate the post-quantum algorithms and parameterizations together, by defining new ciphersuites with selected proposed combinations (e.g., `ecdh+frodo-640`, `ecdh+newhope1024`, etc.); the specific elliptic curve is negotiated using the existing curve negotiation mechanism.

Combining shared secrets. All Internet-Drafts and implementations we have seen so far concatenate the two raw shared secrets (the ECDH shared secret and the PQ shared secret) and use that as the TLS "pre-master secret", which is then input into the TLS key derivation function to compute a master secret from which session keys are derived. Other approaches could include XORing the shared secrets to derive the pre-master secret, or putting them through a KDF to derive the pre-master secret. One issue to consider carefully is checking that such a combination is supported by a security argument. [6, 18] give positive results for concatenation when public keys / ciphertexts are included in the key derivation function, but the basic TLS 1.2 key schedule does not do this.

3.1.2 An example instantiation: OpenSSL 1.0.2

The Open Quantum Safe project implemented post-quantum and hybrid key exchange in TLS 1.2 in a fork of OpenSSL 1.0.2 [37] using KEMs from liboqs. That implementation added both PQ-only and hybrid key exchange.[3] The rest of this subsection explains how hybrid key exchange is implemented in TLS 1.2, since PQ-only is implemented just by adding another algorithm. Only ECDH is supported as the traditional algorithm in the hybrid key exchange.

[2] Sullivan [47] reported in September 2018 that TLS 1.2 still accounted for 86% of encrypted web traffic; at 7% of encrypted traffic, TLS 1.3 just edged out TLS 1.0 (6.8% of encrypted traffic), a protocol obsoleted by TLS 1.1 in 2006. SSL Pulse reports that only 14.8% of websites surveyed even supported TLS 1.3 as of June 2019 [40].

[3] One note about this implementation is that it relies on liboqs's "default" algorithm mechanism rather than naming each PQ algorithm specifically. liboqs provides an interface to use each of its supported algorithms at runtime, as well as a generic function for a "default" algorithm, where the mapping of the default algorithm has to be changed at compile-time. To simplify this preliminary prototype implementation of PQ algorithms in TLS 1.2 via OpenSSL 1.0.2, there are just two ciphersuites: one using liboqs' default algorithm, and one using liboqs' default algorithm in hybrid with ECDH. This means that the TLS implementation will end up using whichever algorithm was configured at compile-time, but cannot switch between them at runtime. This is of course not suitable for a production implementation, but as a basic prototype still allows a developer or researcher who controls both a client and server to test how TLS performs with a specific post-quantum algorithm by recompiling.

The implementation negotiates the combination of algorithms (and parameterizations) together, primarily due to the simplicity of the implementation when doing so. Note that which curve to use for ECDH is negotiated using a separate mechanism in TLS 1.2 (the `NamedCurve` extension). The number of component algorithms in a hybrid mode is fixed to two.

The hybrid implementation conveys cryptographic data (public keys / ciphertexts) of the two KEMs within the existing `ServerKeyExchange` and `ClientKeyExchange` messages, respectively, by concatenating the public keys / ciphertexts of the two algorithms, and then parsing them on receipt.

To compute the combined shared secret, the implementation uses the concatenation method as described above, in particular the premaster secret is the concatenation of the shared secrets from the two algorithms. The premaster secret is used directly in the existing TLS 1.2 KDF, without including any public keys or ciphertexts in the KDF input.

For the experimental results on key exchange in TLS 1.2 using OpenSSL 1.0.2 reported in Table 1, the hybrid key exchange was ECDH with the nistp256 curve, and the authentication method was RSA-3072 signatures.

3.1.3 An example instantiation: s2n

This subsection describes an alternate approach to hybrid key exchange in TLS 1.2, implemented by AWS Cryptography in Amazon's TLS implementation, s2n [2]. The changes to the TLS handshake are formally specified in an IETF draft [11]. The draft and implementation in s2n only define hybrid ciphersuites with exactly one classical key exchange component and exactly one post-quantum key exchange component; in particular they do not implement PQ-only ciphersuites.

s2n modifies the TLS handshake so that the two key exchanges are performed simultaneously and independently. It defines new ciphersuites where the key exchange mechanism is a hybrid between ECDHE and a post-quantum KEM. For example, `TLS_ECDHE_SIKE_ECDSA_WITH_AES_256_GCM_SHA384` is a hybrid ciphersuite with ECDHE for the classical component and SIKE for the PQ component of the key exchange.

The ciphersuite specifies a PQ key exchange *algorithm*, but leaves the *parameters* to an (optional) `ClientHello` extension. It uses a single extension to specify parameters for all PQ schemes, rather than using a different extension for each scheme, simplifying the process of extending the draft and implementation to add support for more PQ KEMs.

After the server accepts a proposed hybrid ciphersuite and selects parameters, it generates a classical ECDHE key pair and a PQ KEM key pair. Both public keys are sent in the `ServerKeyExchange` message by adding a new field for the KEM key. The client runs the KEM encapsulation algorithm and sends a KEM ciphertext in a new field of an augmented `ClientKeyExchange` message. Finally, the server runs the KEM decapsulation algorithm to obtain the KEM secret.

This process produces an ECDHE secret Z and a post-quantum secret K. The TLS premaster secret is the concatenation $Z||K$ of these values. The master secret is derived from the premaster secret using the standard TLS 1.2 KDF, except that the draft also extends the TLS PRF seed by concatenating the `ClientKeyExchange` message as suggested in [5, Section 3.2]. This ensures that the hybrid key exchange is provably secure.

For the experimental results on key exchange in TLS 1.2 using s2n reported in Table 1, the hybrid key exchange was ECDH with the nistp256 curve, and the authentication method was RSA-2048 signatures.

As described in Section 3.1.2, the OQS fork of OpenSSL 1.0.2 implements hybrid key exchange in TLS 1.2 using a different method, but the OQS team has plans to complement that with one that aligns with the specification accompanying the s2n implementation [11] to achieve interoperability.

3.1.4 Lessons learned

Ease of implementation. The design choices for the OpenSSL instantiation led to a speedy implementation with relatively few changes in the OpenSSL codebase and no changes to the TLS 1.2 protocol structure. The s2n instantiation required a bit more work to add a `ClientHello` extension and to extend the `ClientKeyExchange` and `ServerKeyExchange` messages.

