The PQXDH Key Agreement Protocol

Ehren Kret Rolfe Schmidt

Revision 3 , 2023-05-24 Last updated: 2024-01-23

Contents

1. Introduction

This document describes the "PQXDH" (or "Post-Quantum Extended Diffie-Hellman") key agreement protocol. PQXDH establishes a shared secret key between two parties who mutually authenticate each other based on public keys. PQXDH provides post-quantum forward secrecy and a form of cryptographic deniability but still relies on the hardness of the discrete log problem for mutual authentication in this revision of the protocol.

PQXDH is designed for asynchronous settings where one user ("Bob") is offline but has published some information to a server. Another user ("Alice") wants to use that information to send encrypted data to Bob, and also establish a shared secret key for future communication.

2. Preliminaries

2.1. PQXDH parameters

An application using PQXDH must decide on several parameters:

Name	Definition
curve	A Montgomery curve for which XEdDSA [1] is specified, at present this is one of curve25519 or curve448
hash	A 256 or 512-bit hash function (e.g. SHA-256 or SHA-512)
info	An ASCII string identifying the application with a minimum length of 8 bytes
pqkem	A post-quantum key encapsulation mechanism that has IND-CCA post-quantum security (e.g. Crystals-Kyber-1024 [2])
aead	A scheme for authenticated encryption with associated data that has IND-CPA and INT-CTXT post-quantum security
EncodeEC	A function that encodes a *curve* public key into a byte sequence
DecodeEC	A function that decodes a byte sequence into a *curve* public key and is the inverse of *EncodeEC*
EncodeKEM	A function that encodes a *pqkem* public key into a byte sequence
DecodeKEM	A function that decodes a byte sequence into a *pqkem* public key and is the inverse of *EncodeKEM*

For example, an application could choose *curve* as curve25519, *hash* as SHA-512, *info* as "MyProtocol", and *pqkem* as CRYSTALS-KYBER-1024.

The ranges of all encoding functions must be pairwise disjoint.

The recommended implementation of *EncodeEC* consists of a single-byte constant representation of *curve* followed by little-endian encoding of the u-coordinate as specified in [3]. The single-byte representation of *curve* is defined by the implementer. Similarly the recommended implementation of *DecodeEC* reads the first byte to determine the parameter *curve*. If the first byte does not represent a recognized curve, the function fails. Otherwise it applies the little-endian decoding of the u-coordinate for *curve* as specified in [3].

The recommended implementation of *EncodeKEM* consists of a single-byte constant representation of *pqkem* followed by the encoding of the *pqkem* public key specified by *pqkem*. The single-byte representation of *pqkem* is defined by the implementer. Similarly the recommended implementation of *DecodeKEM* reads the first byte to determine the parameter *pqkem*. If the first byte does not represent a recognized key encapsulation mechanism, the function fails. Otherwise it applies the decoding specified by the selected key encapsulation mechanism.

2.2. Cryptographic notation

Throughout this document, all public keys have a corresponding private key, but to simplify descriptions we will identify key pairs by the public key and assume that the corresponding private key can be accessed by the key owner.

This document will use the following notation:

- The concatenation of byte sequences ***X*** and ***Y*** is ***X*** || ***Y***.
- ***DH(PK1, PK2)*** represents a byte sequence which is the shared secret output from an Elliptic Curve Diffie-Hellman function involving the key pairs represented by public keys *PK1* and *PK2*. The Elliptic Curve Diffie-Hellman function will be either the X25519 or X448 function from [3], depending on the *curve* parameter.
- ***Sig(PK, M, Z)*** represents the byte sequence that is a *curve* XEdDSA signature on the byte sequence *M* which was created by signing *M* with *PK*'s corresponding private key and using 64 bytes of randomness *Z*. This signature verifies with public key *PK*. The signing and verification functions for XEdDSA are specified in [1].
- ***KDF(KM)*** represents 32 bytes of output from the HKDF algorithm [4] using *hash* with inputs:
    - *HKDF input key material* = *F* || *KM*, where *KM* is an input byte sequence containing secret key material, and *F* is a byte sequence containing 32 0xFF bytes if *curve* is curve25519, and 57 0xFF bytes if *curve* is curve448. As in in XEdDSA [1], *F* ensures that the first bits of the HKDF input key material are never a valid encoding of a scalar or elliptic curve point.

