openpgp
Internet-Draft
Updates: 4880 (if approved)
Intended status: Informational
Expires: 24 November 2024

D. Shaw
Jabberwocky Tech
A. Gallagher, Ed.
PGPKeys.EU

23 May 2024

OpenPGP Replacement Key Signalling Mechanism
draft-gallagher-openpgp-replacementkey-01

Abstract

This document specifies a method in OpenPGP to suggest a replacement for an expired, revoked, or deprecated key.

About This Document

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

The latest revision of this draft can be found at https://andrewgdotcom.gitlab.io/draft-gallagher-openpgp-replacementkey. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-gallagher-openpgp-replacementkey/.

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

Source for this draft and an issue tracker can be found at https://gitlab.com/andrewgdotcom/draft-gallagher-openpgp-replacementkey.

Status of This Memo

This Internet-Draft will expire on 24 November 2024.

Table of Contents

1. Introduction

The OpenPGP message format [crypto-refresh] defines two ways to invalidate a key. One way is that the key may be explicitly revoked via a revocation signature. OpenPGP also supports the concept of key expiration, a date after which the key should not be used. When a key is revoked or expires, very often there is another key that is intended to replace it.

A key owner may also create a new key that is intended to deprecate and replace their existing key, but without revoking or expiring that key. This is useful during the rollout of new key versions and algorithms which may not (yet) enjoy universal support. In such cases, a key owner may prefer that their correspondents use their new key, but if this is not possible for technical reasons they may continue to use the deprecated key, which remains valid even if it is not preferred.

In the past some key owners have created key transition documents, which are signed, human-readable statements stating that a newer key should be preferred by their correspondents. It is desirable that this process be automated through a standardised machine-readable mechanism.

This document is to specify the format of a signature subpacket to be optionally included in a revocation signature or self-signature on a key. This subpacket contains a pointer to a suggested replacement for the key that is signed over. This replacement key may then be automatically retrieved and (if supported and validated) used instead of the original key.

2. Conventions and Definitions

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

3. The Replacement Key Subpacket

The replacement key subpacket is a Signature Subpacket as defined in [crypto-refresh], section 5.2.3.7, and all general signature subpacket considerations from there apply here as well. The value of the signature subpacket type octet for the replacement key subpacket is (insert this later).

A preferred key server subpacket ([crypto-refresh], section 5.2.3.26) MAY be included in the revocation or self-signature to recommend a location and method to fetch the replacement key. Note however that since this subpacket automatically also applies to the current key, it cannot be used to set the replacement key's preferred keyserver to a different value than that of the current key.

The absence of a replacement key subpacket SHOULD NOT be interpreted as meaning that there is no replacement for the current key. The "no replacement" bit SHOULD be used instead (see below).

The replacement key subpacket is only meaningful in a key revocation or self-signature. It SHOULD NOT be present in any other sort of signature.

4. Format of the Replacement Key Subpacket

The format of the replacement key subpacket is 1 octet of subpacket version and 1 octet of class, followed by an optional 1 octet of key version and N octets of fingerprint.

The subpacket version octet MUST be set to 0x01 to indicate the version of the replacement key subpacket as specified in this document. An implementation that encounters a subpacket version octet that is different than the version(s) it is capable of understanding MUST disregard that replacement key subpacket. Note that if the critical bit for the replacement key subpacket is set, this MAY also mean considering the whole signature to be in error ([crypto-refresh], section 5.2.3.7).

The 0x80 bit of the class octet is the "no replacement" bit. When set, this explicitly specifies there is no replacement for the current key. All other bits of the class octet are currently undefined and MUST be set to zero.

If the class octet does not have the 0x80 bit set to indicate there is no replacement, the replacement key subpacket also contains 1 octet for the key version of the replacement key and N octets for the fingerprint of the replacement key. If present, the length of the fingerprint field MUST equal the fingerprint length corresponding to the key version field, e.g. 20 octets for version 4, or 32 octets for version 6.

If the intent is to state that the replacement key is unknown, then no replacement key subpacket should be included in the revocation signature.

If multiple replacement key subpackets are present, implementations MAY use any method desired to resolve which key (or keys) are the chosen replacement.

5. Trust and Validation of the Replacement Key Subpacket

The existence of a Replacement Key subpacket MUST NOT be considered in any trust calculation over either the current or replacement key. Receiving implementations SHOULD validate the replacement key as they would any other key. If the replacement key is supported, and validates successfully, it SHOULD be preferred over the current key when determining which key to use for correspondence.

Since a Replacement Key subpacket only contains a fingerprint and not a full key, the signature made over it forms a weaker binding than a Certification Signature. A key owner SHOULD therefore also make a Certification Signature over the replacement key using their existing key. It is also suggested that the key owner asks the third parties who certified their current key to certify the replacement key. Distribution of the replacement key over a trusted mechanism (such as WKD) MAY also be used to confer legitimacy.

6. Placement of the Replacement Key Subpacket

While nothing prevents using the replacement key subpacket on a subkey revocation or self-signature, it is mainly useful on a primary key revocation or self-signature as a replacement subkey can be directly added by the keyholder with no need for the indirection provided by this subpacket. The replacement key subpacket SHOULD be placed in the hashed section of the signature to prevent a possible key substitution attack. If the replacement key subpacket was allowed in the unhashed section of the signature, an attacker could add a bogus replacement key subpacket to an existing revocation or self-signature.