Combinatorial explosion. Both instantiations had to add new ciphersuite identifiers corresponding to the new ciphersuites. Because TLS 1.2 negotiates almost everything all at once in a combined ciphersuites, there is a "combinatorial explosion" of identifiers: a single ciphersuite contains the key exchange method (RSA versus finite-field Diffie–Hellman versus ECDH), the authentication method (RSA versus ECDSA), the symmetric cipher (AES-128-CBC, AES-128-GCM, AES-256-CBC, AES-256-GCM, Triple-DES-CBC, and more) and the hash function (SHA-1, SHA-2).

Adding post-quantum and hybrid algorithms to the ciphersuite would further explode the list. As noted in footnote 3, the OpenSSL instantiation actually used an indirect method with a single post-quantum algorithm identifier, and fixed the PQ algorithm at compile-time, which is sufficient for limited prototyping but not suitable for production use. The s2n instantiation limited the explosion by only supporting two PQ KEMs (BIKE and SIKE), and also by negotiating PQ parameters in a `ClientHello` extension rather than as part of the ciphersuite.

Which solution should be used in the long run depends on how many algorithms are selected for standardization by NIST. If a large number of PQ algorithms are standardized, then it may be preferable to follow the approach taken for ECDH and [11] in TLS 1.2 and use a separate extension to negotiate the PQ parameters, despite the requirement to add a new extension and more negotiation logic. However if a small number of PQ algorithms are standardized, then it maybe preferable to put those algorithms directly into the ciphersuite and accept the accompanying combinatorial explosion.[4]

Large PQ message sizes. Handshake protocol messages in TLS 1.2 are limited to 2^{24} bytes, fragmented into packets of 2^{14} bytes or smaller. However, OpenSSL 1.0.2 has some limits smaller than this that affected some post-quantum KEMs, as observed in Table 1. FrodoKEM-1344-AES and FrodoKEM-1344-SHAKE failed with OpenSSL 1.0.2 initially, as the OpenSSL 1.0.2 code limits `ClientKeyExchange` messages to 20,480 bytes,[5] but Frodo-1344 public keys / ciphertexts are approximately 21,600 bytes. Increasing this to a larger value, e.g. 30,000 bytes, allowed the Frodo-1344 methods to succeed.

NTS-KEM(12,64) public keys are larger still – 319,488 bytes. In principle, TLS 1.2 handshake messages can be up to 2^{24} bytes in length. Despite this, the OQS team could not get NTS-KEM(12,64) working in OpenSSL 1.0.2: increasing the aforementioned bound in OpenSSL to something larger than 319,488 led to errors in other places in the code, which they had not been able to overcome at the time of writing.

No duplication. Because the key exchange algorithm is negotiated before any cryptographic data is sent in TLS 1.2, no unnecessary or duplicate data is sent, even in the OpenSSL instantiation where concatenation is used.

Use in applications. The OQS team has used its fork of OpenSSL in a range of scenarios with effectively no changes. The OpenSSL command-line test programs `s_client` and `s_server` can be used to demonstrate OpenSSL's TLS functionality directly, and work without modifications.

[4] This explosion can be somewhat limited by not creating ciphersuites for every possible combination: e.g. one could choose to skip ECDH+PQ+RC4+SHA1, and focus solely on adding PQ to ciphersuites with strong symmetric ciphers.

[5] `ssl/s3_srvr.c`, line 2310

There are a large number of applications built upon OpenSSL. The OQS team has successfully used the Apache web server [3] and the Links command-line web browser [50] with PQ and hybrid ciphersuites from OQS's OpenSSL 1.0.2 fork, simply by recompiling the applications against their version of OpenSSL and specifying the desired ciphersuite in a run-time configuration option. A team from Microsoft Research [15] also successfully built a post-quantum version of the OpenVPN virtual private networking tool [39] based on this fork.

Extensibililty. The initial design of [11] (version 00) used unique extensions to specify the parameters of each PQ algorithm. Since the draft only defined two PQ schemes, this only required two new extensions. However, this design decision made it difficult for others to extend the draft with new algorithms. Thus in the current version of the draft, clients specify the parameters they support for all PQ algorithms in a single `ClientHello` extension.

3.2 Key exchange in TLS 1.3

TLS 1.3 provides many efficiency and security benefits over its predecessor. Although the final specification approved by the IETF does not support quantum-safe cryptography, improved modularity in TLS 1.3's design makes it more amenable to supporting quantum-safe cryptography. The OQS team implemented post-quantum and hybrid key exchange in TLS 1.3 in a fork of OpenSSL 1.1.1 [38] using KEMs from liboqs.

3.2.1 Design considerations

An Internet-Draft by Stebila, Fluhrer, and Gueron [44], upon which part of this document is based, details various design choices for TLS 1.3 along the different axes identified in Section 2.2. We describe some of those here.

Negotiation. TLS 1.3's overall design for negotiation is different from TLS 1.2 in that it does away with the idea of monolithic ciphersuites that negotiate all cryptographic choices at once; instead TLS 1.3 negotiates each component (symmetric cipher, digital signature scheme, key exchange method) separately. Ephemeral key exchange in TLS 1.3 as standardized is based entirely on elliptic curve Diffie–Hellman. A `supported_groups` extension is used to negotiate which named elliptic curve to use. Negotiating hybrid key exchange algorithms in TLS 1.3 could take several approaches.

For negotiating each hybrid component algorithm individually, [42] proposed adding a second extension with a second list of key exchange methods. [44, §3.1.2] proposed two other options for individual algorithm negotiation which use delimiters within the existing `supported_groups` extension.[6] All of these approaches require some change in negotiation logic.

For negotiating hybrid algorithms as a combination, one could define new entries for the `supported_groups` list for each desired combination, as in Section 3.1.2 and in [26]; the identifiers for these new entries have no internal structure, and those require no new processing logic. By contrast, [52] and [44, §3.1.3.3] describe more complicated representations of combinations of algorithms with an internal structure that requires additional processing logic for negotiation.

Conveying cryptographic data. One could just concatenate public keys / ciphertexts in the `key_share` extension in the `ClientHello` and `ServerHello` message, as in [26, 52]. This is simple, but has one drawback, which is that it can result in duplication or additional round trips. For example, suppose a client wants to negotiate ECDH with old servers, and ECDH+PQ with hybrid-aware servers. If the client sends just an ECDH+PQ concatenated public key, an old server will not know

[6] Technically these are not groups in the mathematical sense, but the "groups" terminology is from the TLS 1.3 specification and we use it synonymously with key exchange mechanism for this discussion.

how to parse the ECDH portion from the concatenated public key, without triggering an extra round trip with the `HelloRetryRequest` message. The client must send one key share containing just an ECDH public key, and another keyshare containing an ECDH public key concatenated with a post-quantum key, thus sending two ECDH public keys. Admittedly ECDH public keys happen to be small compared to most PQ public keys, but wasted bytes should be avoided where possible.