  - *HKDF salt* = A zero-filled byte sequence with length equal to the *hash* output length, in bytes.
  - *HKDF info* = The concatenation of string representations of the 4 PQXDH parameters *info*, *curve*, *hash*, and *pqkem* into a single string separated with '`_`' such as "`MyProtocol_CURVE25519_SHA-512_CRYSTALS-KYBER-1024`". The string representations of the PQXDH parameters are defined by the implementer.

- ***(CT, SS) = PQKEM-ENC(PK)*** represents a tuple of the byte sequence that is the KEM ciphertext, *CT*, output by the algorithm *pqkem* together with the shared secret byte sequence *SS* encapsulated by the ciphertext using the public key *PK*.

- ***PQKEM-DEC(PK, CT)*** represents the shared secret byte sequence *SS* decapsulated from a *pqkem* ciphertext using the private key counterpart of the public key *PK* used to encapsulate the ciphertext CT.

2.3. Roles

The PQXDH protocol involves three parties: **Alice**, **Bob**, and a **server**.

- **Alice** wants to send **Bob** some initial data using encryption, and also establish a shared secret key which may be used for bidirectional communication.

- **Bob** wants to allow parties like **Alice** to establish a shared key with him and send encrypted data. However, **Bob** might be offline when **Alice** attempts to do this. To enable this, **Bob** has a relationship with some **server**.

- The **server** can store messages from **Alice** to **Bob** which **Bob** can later retrieve. The **server** also lets **Bob** publish some data which the server will provide to parties like **Alice**. The amount of trust placed in the server is discussed in Section 4.9.

In some systems the **server** role might be divided between multiple entities, but for simplicity we assume a single server that provides the above functions for **Alice** and **Bob**.

2.4. Elliptic Curve Keys

PQXDH uses the following elliptic curve public keys:

Name	Definition
IK_A	Alice's identity key
IK_B	Bob's identity key
EK_A	Alice's ephemeral key
SPK_B	Bob's signed prekey
$(OPK_B{}^1, OPK_B{}^2, \ldots)$	Bob's set of one-time prekeys

The elliptic curve public keys used within a PQXDH protocol run must either all be in curve25519 form, or they must all be in curve448 form, depending on the *curve* parameter [3].

Each party has a long-term identity elliptic curve public key (IK_A for Alice, IK_B for Bob).

Bob also has a signed prekey SPK_B, which he changes periodically and signs each time with IK_B, and a set of one-time prekeys $(OPK_B{}^1, OPK_B{}^2, \ldots)$, which are each used in a single PQXDH protocol run. For each signed prekey or one-time prekey, K, that Bob generates, he also computes an identifier, denoted *IdEC(K)*, that uniquely identifies this key on Bob's device. ("Prekeys" are so named because they are essentially protocol messages which Bob publishes to the server, along with their corresponding identifiers, prior to Alice beginning the protocol run.) These keys will be uploaded to the **server** as described in Section 3.2.

During each protocol run, Alice generates a new ephemeral key pair with public key EK_A.

2.5. Post-Quantum Key Encapsulation Keys

PQXDH uses the following post-quantum key encapsulation public keys:

Name	Definition
$PQSPK_B$	Bob's signed last-resort *pqkem* prekey
$(PQOPK_B{}^1, PQOPK_B{}^2, \ldots)$	Bob's set of signed one-time *pqkem* prekeys

The *pqkem* public keys used within a PQXDH protocol run must all use the same *pqkem* parameter.

Bob has a signed last-resort post-quantum prekey $PQSPK_B$, which he changes periodically and signs each time with IK_B, and a set of signed one-time prekeys $(PQOPK_B{}^1, PQOPK_B{}^2, \ldots)$ which are also signed with IK_B and each used in a single PQXDH protocol run. For each last-resort or ephemeral KEM key, *K*, that Bob generates, he also computes an identifier, denoted *IdKEM(K)*, that uniquely identifies this key on Bob's device. These keys and their corresponding identifiers will be uploaded to the **server** as described in Section 3.2. The name "last-resort" refers to the fact that the last-resort prekey is only used when one-time *pqkem* prekeys are not available. This can happen when the number of prekey bundles downloaded for Bob exceeds the number of one-time *pqkem* prekeys Bob has uploaded (see Section 3 for details about the role of the server). An implementation should provide Bob a way to identify whether a *pqkem* public key corresponds to a one-time *pqkem* key or a last-resort *pqkem* key.

3. The PQXDH protocol

3.1. Overview

PQXDH has three phases:

1. Bob publishes his elliptic curve identity key, elliptic curve prekeys, and *pqkem* prekeys to a server.
2. Alice fetches a "prekey bundle" from the server, and uses it to send an initial message to Bob.
3. Bob receives and processes Alice's initial message.