7. Security Considerations

The replacement key subpacket provides non-sensitive information only. Nevertheless, as noted above, implementations SHOULD NOT trust a replacement key subpacket that is located in the unhashed area of the signature packet, and SHOULD validate the replacement key as they would any other key. In addition, as this document is an update of [crypto-refresh], the security considerations there should be carefully reviewed.

8. IANA Considerations

This document requests that the following entry be added to the OpenPGP Signature Subpacket registry:

Type	Name	Specification
TBC	Replacement Key	This document

Table 1: Signature Subpacket Registry

9. Normative References

[crypto-refresh]
Wouters, P., Huigens, D., Winter, J., and N. Yutaka, "OpenPGP", October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-crypto-refresh-13>.

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

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

Appendix A. Example Workflows

A.1. Alice Revokes her Key

* Bob wants to send Alice a message, but they have not corresponded for some time.

* Bob's client refreshes Alices key from a keyserver (by fingerprint); it contains a revocation signature with a replacement key subpacket.

* Bob's client looks up Alices new key from a keyserver (by fingerprint); it is certified by the same people that certified her old key (some of whom Bob may trust) and/or Alices old key itself (which Bob's policy may consider sufficient).

* Bob's client uses Alices new key instead of the old key.

There are other means to achieve a similar result, such as WKD or Autocrypt, but they may not be available. For example, Alices service provider may not support WKD, and Alice may not have sent Bob an autocrypt message since revoking her old key.

A.2. Alice Creates a V6 Key

* Bob wants to send Alice a message and has Alice's v4 key.

* Either Bob's copy of Alice's key already has the replacement key subpacket pointing to a v6 key, or Bob refreshes Alice's key from a keyserver and sees a new replacement key subpacket.

* If Bob has a v6 implementation, it can proceed with fetching Alice's v6 key, validating it, etc, and then use it to send his message to Alice.

* If Bob doesn't have a v6 implementation, it can continue to use Alice's v4 key.

WKD does not currently allow more than one valid key to be returned for a query, therefore it cannot easily support this use case.

Appendix B. Acknowledgments

The authors would like to thank Daniel Kahn Gillmor, Simon Josefsson, Heiko Schäfer, Falko Strenzke and Aron Wussler for suggestions and discussions.

Appendix C. Document History

Note to RFC Editor: this section should be removed before publication.

C.1. Changes Between -00 and -01

* Added example workflows.

* Specifically describe "deprecation without expiry or revocation" use case.

* Add note about weakness of signatures over fingerprints.

* Miscellaneous clarifications.

C.2. Changes Between draft-shaw-openpgp-replacementkey-00 and draft-gallagher-openpgp-replacementkey-00

* Changed algid octet to key version octet.

* Changed initial subpacket version number to 1.

* Clarified semantics of some edge cases.

Authors' Addresses

Daphne Shaw
Jabberwocky Tech
Email: dshaw@jabberwocky.com

Andrew Gallagher (editor)
PGPKeys.EU
Email: andrewg@andrewg.com

Network Working Group
Internet-Draft
Updates: 4880 (if approved)
Intended status: Standards Track
Expires: 6 July 2024

D. Huigens, Ed.
Proton AG

3 January 2024

Persistent Symmetric Keys in OpenPGP
draft-huigens-openpgp-persistent-symmetric-keys-02

Abstract

This document defines new algorithms for the OpenPGP standard (RFC4880) to support persistent symmetric keys, for message encryption using authenticated encryption with additional data (AEAD) and for authentication with hash-based message authentication codes (HMAC). This enables the use of symmetric cryptography for data storage (and other contexts that do not require asymmetric cryptography), for improved performance, smaller keys, and improved resistance to quantum computing.

About This Document

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

The latest revision of this draft can be found at https://twisstle.gitlab.io/openpgp-persistent-symmetric-keys/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-huigens-openpgp-persistent-symmetric-keys/.

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

Source for this draft and an issue tracker can be found at https://gitlab.com/twisstle/openpgp-persistent-symmetric-keys.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The OpenPGP standard [RFC4880] has supported symmetric encryption for data packets using session keys since its inception, as well as symmetric encryption using password-derived keys. This document extends the use of symmetric cryptography by adding support for persistent symmetric keys which can be stored in a transferable private key, and used to symmetrically encrypt session keys, for long-term storage and archival of messages. This document uses authenticated encryption with associated data (AEAD) as proposed by the OpenPGP crypto refresh [crypto-refresh].

The OpenPGP standard also supports the use of digital signatures for authentication and integrity but no similar symmetric mechanism exists in the standard. This document introduces hash-based message authentication codes (HMAC) as a symmetric counterpart to digital signatures, for long-term storage and archival of attestations of authenticity and certification.

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119]. Any implementation that adheres to the format and methods specified in this document is called a compliant application. Compliant applications are a subset of the broader set of OpenPGP applications described in [RFC4880] and the OpenPGP crypto refresh [crypto-refresh]. Any [RFC2119] keyword within this document applies to compliant applications only.