However, TLS 1.3 actually allows for the `ClientHello key_share` extension to contain multiple public keys from different algorithms, so additional public keys could be included here. (However the `ServerHello key_share` extension only allows a single public key, so an alternative would have to be identified for the server's response.)

Alternatively, [42] adds extensions to the `ClientHello` and `ServerHello` messages for sending additional key shares.

Combining shared secrets. The basic approaches of concatenating, XORing, or KDFing together the shared secrets from each algorithm as described for TLS 1.2 in Section 3.1.2 can also be applied in TLS 1.3. The TLS 1.3 key schedule is more complex than the TLS 1.2 schedule, but conveniently hashes the transcript into the key derivation, so results on safely combining KEM keys from [6, 18] more readily apply. The more complex key schedule provides additional options for combining shared secrets in hybrid key exchange: for example [42] suggests adding a new step to the key schedule for each additional key exchange algorithm in the hybrid mode.

Size limits. The maximum size of a `key_exchange` value in a `key_share` extension in the `ClientHello` is $2^{16}-1$ bytes [41, §4.2.8], which is smaller than public keys/ciphertexts of some Round 2 submissions.

3.2.2 An example instantiation: OpenSSL 1.1.1

The OQS team implemented both PQ-only and hybrid key exchange in OpenSSL 1.1.1.[7]

For negotiation, the basic approach is to define "groups" for the `supported_groups` extension for each new PQ or hybrid scheme (pretending to be elliptic curves for the purposes of negotiation). PQ-only algorithms are negotiated by a new algorithm identifier directly. Hybrid algorithms are negotiated by the combined method, where each combination is a new `NamedGroup` entry with no internal structure to the identifier. As noted above, this means no new negotiation logic is required. The number of algorithms combined in a hybrid mode to fixed to two at a time.

The implementation uses the concatenation approach to convey public keys. This work was started before the TLS 1.3 standard was completed, and before OpenSSL had complete support for the updated protocol. In particular, at the time OpenSSL only supported one `key_share` extension, ruling out some more complicated integrations noted above. This implementation therefore chose an approach that was easy to prototype, and would give a quick indication on how post-quantum algorithms perform in TLS 1.3. Alternatives going forward are discussed in Section 4.1.1.

For computing the shared secret, the implementation concatenates the individual shared secrets and uses them in place of the original ECDH shared secret in the TLS 1.3 key schedule.

For the experimental results on key exchange in TLS 1.3 using OpenSSL 1.1.1 reported in Table 1, the hybrid key exchange was ECDH with the nistp256 curve, and the authentication method was RSA-3072 signatures.

[7] Unlike OQS's TLS 1.2 implementation in OpenSSL 1.0.2, this implementation doesn't rely on liboqs' default identifier; each PQ algorithm gets its own full-fledged identifier in OpenSSL 1.1.1.

3.2.3 Lessons learned

Ease of implementation. The OpenSSL library as currently written is not architected in a way that made modifications as general as would be desired. Note that OpenSSL is broadly structured in two components: libcrypto, which implements cryptographic primitives, and libssl, which implements SSL and TLS by relying on libcrypto for its cryptography. Since a Diffie–Hellman-like key exchange method is expected, the TLS layer calls into the crypto layer using a DH "generate key" and "generate message" API. One the other hand, NIST submissions are KEMs, and the liboqs library used has a 3-step KEM-style API. The lower-level crypto API in OpenSSL does not have the context of the TLS-level caller, so the crypto implementation can't know if it is being called from the client or the server side of the TLS layer. Because of these limitations, the OQS team could not integrate KEM key exchange schemes cleanly at the crypto layer of the OpenSSL library; it must instead do so at the TLS layer itself, forwarding calls to OQS as needed. Perhaps future versions of OpenSSL will provide a KEM API from libcrypto as the KEM formalism becomes more widespread.

(Lack of) combinatorial explosion. The à la carte negotiation approach of TLS 1.3 made parts of the implementation even easier compared to the TLS 1.2 implementation, since it avoided the combinatorial explosion that came from using a single identifier for the full ciphersuite.

Large PQ message sizes. As observed in Table 1, there were some failures involving KEMs with larger public keys and/or ciphertexts.

FrodoKEM-1344-AES and FrodoKEM-1344-SHAKE failed, due to an "excessive message size" error with respect to the `ServerHello` message, containing the KEM ciphertext. For Frodo1344, this ciphertext is 21,632 bytes. The TLS 1.3 specification limits the size of public keys in the `KeyShare` extension to $2^{16} - 1 = 65{,}535$ bytes, which should be large enough to accommodate Frodo1344 key shares. However, OpenSSL 1.1.1 has smaller limits; in particular the maximum size of the `ServerHello` message is constrained to 20,000 bytes.[8] Increasing this to a larger value, e.g. 30,000 bytes, allowed Frodo1344-based key exchanges to succeed.

NTS-KEM(12,64) key exchanges failed since the public key size (319,488 bytes) is larger than the maximum size allowed for key shares in the TLS 1.3 specification (65,535 bytes) [41, §4.2.8]. Other NTS-KEM parameter sets have even larger public keys. Similarly, both parameter sets for Classic McEliece have public keys larger than the TLS 1.3 limit.

Use in applications. The OpenSSL command-line test programs `s_client` and `s_server` can be used to demonstrate OpenSSL's TLS 1.3 functionality directly, and work without modifications. In addition, the OQS team has successfully used the nginx web server [34] in conjunction with OpenSSL's `s_client` to establish PQ and hybrid connections (using both KEX and authentication).

Because OpenSSL 1.1.1 has public API changes compared to the long-lived OpenSSL 0.9 and 1.0 series, major applications are only gradually coming be updated to build against OpenSSL 1.1.1.

3.3 Key exchange in SSHv2

At the highest level, SSH version 2 has a similar architecture to TLS, with an initial negotiation, followed by establishment of an authenticated session key via key exchange and digital signatures, which then is used in symmetric authenticated encryption.