The following sections explain these phases.

3.2. Publishing keys

Bob generates a sequence of 64-byte random values Z_{SPK}, Z_{PQSPK}, Z_1, Z_2, … and publishes a set of keys to the server containing:

- Bob's *curve* identity key IK_B
- Bob's signed *curve* prekey and its identifier $(SPK_B, IdEC(SPK_B))$
- Bob's signature on the *curve* prekey $Sig(IK_B, EncodeEC(SPK_B), Z_{SPK})$
- Bob's signed last-resort *pqkem* prekey and its identifier $(PQSPK_B, IdKEM(PQSPK_B))$
- Bob's signature on the *pqkem* prekey $Sig(IK_B, EncodeKEM(PQSPK_B), Z_{PQSPK})$
- A set of Bob's one-time *curve* prekeys $(OPK_B{}^1, OPK_B{}^2, OPK_B{}^3, \ldots)$ along with their identifiers $(IdEC(OPK_B{}^1), IdEC(OPK_B{}^2), IdEC(OPK_B{}^3), \ldots)$
- A set of Bob's signed one-time *pqkem* prekeys $(PQOPK_B{}^1, PQOPK_B{}^2, PQOPK_B{}^3, \ldots)$ along with their identifiers $(IdKEM(PQOPK_B{}^1), IdKEM(PQOPK_B{}^2), IdKEM(PQOPK_B{}^3), \ldots)$
- The set of Bob's signatures on the signed one-time *pqkem* prekeys $(Sig(IK_B, EncodeKEM(PQOPK_B{}^1), Z_1), Sig(IK_B, EncodeKEM(PQOPK_B{}^2), Z_2), Sig(IK_B, EncodeKEM(PQOPK_B{}^3), Z_3), \ldots)$

Bob only needs to upload his identity key to the server once. However, Bob may upload new one-time prekeys at other times (e.g. when the server informs Bob that the server's store of one-time prekeys is getting low).

For both the signed *curve* prekey and the signed last-resort *pqkem* prekey, Bob will upload a new prekey along with its signature using IK_B at some interval (e.g. once a week or once a month). The new signed prekey and its signatures will replace the previous values.

After uploading a new pair of signed *curve* and signed last-resort *pqkem* prekeys, Bob may keep the private key corresponding to the previous pair around for some period of time to handle messages using it that may have been delayed in transit. Eventually, Bob should delete this private key for forward secrecy (one-time prekey private keys will be deleted as Bob receives messages using them; see Section 3.4).

3.3. Sending the initial message

To perform a PQXDH key agreement with Bob, Alice contacts the server and fetches a "prekey bundle" containing the following values:

- Bob's *curve* identity key IK_B
- Bob's signed *curve* prekey with its identifier $(SPK_B, IdEC(SPK_B))$
- Bob's signature on the *curve* prekey $Sig(IK_B, EncodeEC(SPK_B), Z_{SPK})$
- One of either Bob's signed one-time *pqkem* prekey ${PQOPK_B}^n$ or Bob's last-resort signed *pqkem* prekey $PQSPK_B$ if no signed one-time *pqkem* prekey remains. Call this key $PQPK_B$. The bundle also contains $IdKEM(PQPK_B)$
- Bob's signature on the *pqkem* prekey $Sig(IK_B, EncodeKEM(PQPK_B), Z_{PQPK})$
- (Optionally) Bob's one-time *curve* prekey ${OPK_B}^n$ and its identifier $IdEC({OPK_B}^n)$

The server should provide one of Bob's *curve* one-time prekeys if one exists and then delete it. If all of Bob's *curve* one-time prekeys on the server have been deleted, the bundle will not contain a one-time *curve* prekey element.

The server should prefer to provide one of Bob's *pqkem* one-time signed prekeys ${PQOPK_B}^n$ if one exists and then delete it. If all of Bob's *pqkem* one-time signed prekeys on the server have been deleted, the bundle will instead contain Bob's *pqkem* last-resort signed prekey $PQSPK_B$.