3. Motivation

When compared to asymmetric cryptography, symmetric cryptography can provide improved performance and equivalent security with smaller keys. In contexts that do not require asymmetric cryptography, such as secure data storage where the same user encrypts and decrypts data, symmetric cryptography can be used to take advantage of these benefits.

Additionally, asymmetric algorithms included in OpenPGP are vulnerable to attacks that might become possible on quantum computers [Shor]. Symmetric cryptography is also affected by quantum computing but to a lesser extent, which can be countered by using larger keys [Grover]. While the standardization of quantum-secure asymmetric cryptography in OpenPGP is ongoing [PQCinOpenPGP], and will be required to secure communications, there is a large body of existing messages encrypted with classical algorithms. Once persistent

symmetric keys are available, these messages can be protected against future compromises efficiently by symmetrically re-encrypting the session key, and storing the message symmetrically encrypted for long-term storage and archival.

4. Reusing and Renaming Packets

Rather than introducing new packets for storing persistent symmetric keys, the existing Secret-Key packets are reused for this purpose. To indicate the type of keys, two algorithms (AEAD and HMAC) are registered, whose IDs can be used in the place of public-key algorithm IDs. To accommodate these additions, we propose renaming the Public Key Algorithms registry to Persistent Key Algorithms.

Similarly, we reuse the Signature packet for "symmetric signatures". For session keys encrypted with persistent symmetric keys, while a Symmetric-Key Encrypted Session Key packet exists, its semantics don't match our requirements, as it's intended to encrypt the session key with a user-provided password, and doesn't offer a way to store a reference to a persistent key. Therefore, we reuse the Public-Key Encrypted Session Key packet instead, which does offer the desired semantics. Nevertheless, given this usage, the naming of these packets may be confusing, so we propose to rename them to "String-to-Key Encrypted Session Key packet" and "Persistent Key Encrypted Session Key packet", instead.

5. Persistent Symmetric Key Algorithms

This document defines two new algorithms for use with OpenPGP, extending the table in section 9.1 of [crypto-refresh].

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
64	AEAD	sym. algo, seed hash [Section 5.1]	hash seed, key material	N/A	AEAD algo, IV, length, ciphertext [Section 5.3]
65	HMAC [RFC2104]	hash algo, seed hash [Section 5.2]	hash seed, key material	authentication tag [Section 5.4]	N/A

Table 1: Persistent Symmetric Key Algorithm registrations

These algorithm IDs can be used to store symmetric key material in Secret-Key Packets and Secret-Subkey packets (see section 5.5.3 of [crypto-refresh]). The AEAD algorithm ID can be used to store session keys encrypted using AEAD in PKESK packets (see section 5.1 of [crypto-refresh]). The HMAC algorithm ID can be used to store HMAC-based signatures in Signature packets (see section 5.2 of [crypto-refresh]).

As the secret key material is required for all cryptographic operations with symmetric keys, implementations SHOULD NOT use these algorithm IDs in Public-Key Packets or Public-Subkey Packets, and SHOULD NOT export Public-Key Packets from Secret-Key Packets holding symmetric key material.

5.1. Algorithm-Specific Fields for AEAD keys

The public key is this series of values:

* A one-octet symmetric algorithm identifier (see section 9.3 of [crypto-refresh])

* A 32-octet SHA-256 hash of the seed in the private key material

The private key is this series of values:

* A 32-octet seed value to be hashed for the public key material

* Symmetric key material of appropriate length for the chosen symmetric algorithm

5.2. Algorithm-Specific Fields for HMAC keys

The public key is this series of values:

* A one-octet hash algorithm identifier (see section 9.5 of [crypto-refresh])

* A 32-octet SHA-256 hash of the seed in the private key material

The private key is this series of values:

* A 32-octet seed value to be hashed for the public key material

* Symmetric key material of the length of the hash output size of the chosen hash algorithm

5.3. Algorithm-Specific Fields for AEAD encryption

* A one-octet AEAD algorithm (see section 9.6 of [crypto-refresh])

* A starting initialization vector of size specified by AEAD mode

* A one-octet length of the following field

* A symmetric key encryption of the plaintext value described in section 5.1 of [crypto-refresh], performed using the selected symmetric-key cipher operating in the given AEAD mode, including the authentication tag.

5.4. Algorithm-Specific Fields for HMAC signatures

* An authentication tag of appropriate length for the hash algorithm

Although not required by HMAC, to maintain compatibility with existing signature implementations, HMAC tags are produced from appropriately hashed data, as per section 5.2.4 of [crypto-refresh].

6. Security Considerations

Security considerations are discussed throughout the document where appropriate.

7. IANA Considerations

7.1. Updates to Public Key Algorithms

IANA is requested to rename the "Public Key Algorithms" registry to "Persistent Key Algorithms", and add the entries in Table 1 to the registry.

7.2. Updates to Packet Type Descriptions

IANA is requested to modify the "PGP Packet Types/Tags" registry as follows:

* For Packet Tag 1 ("Public-Key Encrypted Session Key Packet"), change the Packet Type to "Persistent Key Encrypted Session Key Packet".