[8] `ssl/statem/statem_locl.h`, constant `SERVER_HELLO_MAX_LENGTH`

3.3.1 Design considerations

Negotiation. SSHv2 is designed for algorithm agility; one notable difference is that algorithm identifiers in SSHv2 are strings, rather than numbers or binary codes, and the list of supported algorithms is just a comma-separated list of algorithm strings. PQ algorithms can be added directly as new strings. Hybrid combinations can be added as new strings naming both algorithms.

Conveying cryptographic data. In SSHv2, each key exchange method gets to define its own message format for its messages, so it is possible for hybrid key exchange methods to provide distinct fields for each component value.

Combining shared secrets. The output of key exchange in SSHv2 is a shared secret K and an "exchange hash" H; symmetric keys are then derived by hashing K and H with various labels. The computation of the exchange hash H is specified by the key exchange mechanism, but in all cases includes a subset of the transcript including identification strings, negotiation messages, and ephemeral public keys, as well as the shared secret K. The basic approaches of concatenating, XORing, or KDFing together the shared secrets from each algorithm as described for TLS 1.2 in Section 3.1.1 can all be employed.

Size limits. Message lengths in SSHv2 are represented by 4-byte length fields, theoretically accommodating 2^{32}-byte messages, large enough for all Round 2 submissions. However, RFC 4253 [54, §6.1] only requires that implementations be able to process packets containing payloads of size 32,768 bytes, and "SHOULD" be able to process larger packets. OpenSSH has a `PACKET_MAX_SIZE` of 2^{18} = 262,144 bytes, which, while larger than the minimum value required by the RFC, is smaller than the public keys needed for two some Round 2 candidates. In particular, while the SSHv2 specification can theoretically accommodate all Round 2 candidates, all versions of NTS-KEM [1] and both parameter sets for Classic McEliece [4] have public keys that are larger than OpenSSH's 2^{18}-byte limit. See Table 1 for a complete list of algorithms tested with OpenSSH.

3.3.2 An example instantiation: OpenSSH 7.9

A pre-Internet-Draft document by Hansen et al. [20] (not submitted to the IETF) describes the basic approach to how PQ and hybrid key exchange was implemented in the Open Quantum Safe project's fork of OpenSSH[9] [36].

Negotiation is as above; hybrid algorithms have new strings naming both algorithms such as `ecdh-nistp384-bike1-L1-sha384@openquantumsafe.org`. Public keys and ciphertexts are conveyed in specific fields added to the relevant key exchange message. Shared secrets are combined using concatenation.

For the experimental results on key exchange in SSHv2 using OpenSSH 7.9 reported in Table 1, the hybrid key exchange was ECDH with the nistp384 curve, and the authentication method was RSA-3072 signatures.

3.3.3 Lessons learned

There were no unusual circumstances in implementing PQ and hybrid key exchange in SSHv2, except that NTS-KEM could not be used due to too-large public keys. Though this implementation does not include Classic McEliece, we reiterate that both parameter sets for that algorithm also have public keys that are too large for OpenSSH to handle; further modifications to OpenSSH would

[9] [20] only describes the BIKE and SIKE KEMs, but the implementation in [36] includes additional KEMs from liboqs.

be required to support Classic McEliece and NTS-KEM. The implementors attempted to increase constants in OpenSSH to get it to work for NTS-KEM, but were not successful as of the time of writing.

À la carte negotiation of individual cryptographic components avoids combinatorial explosion like in TLS 1.2; however there does not appear to be a way to extend the `SSH_MSG_KEXINIT` negotiation message to provide a separate list for individually negotiating component algorithms of a hybrid mode.

4 Authentication case studies

We now turn to case studies of adding post-quantum and hybrid authentication to the TLS and SSH protocols. Authentication has an additional complication compared to key exchange: there is a long-term credential that must be stored and distributed. In TLS, X.509 certificates are used for long-term credentials; in SSHv2, the usual format is a raw public key (there are some proposals for use of X.509 [23] or other certificates, but raw public keys remain dominant).

4.1 Authentication in TLS 1.3

Although there have been several Internet-Drafts and experimental implementations of PQ and/or hybrid key exchange in TLS as noted in Section 3.1, none of those works considered PQ or hybrid authentication. This is likely to due to the general consensus that confidentiality against quantum adversaries is a more urgent need than authenticity, since quantum adversaries could retroactively attack confidentiality of any passively recorded communication sessions, but could not retroactively impersonate parties establishing a (completed) communication session. Nonetheless, the advent of a quantum computer would mean that we would eventually need to migrate to post-quantum authentication, meriting some preliminary investigation.

While there will certainly a need for implementations to support both old (non-PQ) and new (PQ) algorithms for authentication and to be backwards compatible with implementations that have not yet been upgraded during a transition period, there may perhaps be a slightly weaker need for hybrid authentication than hybrid key exchange, since post-quantum authentication may not be activated until later in the PQ transition when algorithms have had more time to be studied compared to the need for quantum-resistant confidentiality well in advance of a quantum computer. Still we consider some of the issues with hybrid authentication below.

4.1.1 Design considerations

Negotiation. TLS 1.3 has two extensions to negotiate signature algorithms: the first is the `signature_algorithms_cert` extension is used to negotiate which algorithms are supported for signatures in certificates; and the second is the `signature_algorithms` extension for which algorithms are supported in the protocol itself. Both of these extensions are a list of algorithm identifiers. Effectively the same considerations apply for each of these as for the `supported_groups` extension for negotiating the key exchange method as described in Section 3.2.1: to negotiate hybrid components individually, additional lists could be added for each type, or delimiters could be used within the existing lists; to negotiate as a combination, new identifiers for each combination could be defined without internal structure, or with internal structure.

Conveying public keys. In TLS 1.3, public keys for authentication are usually conveyed via X.509 certificates. To convey public keys for multiple algorithms in a hybrid mode, one has to decide

whether to extend the TLS protocol to convey multiple certificates, or try to convey multiple keys within the same certificate.

With regards to conveying multiple certificates within the TLS protocol, the `Certificate` message in TLS 1.3 does have a certificate list which permits multiple certificates, which could theoretically be used for this purpose. Historically, this list was used to convey a single certificate chain from the end-entity certificate through requisite intermediate CAs, and was required to be ordered. The TLS 1.3 specification says "implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first." [41, §4.4.2] This suggests it may be possible to use multiple end-entity certificates with different algorithms in the list, though a survey of implementations would need to be made to check compatibility.