Alice verifies the signatures on the prekeys. If any signature check fails, Alice aborts the protocol. Otherwise, if all signature checks pass, Alice then generates an ephemeral *curve* key pair with public key $\mathrm{EK_A}$. Alice additionally generates a *pqkem* encapsulated shared secret:

$$
\begin{aligned}
&(\mathrm{CT}, \mathrm{SS}) = \text{PQKEM-ENC}(\mathrm{PQPK_B})\\
&\qquad\qquad \text{shared secret SS}\\
&\qquad\qquad \text{ciphertext CT}
\end{aligned}
$$

If the bundle does not contain a *curve* one-time prekey, she calculates:

$$
\begin{aligned}
&\mathrm{DH_1} = \mathrm{DH(IK_A, SPK_B)}\\
&\mathrm{DH_2} = \mathrm{DH(EK_A, IK_B)}\\
&\mathrm{DH_3} = \mathrm{DH(EK_A, SPK_B)}\\
&\mathrm{SK} = \mathrm{KDF(DH_1 \mid\mid DH_2 \mid\mid DH_3 \mid\mid SS)}
\end{aligned}
$$

If the bundle does contain a *curve* one-time prekey, the calculation is modified to include an additional *DH*:

$$
\begin{aligned}
&\mathrm{DH_4} = \mathrm{DH(EK_A, OPK_B)}\\
&\mathrm{SK} = \mathrm{KDF(DH_1 \mid\mid DH_2 \mid\mid DH_3 \mid\mid DH_4 \mid\mid SS)}
\end{aligned}
$$

After calculating *SK*, Alice deletes her ephemeral private key, the *DH* outputs and the shared secret *SS*.

Alice then calculates an "associated data" byte sequence *AD* that contains identity information for both parties:

$\mathrm{AD} = \mathrm{EncodeEC}(\mathrm{IK_A})\ ||\ \mathrm{EncodeEC}(\mathrm{IK_B})$

If *pqkem* does not incorporate $PQPK_B$ into the ciphertext, Alice must also append *EncodeKEM*($PQPK_B$) to *AD* (see the discussion in Section 4.12). Alice may optionally append additional information to *AD*, such as Alice and Bob's usernames, certificates, or other identifying information.

Alice then sends Bob an initial message containing:

- Alice's identity key IK_A
- Alice's ephemeral key EK_A
- The *pqkem* ciphertext *CT* encapsulating *SS* for $PQPK_B$
- Identifiers stating which of Bob's prekeys Alice used
- An initial ciphertext encrypted with some AEAD encryption scheme [5] using *AD* as associated data and using an encryption key which is either *SK* or the output from some cryptographic PRF keyed by *SK*.

The initial ciphertext is typically the first message in some post-PQXDH communication protocol. In other words, this ciphertext typically has two roles, serving as the first message within some post-PQXDH protocol, and as part of Alice's PQXDH initial message.

The initial message must be encoded in an unambiguous format to avoid confusion of the message items by the recipient.

After sending this, Alice deletes the ciphertext *CT* and may continue using *SK* or keys derived from *SK* within the post-PQXDH protocol for communication with Bob, subject to the security considerations discussed in Section 4.

3.4. Receiving the initial message

Upon receiving Alice's initial message, Bob retrieves Alice's identity key and ephemeral key from the message. Bob also loads his identity private key and uses the key identifiers to load the private key(s) corresponding to the signed prekeys, one-time prekeys, and KEM key Alice used.

Using these keys, Bob calculates *PQKEM-DEC*($PQPK_B$, *CT*) as the shared secret *SS* and repeats the *DH* and *KDF* calculations from the previous section to derive *SK*, and then deletes the *DH* values and *SS* values.

Bob then constructs the *AD* byte sequence using IK_A and IK_B as described in the previous section. Finally, Bob attempts to decrypt the initial ciphertext using *SK* and *AD*. If the initial ciphertext fails to decrypt, then Bob aborts the protocol and deletes *SK*.

If the initial ciphertext decrypts successfully, the protocol is complete for Bob. For forward secrecy, Bob deletes the ciphertext and any one-time prekey private key that was used. Bob may then continue using SK or keys derived from SK within the post-PQXDH protocol for communication with Alice subject to the security considerations discussed in Section 4.

4. Security considerations

The security of the composition of X3DH [6] with the Double Ratchet [7] was formally studied in [8] and proven secure under the Gap Diffie-Hellman assumption (GapDH)[9] while making simplifying assumptions that avoid modeling the reuse of IK_B for both key agreement and signing. PQXDH composed with the Double Ratchet retains this security against an adversary without access to a quantum computer, but strengthens the security of the initial handshake to require the solution of both GapDH and Module-LWE [10].