* For Packet Tag 3 ("Symmetric-Key Encrypted Session Key Packet"), change the Packet Type to "String-to-Key Encrypted Session Key Packet".

8. Acknowledgements

An initial version of this draft was written by Dan Ristea (Proton AG), with guidance from Dr Philipp Jovanovic (University College London).

9. References

9.1. Normative References

[crypto-refresh] Wouters, P., Huigens, D., Winter, J., and N. Yutaka, "OpenPGP", October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-crypto-refresh-12>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

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

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <https://www.rfc-editor.org/info/rfc4880>.

9.2. Informative References

[Grover] Grover, L., "Quantum mechanics helps in searching for a needle in a haystack", 1997, <https://arxiv.org/abs/quant-ph/9706033>.

[PQCinOpenPGP] Kousidis, S., Strenzke, F., and A. Wussler, "Post-Quantum Cryptography in OpenPGP", October 2023, <https://datatracker.ietf.org/doc/html/draft-wussler-openpgp-pqc-03>.

[Shor] Shor, P., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", October 1997, <http://dx.doi.org/10.1137/S0097539795293172>.

Author's Address

Daniel Huigens (editor)
Proton AG
Route de la Galaise 32
CH-1228 Plan-les-Ouates
Switzerland
Email: d.huigens@protonmail.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 5 September 2024

S. Kousidis
BSI
J. Roth
F. Strenzke
MTG AG
A. Wussler
Proton AG

4 March 2024

Post-Quantum Cryptography in OpenPGP
draft-ietf-openpgp-pqc-02

Abstract

This document defines a post-quantum public-key algorithm extension for the OpenPGP protocol. Given the generally assumed threat of a cryptographically relevant quantum computer, this extension provides a basis for long-term secure OpenPGP signatures and ciphertexts. Specifically, it defines composite public-key encryption based on ML-KEM (formerly CRYSTALS-Kyber), composite public-key signatures based on ML-DSA (formerly CRYSTALS-Dilithium), both in combination with elliptic curve cryptography, and SLH-DSA (formerly SPHINCS+) as a standalone public key signature scheme.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-openpgp-pqc/.

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

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

Status of This Memo

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

This Internet-Draft will expire on 5 September 2024.

Table of Contents

1. Introduction

The OpenPGP protocol supports various traditional public-key algorithms based on the factoring or discrete logarithm problem. As the security of algorithms based on these mathematical problems is endangered by the advent of quantum computers, there is a need to extend OpenPGP by algorithms that remain secure in the presence of quantum computers.

Such cryptographic algorithms are referred to as post-quantum cryptography. The algorithms defined in this extension were chosen for standardization by the National Institute of Standards and Technology (NIST) in mid 2022 [NISTIR-8413] as the result of the NIST Post-Quantum Cryptography Standardization process initiated in 2016 [NIST-PQC]. Namely, these are ML-KEM [FIPS-203] as a Key Encapsulation Mechanism (KEM), a KEM being a modern building block for public-key encryption, and ML-DSA [FIPS-204] as well as SLH-DSA [FIPS-205] as signature schemes.

For the two ML-* schemes, this document follows the conservative strategy to deploy post-quantum in combination with traditional schemes such that the security is retained even if all schemes but one in the combination are broken. In contrast, the stateless hash-based signature scheme SLH-DSA is considered to be sufficiently well understood with respect to its security assumptions in order to be used standalone. To this end, this document specifies the following new set: SLH-DSA standalone and the two ML-* as composite with ECC-based KEM and digital signature schemes. Here, the term "composite" indicates that any data structure or algorithm pertaining to the combination of the two components appears as single data structure or algorithm from the protocol perspective.

The document specifies the conventions for interoperability between compliant OpenPGP implementations that make use of this extension and the newly defined algorithms or algorithm combinations.

1.1. Conventions used in this Document

1.1.1. Terminology for Multi-Algorithm Schemes

The terminology in this document is oriented towards the definitions in [draft-driscoll-pqt-hybrid-terminology]. Specifically, the terms "multi-algorithm", "composite" and "non-composite" are used in correspondence with the definitions therein. The abbreviation "PQ" is used for post-quantum schemes. To denote the combination of post-quantum and traditional schemes, the abbreviation "PQ/T" is used. The short form "PQ(/T)" stands for PQ or PQ/T.

1.2. Post-Quantum Cryptography

This section describes the individual post-quantum cryptographic schemes. All schemes listed here are believed to provide security in the presence of a cryptographically relevant quantum computer. However, the mathematical problems on which the two ML-* schemes and SLH-DSA are based, are fundamentally different, and accordingly the level of trust commonly placed in them as well as their performance characteristics vary.

[Note to the reader: This specification refers to the NIST PQC draft standards FIPS 203, FIPS 204, and FIPS 205 as if they were a final specification. This is a temporary solution until the final versions of these documents are available. The goal is to provide a sufficiently precise specification of the algorithms already at the draft stage of this specification, so that it is possible for implementers to create interoperable implementations. Furthermore, we want to point out that, depending on possible future changes to the draft standards by NIST, this specification may be updated as soon as corresponding information becomes available.]