The alternative would be to have a single X.509 certificate contain multiple public keys. Again there are choices here: should the multiple algorithms be treated individually (finding different locations within the certificates to store the different keys) or combined (by concatenating them into an opaque data structure)? Similarly, how should a hybrid signature by the certificate authority be treated? Part of the answer to this question depends on whether the same certificate can be targetted to solely new hybrid-aware parties, or must be backwards-compatible with old non-hybrid-aware parties. Bindel et al. [7] and Kampanakis et al. [24] explore various ways for X.509 certificates to convey multiple keys and signatures in backwards-compatible ways.

Conveying signatures. Parties in TLS 1.3 sign the handshake transcript and convey that signature in the `CertificateVerify` message. For hybrid authentication, there would need to be a way to convey two signatures.

Unfortunately, the `CertificateVerify` message does not have any built-in way of being extended, so it could only be extended or duplicated with a change in the protocol's logic or state machine based on the result of negotiation. The simpler approach is to concatenate the two signatures into a single message; at this point in the protocol, the parties have already agreed to use a hybrid algorithm, so there is no backwards compatibility risk nor any fear of duplicating values.

With hybrid signatures, it should be noted that there is a question of what to sign: do both algorithms sign the message, or does one algorithm sign the output (signature) from the other algorithm? This is discussed in Bindel et al. [7], but the basic answer is that both algorithms should sign the same message (or at least the hash of that message).

Size limits. The maximum size of an X.509 certificate (or raw public key) in TLS 1.3 is $2^{24}-1$ bytes, which is large enough for all Round 2 submissions. Signature size in TLS 1.3 is limited to $2^{16}-1$ bytes, which is too small for some Round 2 signature schemes, for example Picnic-{L3,L5}-{FS,UR}.[10]

4.1.2 An example instantiation: OpenSSL 1.1

The OQS team's implementation in OpenSSL 1.1.1 added both PQ-only and hybrid authentication, including generation of X.509 certificates with those keys and signatures.

It takes the concatenation approach to defining hybrid combinations: new algorithm identifiers are defined for each desired combination (with no internal structure to the identifier); public keys are concatenated; both signatures are on the same data, and are concatenated. This approach allowed for a simpler integration that could be traced through deeper into OpenSSL's libcrypto layer (specifically its "envelope" (EVP) API) and enabling all functionality (basic signatures, certificate management, and TLS authentication) to be supported for hybrid algorithms.

[10] See Section 4.1.3 for a discussion on how to work around these limits.

Specifically for hybrid signatures, a traditional and a PQ signature are generated on the same data, and the resulting signatures are concatenated; the traditional and PQ keys are also concatenated when serialized. The signed data is first hashed using the SHA-2 hash function matching the security level of the PQ scheme (SHA-256 for NIST level 1, SHA-384 for NIST levels 2 or 3, SHA-512 for NIST levels 4 or 5) before being signed by the traditional algorithm (which can't support arbitrarily long messages), but is passed directly to the PQ signature API (which handles arbitrarily long messages, typically via hash-and-sign). The hybrid scheme is identified as a new combo scheme with a unique identifier. Currently, the traditional algorithms in hybrid mode are ECDSA with nistp256 and RSA-3072 with NIST level 1 PQ schemes, and ECDSA with nistp384 with level 3 schemes.

For the experimental results on authentication in TLS 1.3 using OpenSSL 1.1.1 reported in Table 2, the classical signature algorithm used was RSA-3072 and the key exchange method used was ECDH with the nistp256 curve. A single certificate was sent from the server to the client, representing either the scenario where a self-signed certificate is used, or the scenario where the certificate authority is pre-installed in the client, and only the end-entity certificate needs to be sent. In the certificates, the signing algorithm and subject public key algorithm were the same. Longer certificate chains with intermediate CAs were not tested.

4.1.3 Lessons learned

Ease of implementation. As noted above, supporting post-quantum authentication requires support in more places through the codebase since certificates come into play.[11] For hybrid, the concatenation approach of making combined algorithms allowed for a simpler implementation, since the hybrid signatures would be treated monolithically within the existing APIs, rather than needing to adapt every API to handle two certificates, two public keys, two signatures, etc.

Large PQ sizes. The OQS team encountered several different problems due to large PQ public key or signature sizes, some of which they were able to overcome, and some of which they were not. Recall that the maximum X.509 certificate size in TLS 1.3 is $2^{24} - 1$ bytes, while the maximum signature size (in the `CertificateVerify` message) is $2^{16} - 1$ bytes. However, OpenSSL 1.1.1's code base imposes some additional constraints:

- OpenSSL 1.1.1 limits the default maximum size of the `Certificate` message, which includes the chain of all X.509 certificates except the root, to 102,400 bytes,[12] though this value can be set by a calling application at runtime.[13] Raising this value to $2^{24} - 1$ bytes allowed all remaining Rainbow schemes to work (though Rainbow-Ia-Cyclic and Ia-Cyclic-Compressed worked even without this fix).
- OpenSSL 1.1.1 limits the maximum size of a signature in the `CertificateVerify` message to 2^{14} bytes. This is too small for many post-quantum signatures. Raising this value to $2^{16} - 1$,[14] allowed the following schemes to work: Picnic-L1-{FS,UR}, Picnic2-{L3,L5}-FS, and all SPHINCS+ variants (though SPHINCS+ 128s variants worked even without this fix).

The Picnic-{L3,L5}-{FS,UR} schemes have signatures that exceed $2^{16} - 1$ bytes, and thus cannot be be accommodated within the current TLS 1.3 specification. This is also the case for TLS 1.2. An earlier experimental version of the OQS fork of OpenSSL 1.0.2 included post-quantum authentication

[11] Several people have asked us about using OpenSSL's ENGINE API for adding new algorithms. As far as we can tell, OpenSSL's ENGINE API can be used to add new implementations of existing algorithms, but cannot be used to add new algorithms, since algorithm identifiers, data structures, and framework code must be inserted manually into many places in the code.