In [11] PQXDH has been formally analyzed in the symbolic model with ProVerif [12] and in the computational model with CryptoVerif [13]. With ProVerif, the authors prove both authentication and secrecy in the symbolic model and enumerate the precise conditions under which the attacker can break these properties. These security properties notably imply forward secrecy, resistance to harvest now decrypt later attacks, resistance to key compromise impersonation, and session independence.

Using the CryptoVerif prover, the authors prove the computational secrecy and authentication of any completed key exchange under the GapDH assumption for the X25519 curve, the UF-CMA assumption on XEdDSA (assuming no key reuse between XEdDSA and X25519), the hash function modeled as a random oracle, and the IND-CPA+INT-CTXT assumptions for the AEAD. Moreover, they also show forward secrecy when the signature was UF-CMA secure at the time the key exchange took place, assuming post-quantum IND-CCA security for the KEM, modelling the hash function as a PRF, and IND-CPA+INT-CTXT security for the AEAD.

For both PQXDH and X3DH, however, a full proof of security under a joint assumption of GapDH and UF-CMA security for X25519 and XEdDSA is still needed.

The remainder of this section discusses an incomplete list of further security considerations.

4.1. Authentication

Before or after a PQXDH key agreement, the parties may compare their identity public keys IK_A and IK_B through some authenticated channel. For example, they may compare public key fingerprints manually, or by scanning a QR code. Methods for doing this are outside the scope of this document.

Authentication in PQXDH is not quantum-secure. In the presence of an active quantum adversary, the parties receive no cryptographic guarantees as to who they are communicating with. Post-quantum secure deniable mutual authentication is an open research problem which we hope to address with a future revision of this protocol.

If authentication is not performed, the parties receive no cryptographic guarantee as to who they are communicating with.

4.2. Protocol replay

If Alice's initial message doesn't use a one-time prekey, it may be replayed to Bob and he will accept it. This could cause Bob to think Alice had sent him the same message (or messages) repeatedly.

To mitigate this, a post-PQXDH protocol may wish to quickly negotiate a new encryption key for Alice based on fresh random input from Bob. This is the typical behavior of Diffie-Hellman-based ratcheting protocols [7].

Bob could attempt other mitigations, such as maintaining a blacklist of observed messages, or replacing old signed prekeys more rapidly. Analyzing these mitigations is beyond the scope of this document.

4.3. Replay and key reuse

Another consequence of the replays discussed in the previous section is that a successfully replayed initial message would cause Bob to derive the same *SK* in different protocol runs.

For this reason, any post-PQXDH protocol that uses *SK* to derive encryption keys MUST take measures to prevent catastrophic key reuse. For example, Bob could use a DH-based ratcheting protocol to combine *SK* with a freshly generated *DH* output to get a randomized encryption key [7].

4.4. Deniability

Informally, cryptographic deniability means that a protocol neither gives its participants a publishable cryptographic proof of the contents of their communication nor proof of the fact that they communicated. PQXDH, like X3DH, aims to provide both Alice and Bob deniablilty that they communicated with each other in a context where a "judge" who may have access to one or more party's secret keys is presented with a transcript allegedly created by communication between Alice and Bob.

We focus on offline deniability because if either party is collaborating with a third party during protocol execution, they will be able to provide proof of their communication to such a third party. This limitation on "online" deniability appears to be intrinsic to the asynchronous setting [14].

PQXDH has some forms of cryptographic deniability. Motivated by the goals of X3DH, Brendel et al. [15] introduce a notion of 1-out-of-2 deniability for semi-honest parties and a "big brother" judge with access to all parties' secret keys. Since either Alice or Bob can create a fake transcript using only their own secret keys, PQXDH has this deniability property. Vatandas, et al. [16] prove that X3DH is deniable in a different sense subject to certain "Knowledge of Diffie-Hellman Assumptions". PQXDH is deniable in this sense for Alice, subject to the same assumptions, and we conjecture that it is deniable for Bob subject to an additional Plaintext Awareness (PA) assumption for *pqkem*. We note that Kyber uses a variant of the Fujisaki-Okamoto transform with implicit rejection [17] and is therefore not PA as is. However, in PQXDH, an AEAD ciphertext encrypted with the session key is always sent along with the Kyber ciphertext. This should offer the same guarantees as PA. We encourage the community to investigate the precise deniability properties of PQXDH.

These assertions all pertain to deniability in the classical setting. As discussed in [18] we expect that for future revisions of this protocol (that provide post-quantum mutual authentication) assertions about deniability against semi-honest quantum advsersaries will hold. Deniability in the face of malicious quantum adversaries requires further research.