1.2.1. ML-KEM

ML-KEM [FIPS-203] is based on the hardness of solving the learning-with-errors problem in module lattices (MLWE). The scheme is believed to provide security against cryptanalytic attacks by classical as well as quantum computers. This specification defines ML-KEM only in composite combination with ECC-based encryption schemes in order to provide a pre-quantum security fallback.

1.2.2. ML-DSA

ML-DSA [FIPS-204] is a signature scheme that, like ML-KEM, is based on the hardness of solving the Learning With Errors problem and a variant of the Short Integer Solution problem in module lattices (MLWE and SelfTargetMSIS). Accordingly, this specification only defines ML-DSA in composite combination with ECC-based signature schemes.

1.2.3. SLH-DSA

SLH-DSA [FIPS-205] is a stateless hash-based signature scheme. Its security relies on the hardness of finding preimages for cryptographic hash functions. This feature is generally considered to be a high security guarantee. Therefore, this specification defines SLH-DSA as a standalone signature scheme.

In deployments the performance characteristics of SLH-DSA should be taken into account. We refer to Section 11.1 for a discussion of the performance characteristics of this scheme.

1.3. Elliptic Curve Cryptography

The ECC-based encryption is defined here as a KEM. This is in contrast to [I-D.ietf-openpgp-crypto-refresh] where the ECC-based encryption is defined as a public-key encryption scheme.

All elliptic curves for the use in the composite combinations are taken from [I-D.ietf-openpgp-crypto-refresh]. However, as explained in the following, in the case of Curve25519 encoding changes are applied to the new composite schemes.

1.3.1. Curve25519 and Curve448

Curve25519 and Curve448 are defined in [RFC7748] for use in a Diffie-Hellman key agreement scheme and defined in [RFC8032] for use in a digital signature scheme. For Curve25519 this specification adopts the encoding of objects as defined in [RFC7748].

1.3.2. Generic Prime Curves

For interoperability this extension offers CRYSTALS-* in composite combinations with the NIST curves P-256, P-384 defined in [SP800-186] and the Brainpool curves brainpoolP256r1, brainpoolP384r1 defined in [RFC5639].

1.4. Standalone and Multi-Algorithm Schemes

This section provides a categorization of the new algorithms and their combinations.

1.4.1. Standalone and Composite Multi-Algorithm Schemes

This specification introduces new cryptographic schemes, which can be categorized as follows:

* PQ/T multi-algorithm public-key encryption, namely a composite combination of ML-KEM with an ECC-based KEM,

* PQ/T multi-algorithm digital signature, namely composite combinations of ML-DSA with ECC-based signature schemes,

* PQ digital signature, namely SLH-DSA as a standalone cryptographic algorithm.

For each of the composite schemes, this specification mandates that the recipient has to successfully perform the cryptographic algorithms for each of the component schemes used in a cryptographic message, in order for the message to be deciphered and considered as valid. This means that all component signatures must be verified successfully in order to achieve a successful verification of the composite signature. In the case of the composite public-key decryption, each of the component KEM decapsulation operations must succeed.

1.4.2. Non-Composite Algorithm Combinations

As the OpenPGP protocol [I-D.ietf-openpgp-crypto-refresh] allows for multiple signatures to be applied to a single message, it is also possible to realize non-composite combinations of signatures. Furthermore, multiple OpenPGP signatures may be combined on the application layer. These latter two cases realize non-composite combinations of signatures. Section 4.4 specifies how implementations should handle the verification of such combinations of signatures.

Furthermore, the OpenPGP protocol also allows for parallel encryption to different keys held by the same recipient. Accordingly, if the sender makes use of this feature and sends an encrypted message with multiple PKESK packages for different encryption keys held by the same recipient, a non-composite multi-algorithm public-key encryption is realized where the recipient has to decrypt only one of the PKESK packages in order to decrypt the message. See Section 4.2 for restrictions on parallel encryption mandated by this specification.

2. Preliminaries

This section provides some preliminaries for the definitions in the subsequent sections.

2.1. Elliptic curves

2.1.1. SEC1 EC Point Wire Format

Elliptic curve points of the generic prime curves are encoded using the SEC1 (uncompressed) format as the following octet string:

B = 04 || X || Y

where X and Y are coordinates of the elliptic curve point P = (X, Y), and each coordinate is encoded in the big-endian format and zero-padded to the adjusted underlying field size. The adjusted underlying field size is the underlying field size rounded up to the nearest 8-bit boundary, as noted in the "Field size" column in Table 6, Table 7, or Table 11. This encoding is compatible with the definition given in [SEC1].

2.1.2. Measures to Ensure Secure Implementations

In the following measures are described that ensure secure implementations according to existing best practices and standards defining the operations of Elliptic Curve Cryptography.

Even though the zero point, also called the point at infinity, may occur as a result of arithmetic operations on points of an elliptic curve, it MUST NOT appear in any ECC data structure defined in this document.

Furthermore, when performing the explicitly listed operations in Section 5.1.1.1, Section 5.1.1.2 or Section 5.1.1.3 it is REQUIRED to follow the specification and security advisory mandated from the respective elliptic curve specification.