[12] `ssl/statem/statem_clnt.c`, line 982, referencing `include/openssl/ssl.h` constant `SSL_MAX_CERT_LIST_DEFAULT`

[13] Using the `SSL_CTX_set_max_cert_list` function

[14] `ssl/statem/statem_clnt.c`, line 984

in TLS 1.2, and the OQS team successfully patched the code to allow a larger signature size ($2^{24}-1$ rather than $2^{16}-1$ by increasing a 2-byte length field to a 3-byte length field), and were subsequently able to use larger signatures successfully. This suggests that the TLS 1.3 specification could be altered to allow larger signatures, although some flag must then be used to communicate that larger length fields are being used. This may however affect compatibility with middle boxes, for example.

4.2 Authentication in SSHv2

Now we examine adding post-quantum and hybrid authentication in SSHv2 via OpenSSH.

4.2.1 Design considerations

Negotiation. Similarly to negotiation of key exchange in SSHv2, authentication is negotiated using a list of comma-separated strings, to which we can add PQ algorithms and hybrid combinations as new strings.

Conveying public keys. SSHv2 primarily uses raw public keys for authentication. Each authentication method can define its own format for the "public key blob" value, so it is possible for hybrid authentication methods to provide distinct fields for each component value.

Conveying signatures. The signature value is also algorithm-defined, so can easily accommodate concatenated signatures.

Message sizes. Message lengths in SSHv2 are represented by 4-byte length fields, theoretically accommodating 2^{32}-byte messages, large enough for all Round 2 submissions. However, RFC 4253 [54, §6.1] only *requires* that implementations be able to process packets containing payloads of size 32,768 bytes, and "SHOULD" be able to process larger packets. This means that while some schemes pose a challenge for TLS 1.3, they may be usable with compliant SSHv2 implementations.

4.2.2 An example instantiation: OpenSSH 7.9

The implementation in OQS's fork of OpenSSH v7.9 added both PQ-only and hybrid authentication.

For hybrid modes, the basic approach is concatenation.

For negotiation, new key types have been defined for the hybrid cases, identified by concatenating algorithm names; the implementation supports RSA-3072 or ECDSA with nistp256 (for NIST level 1 schemes) or nistp384 (for NIST levels 2–5) as the traditional algorithm.

Public keys are serialized sequentially: the traditional key is serialized first, followed by the PQ one. The SSH key encoding contains all the length and serialization information, so the OpenSSH serialization for each type is called sequentially. These concatenated public keys are used both on the wire and in local keystores.

The traditional and PQ signature are generated on the same data, and the resulting signatures are concatenated. The OpenSSH signature code is called sequentially: the traditional handling is performed first (including hashing the signed data with the appropriate SHA-2 functions (SHA-256 for NIST level 1, SHA-384 for NIST levels 2 or 3)), followed by the PQ one (in which case the data is signed/verified directly).

For the experimental results on authentication in SSHv2 using OpenSSH 7.9 reported in Table 2, the hybrid signature algorithm used was ECDSA with nistp256 and the key exchange method used was ECDH with the nistp384 curve.

4.2.3 Lessons learned

Ease of implementation. For the most part, as with key exchange in SSHv2, adding methods for post-quantum and hybrid authentication using concatenation was relatively straightforward, especially due to the lack of X.509 certificates. However the OQS team did encounter some difficulties with some larger algorithms that may be due to implementation challenges, as described below.

Usage scenario. A general observation that applies both in key exchange and in authentication is that users of SSH may be less sensitive to slow downs from larger communications sizes or slower cryptographic computations of some PQ schemes due to the 1-on-1 usage scenario of SSH with infrequently established connections, compared to a TLS-enabled web server handling many concurrent connections from various clients.

Large PQ sizes. As noted above, the SSHv2 spec can theoretically accommodate very large messages. However, OpenSSH has a variety of internal limits that prevent it from doing so, which were encountered in various ways with variants of the Rainbow scheme, which have the largest public keys among those tested. The Rainbow-Ia-Cyclic variants worked with OpenSSH directly. While the public key size of Rainbow-Ia-Classic (148,992 bytes) was smaller than OpenSSH's hard-coded `PACKET_MAX_SIZE` of 2^{18} bytes, it did not work without some modifications to the code, but it was eventually successfully integrated. The OQS team could not get any other Rainbow III and V variants to work, including the Rainbow-IIIc-Cyclic variants whose public keys are smaller than the 2^{18} bytes, but they could not trace through the sequence of failures in the code by the time of writing.

5 Conclusion and future work

The case studies we explored provide a preliminary investigation into approaches for implementing post-quantum and hybrid key exchange and authentication in TLS 1.2, TLS 1.3, and SSH, but they are certainly not exhaustive.

Standards bodies employing hybrid cryptography will have to make choices for the various design considerations discussed in this document, and may make different choices depending on their scenarios.

The case studies revealed some challenges in TLS and SSH standards and implementations with respect to limits on message sizes for key exchange and signatures that may affect some Round 2 submissions. Some size limits were implementation-imposed limits, and increasing those implementation-specific limits within standards-compliant limits enabled some additional schemes to function. Note that increasing those limits may have deleterious effects on performance on resistance to denial of service attacks, and should be investigated further. Some size limits were imposed by the standards; in some cases implementations experimented with increasing those size limits beyond the standards, and had some preliminarily positive results, but doing so would in deployment require carefully negotiating the use of larger length fields, and still risks compatibility with middle boxes and other pieces of the infrastructure. A few failures in the experimental results reported are not fully diagnosed, and may be resolvable with further engineering effort.

The experimental results to date from the case studies explored the size impacts of post-quantum KEMs and post-quantum signature schemes independently. An obvious next step is to evaluate the $O(n^2)$ combinations of KEMs and signatures for compliance with standards and implementation constraints.

The case studies above omitted some Round 2 submissions that have not yet been incorporated into their underlying frameworks. The OQS team intends to extend the OpenSSL and OpenSSH

implementations described in this report to include all Round 2 KEMs and signature schemes. The first step is to get all Round 2 KEMs and signature schemes into liboqs, which OQS is working towards with the help of the PQClean project [25], and we welcome external contributors. Once in liboqs, the algorithms will be enabled in the OpenSSL and OpenSSH forks.

Beyond compliance with standards and implementation constraints, future steps include benchmarking network performance:

- Network performance in lab conditions: Following the methodology of [8, 9, 19], how does latency and throughput behave on isolated networks in the lab?
- Network performance in more realistic conditions: Attempt to develop a simulation that reflects network conditions described by Langley [29] to assess latency and throughput in more realistic network conditions.

There are many other network protocols and applications also of interest for the post-quantum transition.