4.5. Signatures

It might be tempting to omit the prekey signature after observing that mutual authentication and forward secrecy are achieved by the *DH* calculations. However, this would allow a "weak forward secrecy" attack: A malicious server could provide Alice a prekey bundle with forged prekeys, and later compromise Bob's IK_B to calculate *SK*.

Alternatively, it might be tempting to replace the DH-based mutual authentication (i.e. DH_1 and DH_2) with signatures from the identity keys. However, this reduces deniability, increases the size of initial messages, and increases the damage done if ephemeral or prekey private keys are compromised, or if the signature scheme is broken.

4.6. Key compromise

Compromise of a party's private keys has a disastrous effect on security, though the use of ephemeral keys and prekeys provides some mitigation.

Compromise of a party's identity private key allows impersonation of that party to others. Compromise of a party's prekey private keys may affect the security of older or newer *SK* values, depending on many considerations.

A full analysis of all possible compromise scenarios is outside the scope of this document, however a partial analysis of some plausible scenarios is below:

- If either an elliptic curve one-time prekey (OPK_B) or a post-quantum key encapsulation one-time prekey ($PQOPK_B$) are used for a protocol run and deleted as specified, then a compromise of Bob's identity key and prekey private keys at some future time will not compromise the older *SK*.

- If one-time prekeys were not used for a protocol run, then a compromise of the private keys for IK_B, SPK_B, and $PQSPK_B$ from that protocol run would compromise the *SK* that was calculated earlier. Frequent replacement of signed prekeys mitigates this, as does using a post-PQXDH ratcheting protocol which rapidly replaces *SK* with new keys to provide fresh forward secrecy [7].
- Compromise of prekey private keys may enable attacks that extend into the future, such as passive calculation of *SK* values, and impersonation of arbitrary other parties to the compromised party ("key-compromise impersonation"). These attacks are possible until the compromised party replaces his compromised prekeys on the server (in the case of passive attack); or deletes his compromised signed prekey's private key (in the case of key-compromise impersonation).

4.7. Passive quantum adversaries

PQXDH is designed to prevent "harvest now, decrypt later" attacks by adversaries with access to a quantum computer capable of computing discrete logarithms in *curve*. While this security is primarily derived from *pqkem*, it also requires that *aead* provides post-quantum IND-CPA and INT-CTXT security. There is great uncertainty in estimating post-quantum security strength of cryptosystems, making it challenging to define this requirement precisely. Taking the NIST evaluation criteria for post-quantum cryptography submissions [19] as a guide, it places a key-search attack on AES256 at its highest security level. While this does not correspond exactly to our security requirements, it suggests that using an appropriate AEAD mode of AES256 will suffice. We note some particular security properties of PQXDH in this setting.

- If an attacker has recorded the public information and the message from Alice to Bob, even access to a quantum computer will not compromise *SK*.
- If a post-quantum key encapsulation one-time prekey ($PQOPK_B$) is used for a protocol run and deleted as specified then compromise after deletion and access to a quantum computer at some future time will not compromise the older *SK*.
- If post-quantum one-time prekeys were not used for a protocol run, then access to a quantum computer and a compromise of the private key for $PQSPK_B$ from that protocol run would compromise the *SK* that was calculated earlier. Frequent replacement of signed prekeys mitigates this, as does using a post-PQXDH ratcheting protocol which rapidly replaces *SK* with new keys to provide fresh forward secrecy [7].

4.8. Active quantum adversaries

PQXDH is not designed to provide protection against active quantum attackers. An active attacker with access to a quantum computer capable of computing discrete logarithms in *curve* can compute $DH(PK_1, PK_2)$ and $Sig(PK, M, Z)$ for all elliptic *curve* keys PK_1, PK_2, and PK. This allows an attacker to impersonate Alice by using the quantum computer to compute the secret key corresponding to PK_A then continuing with the protocol. A malicious server with access to such a quantum computer could impersonate Bob by generating new key pairs $PQSPK'_B$ and $PQOPK'_B$, computing the secret key corresponding to PK_B, then using PK_B to sign the newly generated post-quantum KEM keys and delivering these attacker-generated keys in place of Bob's post-quantum KEM key when Alice requests a prekey bundle.