3. Supported Public Key Algorithms

This section specifies the composite ML-KEM + ECC and ML-DSA + ECC schemes as well as the standalone SLH-DSA signature scheme. The composite schemes are fully specified via their algorithm ID. The SLH-DSA signature schemes are fully specified by their algorithm ID and an additional parameter ID.

3.1. Algorithm Specifications

For encryption, the following composite KEM schemes are specified:

ID	Algorithm	Requirement	Definition
TBD (105 for testing)	ML-KEM-768 + X25519	MUST	Section 5.2
TBD (106 for testing)	ML-KEM-1024 + X448	SHOULD	Section 5.2
TBD	ML-KEM-768 + ECDH-NIST-P-256	MAY	Section 5.2
TBD	ML-KEM-1024 + ECDH-NIST-P-384	MAY	Section 5.2
TBD	ML-KEM-768 + ECDH-brainpoolP256r1	MAY	Section 5.2
TBD	ML-KEM-1024 + ECDH-brainpoolP384r1	MAY	Section 5.2

Table 1: KEM algorithm specifications

For signatures, the following (composite) signature schemes are specified:

ID	Algorithm	Requirement	Definition
TBD (107 for testing)	ML-DSA-65 + Ed25519	MUST	Section 6.2
TBD (108 for testing)	ML-DSA-87 + Ed448	SHOULD	Section 6.2
TBD	ML-DSA-65 + ECDSA-NIST-P-256	MAY	Section 6.2
TBD	ML-DSA-87 + ECDSA-NIST-P-384	MAY	Section 6.2
TBD	ML-DSA-65 + ECDSA-brainpoolP256r1	MAY	Section 6.2
TBD	ML-DSA-87 + ECDSA-brainpoolP384r1	MAY	Section 6.2
TBD (109 for testing)	SLH-DSA-SHA2	SHOULD	Section 7.1
TBD	SLH-DSA-SHAKE	MAY	Section 7.1

Table 2: Signature algorithm specifications

3.1.1. Experimental Codepoints for Interop Testing

[Note: this section to be removed before publication]

Algorithms indicated as MAY are not assigned a codepoint in the current state of the draft since there are not enough private/experimental code points available to cover all newly introduced public-key algorithm identifiers.

The use of private/experimental codepoints during development are intended to be used in non-released software only, for experimentation and interop testing purposes only. An OpenPGP implementation MUST NOT produce a formal release using these experimental codepoints. This draft will not be sent to IANA without every listed algorithm having a non-experimental codepoint.

3.2. Parameter Specification

3.2.1. SLH-DSA-SHA2

For the SLH-DSA-SHA2 signature algorithm from Table 2, the following parameters are specified:

Parameter ID	Parameter
1	SLH-DSA-SHA2-128s
2	SLH-DSA-SHA2-128f
3	SLH-DSA-SHA2-192s
4	SLH-DSA-SHA2-192f
5	SLH-DSA-SHA2-256s
6	SLH-DSA-SHA2-256f

Table 3: SLH-DSA-SHA2 security parameters

All security parameters inherit the requirement of SLH-DSA-SHA2 from Table 2. That is, implementations SHOULD implement the parameters specified in Table 3. The values 0x00 and 0xFF are reserved for future extensions.

3.2.2. SLH-DSA-SHAKE

For the SLH-DSA-SHAKE signature algorithm from Table 2, the following parameters are specified:

Parameter ID	Parameter
1	SLH-DSA-SHAKE-128s
2	SLH-DSA-SHAKE-128f
3	SLH-DSA-SHAKE-192s
4	SLH-DSA-SHAKE-192f
5	SLH-DSA-SHAKE-256s
6	SLH-DSA-SHAKE-256f

Table 4: SLH-DSA-SHAKE security parameters

All security parameters inherit the requirement of SLH-DSA-SHAKE from Table 2. That is, implementations MAY implement the parameters specified in Table 4. The values 0x00 and 0xFF are reserved for future extensions.

4. Algorithm Combinations

4.1. Composite KEMs

The ML-KEM + ECC public-key encryption involves both the ML-KEM and an ECC-based KEM in an a priori non-separable manner. This is achieved via KEM combination, i.e. both key encapsulations/ decapsulations are performed in parallel, and the resulting key shares are fed into a key combiner to produce a single shared secret for message encryption.

4.2. Parallel Public-Key Encryption

As explained in Section 1.4.2, the OpenPGP protocol inherently supports parallel encryption to different keys of the same recipient. Implementations MUST NOT encrypt a message with a purely traditional public-key encryption key of a recipient if it is encrypted with a PQ/T key of the same recipient.

4.3. Composite Signatures

The ML-DSA + ECC signature consists of independent ML-DSA and ECC signatures, and an implementation MUST successfully validate both signatures to state that the ML-DSA + ECC signature is valid.

4.4. Multiple Signatures

The OpenPGP message format allows multiple signatures of a message, i.e. the attachment of multiple signature packets.

An implementation MAY sign a message with a traditional key and a PQ(/T) key from the same sender. This ensures backwards compatibility due to [I-D.ietf-openpgp-crypto-refresh] Section 5.2.5, since a legacy implementation without PQ(/T) support can fall back on the traditional signature.