Acknowledgments

This ideas in this document are based on discussions with many people, including Matthew Campagna, Shay Gueron, and Torben Hansen (Amazon Web Services); Christopher Wood; Michele Mosca and John Schanck (University of Waterloo); and others. Goutam Tamvada assisted with some of the data collection.

Contributors to the Open Quantum Safe projects described in this document are listed on the relevant GitHub sites [35, 36, 37, 38], and include: Nicholas Allen, Maxime Anvari, Mira Belenkiy, Ben Davies, Nir Drucker, Javad Doliskani, Vlad Gheorghiu, Shay Gueron, Torben Hansen, Andrew Hopkins, Kevin Kane, Karl Knopf, Tancrède Lepoint, Shravan Mishra, Christian Paquin, Alex Parent, Peter Schwabe, Douglas Stebila, John Underhill, and Sebastian Verschoor.

The PQ algorithm implementations used in the experiments are directly or indirectly from the original NIST submission teams. Some of the PQ algorithm implementations in liboqs come indirectly via the PQClean project by Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Thom Wiggers (Radboud University) and D.S.

The Open Quantum Safe project has received funding from Amazon Web Services and the Tutte Institute for Mathematics and Computing, and in-kind contributions of developer time from Amazon Web Services, Cisco Systems, evolutionQ, and Microsoft Research. D.S. is supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada Discovery grant RGPIN-2016-05146 and a NSERC Discovery Accelerator Supplement.

References

[1] M. Albrecht, C. Cid, K. G. Paterson, C. J. Tjhai, and M. Tomlinson. NTS-KEM, 2019. URL: https://nts-kem.io/.

[2] Amazon Web Services. s2n. https://github.com/awslabs/s2n, 2014.

[3] Apache Software Foundation. Apache HTTP server project (httpd). URL: https://httpd.apache.org.

[4] D. J. Bernstein, T. Chou, T. Lange, I. von Maurich, R. Misoczki, R. Niederhagen, E. Persichetti, C. Peters, P. Schwabe, N. Sendrier, J. Szefer, and W. Wang. Classic McEliece: conservative code-based cryptography, 2019. URL: https://classic.mceliece.org/.

[5] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, and D. Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. Cryptology ePrint Archive, Report 2018/903, 2018. https://eprint.iacr.org/2018/903.

[6] N. Bindel, J. Brendel, M. Fischlin, B. Goncalves, and D. Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In J. Ding and R. Steinwandt, editors, *Post-Quantum Cryptography - 10th International Conference, PQCrypto 2019*. Springer, May 2019.

[7] N. Bindel, U. Herath, M. McKague, and D. Stebila. Transitioning to a quantum-resistant public key infrastructure. In T. Lange and T. Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 384–405. Springer, Heidelberg, 2017. doi:10.1007/978-3-319-59879-6_22.

[8] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 1006–1018. ACM Press, Oct. 2016. doi:10.1145/2976749.2978425.

[9] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila. Post-quantum key exchange for the TLS protocol from the ring learning with errors problem. In *2015 IEEE Symposium on Security and Privacy*, pages 553–570. IEEE Computer Society Press, May 2015. doi:10.1109/SP.2015.40.

[10] M. Braithwaite. Experimenting with post-quantum cryptography, July 2016. URL: https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html.

[11] M. Campagna and E. Crockett. Hybrid Post-Quantum Key Encapsulation Methods (PQ KEM) for Transport Layer Security 1.2 (TLS). Internet-Draft draft-campagna-tls-bike-sike-hybrid-01, Internet Engineering Task Force, May 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-campagna-tls-bike-sike-hybrid-01.

[12] M. Campagna, editor. Quantum safe cryptography and security: An introduction, benefits, enablers and challengers. Technical Report 8, June 2015. URL: https://www.etsi.org/images/files/ETSIWhitePapers/QuantumSafeWhitepaper.pdf.

[13] P. Chown. Advanced Encryption Standard (AES) ciphersuites for Transport Layer Security (TLS). RFC 3268, July 2002. URL: https://rfc-editor.org/rfc/rfc3268.txt, doi:10.17487/RFC3268.

[14] Y. Dodis and J. Katz. Chosen-ciphertext security of multiple encryption. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 188–209. Springer, Heidelberg, Feb. 2005. doi:10.1007/978-3-540-30576-7_11.

[15] K. Easterbrook, K. Kane, B. LaMacchia, D. Shumow, and G. Zaverucha. Post-quantum cryptography VPN, May 2018. URL: https://www.microsoft.com/en-us/research/project/post-quantum-crypto-vpn/.

[16] S. Even and O. Goldreich. On the power of cascade ciphers. In D. Chaum, editor, *CRYPTO'83*, pages 43–50. Plenum Press, New York, USA, 1983.

[17] S. Fluhrer, D. McGrew, P. Kampanakis, and V. Smyslov. Postquantum preshared keys for IKEv2. Internet-Draft draft-ietf-ipsecme-qr-ikev2-08, Internet Engineering Task Force, Mar. 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-ietf-ipsecme-qr-ikev2-08.

[18] F. Giacon, F. Heuer, and B. Poettering. KEM combiners. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 190–218. Springer, Heidelberg, Mar. 2018. doi:10.1007/978-3-319-76578-5_7.

[19] V. Gupta, D. Stebila, S. Fung, S. C. Shantz, N. Gura, and H. Eberle. Speeding up secure web transactions using elliptic curve cryptography. In *NDSS 2004*. The Internet Society, Feb. 2004.

[20] T. Hansen, M. Campagna, and E. Crockett. Pre-draft: Hybrid key exchange integration in the Secure Shell transport layer, June 2018. URL: https://github.com/open-quantum-safe/openssh-portable/blob/OQS-master/ietf_pre_draft_sike_bike_hybrid_kex.txt.

[21] D. Harnik, J. Kilian, M. Naor, O. Reingold, and A. Rosen. On robust combiners for oblivious transfer and other primitives. In R. Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 96–113. Springer, Heidelberg, May 2005. doi:10.1007/11426639_6.

[22] P. E. Hoffman. The transition from classical to post-quantum cryptography. Internet-Draft draft-hoffman-c2pq-05, Internet Engineering Task Force, May 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-hoffman-c2pq-05.

[23] K. Igoe and D. Stebila. X.509v3 certificates for Secure Shell authentication. RFC 6187, Mar. 2011. URL: https://rfc-editor.org/rfc/rfc6187.txt, doi:10.17487/RFC6187.