It is tempting to consider adding a post-quantum identity key that Bob could use to sign the post-quantum prekeys. This would prevent the malicious server attack described above and provide Alice a cryptographic guarantee that she is communicating with Bob, but it does not provide mutual authentication. Bob does not have any cryptographic guarantee about who he is communicating with. The post-quantum KEM and signature schemes being standardized by NIST [20] do not provide a mechanism for post-quantum deniable mutual authentication, although this can be achieved through the use of a post-quantum ring signature or designated verifier signature [15], [18]. We urge the community to work toward standardization of these or other mechanisms that will allow deniable mutual authentication.

4.9. Server trust

A malicious server could cause communication between Alice and Bob to fail (e.g. by refusing to deliver messages).

If Alice and Bob authenticate each other as in Section 4.1, then the only additional attack available to the server is to refuse to hand out one-time prekeys, causing forward secrecy for *SK* to depend on the signed prekey's lifetime (as analyzed in Section 4.6).

This reduction in initial forward secrecy could also happen if one party maliciously drains another party's one-time prekeys, so the server should attempt to prevent this (e.g. with rate limits on fetching prekey bundles).

4.10. Identity binding

Authentication as in Section 4.1 does not necessarily prevent an "identity misbinding" or "unknown key share" attack.

This results when an attacker ("Charlie") falsely presents Bob's identity key fingerprint to Alice as his (Charlie's) own, and then either forwards Alice's initial message to Bob, or falsely presents Bob's contact information as his own. The effect of this is that Alice thinks she is sending an initial message to Charlie when she is actually sending it to Bob.

To make this more difficult the parties can include more identifying information into *AD*, or hash more identifying information into the fingerprint, such as usernames, phone numbers, real names, or other identifying information. Charlie would be forced to lie about these additional values, which might be difficult.

However, there is no way to reliably prevent Charlie from lying about additional values, and including more identity information into the protocol often brings trade-offs in terms of privacy, flexibility, and user interface. A detailed analysis of these trade-offs is beyond the scope of this document.

4.11. Risks of weak randomness sources

In addition to concerns about the generation of the keys themselves, the security of the PQKEM shared secret relies on the random source available to Alice's machine at the time of running the ***PQKEM-ENC*** operation. This leads to a situation similar to what we face with a Diffie-Hellman exchange. For both Diffie-Hellman and Kyber, if Alice has weak entropy then the resulting shared secret will have low entropy when conditioned on Bob's public key. Thus both the classical and post-quantum security of *SK* depend on the strength of Alice's random source.

Kyber hashes Bob's public key with Alice's random bits to generate the shared secret, making Bob's key contributory, as it is with a Diffie-Hellman key exchange. This does not reduce the dependence on Alice's entropy source, as described above, but it does limit Alice's ability to control the post-quantum shared secret. Not all KEMs make Bob's key contributory and this is a property to consider when selecting *pqkem*.

4.12 Preventing KEM Re-encapsulation Attacks

Typically, when using a KEM that relies one a public key encryption to encrypt a fresh shared secret, the fresh shared secret is not tied to the public key. A shared secret corresponding to the encapsulation of a given compromised public key can easily be re-encapsulated against another uncompromised public key. IND-CCA security of the KEM does not prevent this behavior.

For a KEM for which this attack is possible, as soon as one PQPK is compromised, an attacker can force all initiators to use this compromised PQPK, and by always reencapsulating the shared secret against another fresh uncompromised PQPK, make the responder believe that nothing is going awry. This notably breaks the usual notion of session independence: compromising one PQPK of a responder can in fact impact the security of other sessions of the responder that should be using distinct and independent PQPKs.

The Kyber KEM incorporates the KEM public key into the generation of the shared secret, preventing this attack. For a generic IND-CCA KEM, this attack can be prevented by adding $PQPK_B$ to AD for the initial message. See [11] for more details about this attack and mitigations.

4.13 Key Identifiers

The public key identifiers are not security critical, notably as the actual values of the keys are signed or used within the AD. Note however that identifiers that would collide too often would cause decryption failures on the responder side, as the responder would try to complete the key exchange with the wrong public key, which would fail.

An application can choose to use public keys as key identifiers, but may choose an identifier with a smaller representation to reduce message sizes, provided that collisions are unlikely. Possible implementations include a hash of the public key, a random value, or sequentially generated values starting from a random offset.

5. IPR

6. Acknowledgements

The PQXDH protocol was developed by Ehren Kret and Rolfe Schmidt as an extension of the X3DH protocol [6] by Moxie Marlinspike and Trevor Perrin. Thanks to Trevor Perrin for discussions on the design of this protocol.