Newer implementations with PQ(/T) support MAY ignore the traditional signature(s) during validation.

Implementations SHOULD consider the message correctly signed if at least one of the non-ignored signatures validates successfully.

[Note to the reader: The last requirement, that one valid signature is sufficient to identify a message as correctly signed, is an interpretation of [I-D.ietf-openpgp-crypto-refresh] Section 5.2.5.]

5. Composite KEM schemes

5.1. Building Blocks

5.1.1. ECC-Based KEMs

In this section we define the encryption, decryption, and data formats for the ECDH component of the composite algorithms.

Table 5, Table 6, and Table 7 describe the ECC-KEM parameters and artifact lengths. The artifacts in Table 5 follow the encodings described in [RFC7748].

	X25519	X448
Algorithm ID reference	TBD (105 for testing)	TBD (106 for testing)
Field size	32 octets	56 octets
ECC-KEM	x25519Kem (Section 5.1.1.1)	x448Kem (Section 5.1.1.2)
ECDH public key	32 octets [RFC7748]	56 octets [RFC7748]
ECDH secret key	32 octets [RFC7748]	56 octets [RFC7748]
ECDH ephemeral	32 octets [RFC7748]	56 octets [RFC7748]
ECDH share	32 octets [RFC7748]	56 octets [RFC7748]
Key share	32 octets	64 octets
Hash	SHA3-256	SHA3-512

Table 5: Montgomery curves parameters and artifact lengths

	NIST P-256	NIST P-384
Algorithm ID reference	TBD (ML-KEM-768 + ECDH-NIST-P-256)	TBD (ML-KEM-1024 + ECDH-NIST-P-384)
Field size	32 octets	48 octets
ECC-KEM	ecdhKem (Section 5.1.1.3)	ecdhKem (Section 5.1.1.3)
ECDH public key	65 octets of SEC1-encoded public point	97 octets of SEC1-encoded public point
ECDH secret key	32 octets big-endian encoded secret scalar	48 octets big-endian encoded secret scalar
ECDH ephemeral	65 octets of SEC1-encoded ephemeral point	97 octets of SEC1-encoded ephemeral point
ECDH share	65 octets of SEC1-encoded shared point	97 octets of SEC1-encoded shared point
Key share	32 octets	64 octets
Hash	SHA3-256	SHA3-512

Table 6: NIST curves parameters and artifact lengths

	brainpoolP256r1	brainpoolP384r1
Algorithm ID reference	TBD (ML-KEM-768 + ECDH-brainpoolP256r1)	TBD (ML-KEM-1024 + ECDH-brainpoolP384r1)
Field size	32 octets	48 octets
ECC-KEM	ecdhKem (Section 5.1.1.3)	ecdhKem (Section 5.1.1.3)
ECDH public key	65 octets of SEC1-encoded public point	97 octets of SEC1-encoded public point
ECDH secret key	32 octets big-endian encoded secret scalar	48 octets big-endian encoded secret scalar
ECDH ephemeral	65 octets of SEC1-encoded ephemeral point	97 octets of SEC1-encoded ephemeral point
ECDH share	65 octets of SEC1-encoded shared point	97 octets of SEC1-encoded shared point
Key share	32 octets	64 octets
Hash	SHA3-256	SHA3-512

Table 7: Brainpool curves parameters and artifact lengths

The SEC1 format for point encoding is defined in Section 2.1.1.

The various procedures to perform the operations of an ECC-based KEM are defined in the following subsections. Specifically, each of these subsections defines the instances of the following operations:

(eccCipherText, eccKeyShare) <- ECC-KEM.Encaps(eccPublicKey)

and

(eccKeyShare) <- ECC-KEM.Decaps(eccSecretKey, eccCipherText, eccPublicKey)

To instantiate ECC-KEM, one must select a parameter set from Table 5, Table 6, or Table 7.

5.1.1.1. X25519-KEM

The encapsulation and decapsulation operations of x25519kem are described using the function X25519() and encodings defined in [RFC7748]. The eccSecretKey is denoted as r, the eccPublicKey as R, they are subject to the equation R = X25519(r, U(P)). Here, U(P) denotes the u-coordinate of the base point of Curve25519.

The operation x25519Kem.Encaps() is defined as follows:

1. Generate an ephemeral key pair {v, V} via V = X25519(v,U(P)) where v is a randomly generated octet string with a length of 32 octets

2. Compute the shared coordinate X = X25519(v, R) where R is the recipient's public key eccPublicKey

3. Set the output eccCipherText to V

4. Set the output eccKeyShare to SHA3-256(X || eccCipherText || eccPublicKey)

The operation x25519Kem.Decaps() is defined as follows:

1. Compute the shared coordinate X = X25519(r, V), where r is the eccSecretKey and V is the eccCipherText

2. Set the output eccKeyShare to SHA3-256(X || eccCipherText || eccPublicKey)

5.1.1.2. X448-KEM

The encapsulation and decapsulation operations of x448kem are described using the function X448() and encodings defined in [RFC7748]. The eccSecretKey is denoted as r, the eccPublicKey as R, they are subject to the equation R = X25519(r, U(P)). Here, U(P) denotes the u-coordinate of the base point of Curve448.