[24] P. Kampanakis, P. Panburana, E. Daw, and D. V. Geest. The viability of post-quantum X.509 certificates. Cryptology ePrint Archive, Report 2018/063, 2018. https://eprint.iacr.org/2018/063.

[25] M. J. Kannwischer, J. Rijneveld, P. Schwabe, D. Stebila, and T. Wiggers. The PQClean project, May 2019. URL: https://github.com/PQClean/PQClean.

[26] F. Kiefer and K. Kwiatkowski. Hybrid ECDHE-SIDH key exchange for TLS. Internet-Draft draft-kiefer-tls-ecdhe-sidh-00, Internet Engineering Task Force, Nov. 2018. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-kiefer-tls-ecdhe-sidh-00.

[27] A. Langley. Forward secrecy for Google HTTPS, Dec. 2011. URL: https://www.imperialviolet.org/2011/11/22/forwardsecret.html.

[28] A. Langley. CECPQ2, Dec. 2018. URL: https://www.imperialviolet.org/2018/12/12/cecpq2.html.

[29] A. Langley. Post-quantum confidentiality for TLS, Apr. 2018. URL: https://www.imperialviolet.org/2018/04/11/pqconftls.html.

[30] B. Moeller, N. Bolyard, V. Gupta, S. Blake-Wilson, and C. Hawk. Elliptic curve cryptography (ECC) cipher suites for Transport Layer Security (TLS). RFC 4492, May 2006. URL: https://rfc-editor.org/rfc/rfc4492.txt, doi:10.17487/RFC4492.

[31] National Institute of Standards and Technology. Specification for the Digital Signature Standard (DSS). Federal Information Professing Standards (FIPS) 186-2, Jan. 2000. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/186/2/archive/2001-10-05/documents/fips186-2-change1.pdf.

[32] National Institute of Standards and Technology. Specification for the Advanced Encryption Standard (AES). Federal Information Professing Standards (FIPS) 197, Nov. 2001. URL: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf.

[33] National Institute of Standards and Technology. Specification for the Secure Hash Standard. Federal Information Professing Standards (FIPS) 180-2, Aug. 2002. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf.

[34] NGINX, Inc. nginx. URL: https://www.nginx.com/.

[35] Open Quantum Safe Project. liboqs, July 2019. URL: https://github.com/open-quantum-safe/liboqs.

[36] Open Quantum Safe Project. OQS-OpenSSH, July 2019. URL: https://github.com/open-quantum-safe/openssh-portable.

[37] Open Quantum Safe Project. OQS-OpenSSL_1_0_2-stable, July 2019. URL: https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_0_2-stable.

[38] Open Quantum Safe Project. OQS-OpenSSL_1_1_1-stable, July 2019. URL: https://github.com/open-quantum-safe/openssl/tree/OQS-OpenSSL_1_1_1-stable.

[39] OpenVPN. OpenVPN. URL: https://openvpn.net.

[40] Qualys SSL Labs. SSL pulse, June 2019. URL: https://www.ssllabs.com/ssl-pulse/.

[41] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018. URL: https://rfc-editor.org/rfc/rfc8446.txt, doi:10.17487/RFC8446.

[42] J. M. Schanck and D. Stebila. A Transport Layer Security (TLS) extension for establishing an additional shared secret. Internet-Draft draft-schanck-tls-additional-keyshare-00, Internet Engineering Task Force, Apr. 2017. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-schanck-tls-additional-keyshare-00.

[43] J. M. Schanck, W. Whyte, and Z. Zhang. Quantum-safe hybrid (QSH) ciphersuite for Transport Layer Security (TLS) version 1.2. Internet-Draft draft-whyte-qsh-tls12-02, Internet Engineering Task Force, July 2016. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls12-02.

[44] D. Stebila, S. Fluhrer, and S. Gueron. Design issues for hybrid key exchange in TLS 1.3. Internet-Draft draft-stebila-tls-hybrid-design-01, Internet Engineering Task Force, July 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-stebila-tls-hybrid-design-01.

[45] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. Cryptology ePrint Archive, Report 2017/190, 2017. http://eprint.iacr.org/2017/190.

[46] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full SHA-1. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, Aug. 2017. doi:10.1007/978-3-319-63688-7_19.

[47] N. Sullivan. Haskell numeric prelude. Twitter, Sept. 2018. URL: https://twitter.com/grittygrease/status/1039656938768756736.

[48] The OpenSSL project. Changelog, May 2019. URL: https://www.openssl.org/news/changelog.html.

[49] C. Tjhai, M. Tomlinson, G. Bartlett, S. Fluhrer, D. V. Geest, O. Garcia-Morchon, and V. Smyslov. Framework to integrate post-quantum key exchanges into Internet Key Exchange protocol version 2 (IKEv2). Internet-Draft draft-tjhai-ipsecme-hybrid-qske-ikev2-03, Internet Engineering Task Force, Jan. 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-tjhai-ipsecme-hybrid-qske-ikev2-03.

[50] Twibright Labs. Links 2.17, Sept. 2018. URL: http://links.twibright.com.

[51] X. Wang, A. Yao, and F. Yao. New collision search for SHA-1. Crypto 2005 Rump Session, Aug. 2005. URL: https://www.iacr.org/conferences/crypto2005/r/2.pdf.

[52] W. Whyte, Z. Zhang, S. Fluhrer, and O. Garcia-Morchon. Quantum-safe hybrid (QSH) key exchange for Transport Layer Security (TLS) version 1.3. Internet-Draft draft-whyte-qsh-tls13-06, Internet Engineering Task Force, Oct. 2017. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-whyte-qsh-tls13-06.

[53] Wikipedia contributors. SHA-1 — Wikipedia, the free encyclopedia, 2019. [Online; accessed 29-May-2019]. URL: https://en.wikipedia.org/w/index.php?title=SHA-1&oldid=899014033.

[54] T. Ylonen and C. Lonvick. The Secure Shell (SSH) transport layer protocol. RFC 4253, Jan. 2006. URL: https://tools.ietf.org/html/rfc4253.

[55] R. Zhang, G. Hanaoka, J. Shikata, and H. Imai. On the security of multiple encryption or CCA-security+CCA-security=CCA-security? In F. Bao, R. Deng, and J. Zhou, editors, *PKC 2004*, volume 2947 of *LNCS*, pages 360–374. Springer, Heidelberg, Mar. 2004. doi:10.1007/978-3-540-24632-9_26.