Thanks to Bas Westerbaan, Charlie Jacomme, Chris Peikert, Daniel Collins, Deirdre Connolly, Franziskus Kiefer, John Schanck, Jon Millican, Jordan Rose, Karthik Bhargavan, Loïs Huguenin-Dumittan, Peter Schwabe, Rune Fiedler, Shuichi Katsumata, Sofía Celi, Théophile Wallez, Tomas Duchac, and Yo'av Rieck for helpful discussions and editorial feedback.

Thanks to the Kyber team [21] for their work on the Kyber key encapsulation mechanism.

7. References

[1] T. Perrin, "The XEdDSA and VXEdDSA Signature Schemes," 2016. https://signal.org/docs/specifications/xeddsa/

[2] "Module-lattice-based key-encapsulation mechanism standard." https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.ipd.pdf

[3] A. Langley, M. Hamburg, and S. Turner, "Elliptic Curves for Security." Internet Engineering Task Force; RFC 7748 (Informational); IETF, Jan-2016. http://www.ietf.org/rfc/rfc7748.txt

[4] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." Internet Engineering Task Force; RFC 5869 (Informational); IETF, May-2010. http://www.ietf.org/rfc/rfc5869.txt

[5] P. Rogaway, "Authenticated-encryption with Associated-data," in Proceedings of the 9th ACM Conference on Computer and Communications Security, 2002. http://web.cs.ucdavis.edu/~rogaway/papers/ad.pdf

[6] M. Marlinspike and T. Perrin, "The X3DH Key Agreement Protocol," 2016. https://signal.org/docs/specifications/x3dh/

[7] T. Perrin and M. Marlinspike, "The Double Ratchet Algorithm," 2016. https://signal.org/docs/specifications/doubleratchet/

[8] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," J. Cryptol., vol. 33, no. 4, 2020. https://doi.org/10.1007/s00145-020-09360-1

[9] T. Okamoto and D. Pointcheval, "The gap-problems: A new class of problems for the security of cryptographic schemes," in Proceedings of the 4th international workshop on practice and theory in public key cryptography: Public key cryptography, 2001.

[10] A. Langlois and D. Stehlé, "Worst-case to average-case reductions for module lattices," Des. Codes Cryptography, vol. 75, no. 3, Jun. 2015. https://doi.org/10.1007/s10623-014-9938-4

[11] K. Bhargavan, C. Jacomme, and F. Kiefer, "PQXDH formal analysis git repository." https://github.com/Inria-Prosecco/pqxdh-analysis

[12] "ProVerif." https://bblanche.gitlabpages.inria.fr/proverif/

[13] "CryptoVerif." https://bblanche.gitlabpages.inria.fr/CryptoVerif/

[14] N. Unger and I. Goldberg, "Deniable Key Exchanges for Secure Messaging," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015. https://cypherpunks.ca/~iang/pubs/dake-ccs15.pdf

[15] J. Brendel, R. Fiedler, F. Günther, C. Janson, and D. Stebila, "Post-quantum asynchronous deniable key exchange and the signal handshake," in Public-key cryptography - PKC 2022 - 25th IACR international conference on practice and theory of public-key cryptography, virtual event, march 8-11, 2022, proceedings, part II, 2022, vol. 13178. https://doi.org/10.1007/978-3-030-97131-1_1

[16] N. Vatandas, R. Gennaro, B. Ithurburn, and H. Krawczyk, "On the cryptographic deniability of the signal protocol," in Applied cryptography and network security - 18th international conference, ACNS 2020, rome, italy, october 19-22, 2020, proceedings, part II, 2020, vol. 12147. https://doi.org/10.1007/978-3-030-57878-7_10

[17] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the fujisaki-okamoto transformation," in Theory of cryptography - 15th international conference, TCC 2017, baltimore, MD, USA, november 12-15, 2017, proceedings, part I, 2017, vol. 10677. https://doi.org/10.1007/978-3-319-70500-2_12

[18] K. Hashimoto, S. Katsumata, K. Kwiatkowski, and T. Prest, “An efficient and generic construction for signal’s handshake (X3DH): Post-quantum, state leakage secure, and deniable,” J. Cryptol., vol. 35, no. 3, 2022. https://doi.org/10.1007/s00145-022-09427-1

[19] NIST, “Submission requirements and evaluation criteria for the post-quantum cryptography standardization process,” 2016. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[20] NIST, “Post-quantum cryptography.” https://csrc.nist.gov/Projects/post-quantum-cryptography

[21] “Kyber key encapsulation mechanism.” https://pq-crystals.org/kyber/