The operation x448.Encaps() is defined as follows:

1. Generate an ephemeral key pair {v, V} via V = X448(v,U(P)) where v is a randomly generated octet string with a length of 56 octets

2. Compute the shared coordinate X = X448(v, R) where R is the recipient's public key eccPublicKey

3. Set the output eccCipherText to V

4. Set the output eccKeyShare to SHA3-512(X || eccCipherText || eccPublicKey)

The operation x448Kem.Decaps() is defined as follows:

1. Compute the shared coordinate X = X448(r, V), where r is the eccSecretKey and V is the eccCipherText

2. Set the output eccKeyShare to SHA3-512(X || eccCipherText || eccPublicKey)

5.1.1.3. ECDH-KEM

The operation ecdhKem.Encaps() is defined as follows:

1. Generate an ephemeral key pair {v, V=vG} as defined in [SP800-186] or [RFC5639] where v is a random scalar with 0 < v < n, n being the base point order of the elliptic curve domain parameters

2. Compute the shared point S = vR, where R is the component public key eccPublicKey, according to [SP800-186] or [RFC5639]

3. Extract the X coordinate from the SEC1 encoded point S = 04 || X || Y as defined in section Section 2.1.1

4. Set the output eccCipherText to the SEC1 encoding of V

5. Set the output eccKeyShare to Hash(X || eccCipherText || eccPublicKey), with Hash chosen according to Table 6 or Table 7

The operation ecdhKem.Decaps() is defined as follows:

1. Compute the shared Point S as rV, where r is the eccSecretKey and V is the eccCipherText, according to [SP800-186] or [RFC5639]

2. Extract the X coordinate from the SEC1 encoded point S = 04 || X || Y as defined in section Section 2.1.1

3. Set the output eccKeyShare to Hash(X || eccCipherText || eccPublicKey), with Hash chosen according to Table 6 or Table 7

5.1.2. ML-KEM

ML-KEM features the following operations:

(mlkemCipherText, mlkemKeyShare) <- ML-KEM.Encaps(mlkemPublicKey)

and

(mlkemKeyShare) <- ML-KEM.Decaps(mlkemCipherText, mlkemSecretKey)

The above are the operations ML-KEM.Encaps and ML-KEM.Decaps defined in [FIPS-203]. Note that mlkemPublicKey is the encapsulation and mlkemSecretKey is the decapsulation key.

ML-KEM has the parametrization with the corresponding artifact lengths in octets as given in Table 8. All artifacts are encoded as defined in [FIPS-203].

Algorithm ID reference	ML-KEM	Public key	Secret key	Ciphertext	Key share
TBD	ML-KEM-768	1184	2400	1088	32
TBD	ML-KEM-1024	1568	3168	1568	32

Table 8: ML-KEM parameters artifact lengths in octets

To instantiate ML-KEM, one must select a parameter set from the column "ML-KEM" of Table 8.

The procedure to perform ML-KEM.Encaps() is as follows:

1. Invoke (mlkemCipherText, mlkemKeyShare) <- ML-KEM.Encaps(mlkemPublicKey), where mlkemPublicKey is the recipient's public key

2. Set mlkemCipherText as the ML-KEM ciphertext

3. Set mlkemKeyShare as the ML-KEM symmetric key share

The procedure to perform ML-KEM.Decaps() is as follows:

1. Invoke mlkemKeyShare <- ML-KEM.Decaps(mlkemCipherText, mlkemSecretKey)

2. Set mlkemKeyShare as the ML-KEM symmetric key share

5.2. Composite Encryption Schemes with ML-KEM

Table 1 specifies the following ML-KEM + ECC composite public-key encryption schemes:

Algorithm ID reference	ML-KEM	ECC-KEM	ECC-KEM curve
TBD (105 for testing)	ML-KEM-768	x25519Kem	Curve25519
TBD (106 for testing)	ML-KEM-1024	x448Kem	Curve448
TBD (ML-KEM-768 + ECDH-NIST-P-256)	ML-KEM-768	ecdhKem	NIST P-256
TBD (ML-KEM-1024 + ECDH-NIST-P-384)	ML-KEM-1024	ecdhKem	NIST P-384
TBD (ML-KEM-768 + ECDH-brainpoolP256r1)	ML-KEM-768	ecdhKem	brainpoolP256r1
TBD (ML-KEM-1024 + ECDH-brainpoolP384r1)	ML-KEM-1024	ecdhKem	brainpoolP384r1

Table 9: ML-KEM + ECC composite schemes

The ML-KEM + ECC composite public-key encryption schemes are built according to the following principal design:

* The ML-KEM encapsulation algorithm is invoked to create a ML-KEM ciphertext together with a ML-KEM symmetric key share.

* The encapsulation algorithm of an ECC-based KEM, namely one out of X25519-KEM, X448-KEM, or ECDH-KEM is invoked to create an ECC ciphertext together with an ECC symmetric key share.

* A Key-Encryption-Key (KEK) is computed as the output of a key combiner that receives as input both of the above created symmetric key shares and the protocol binding information.

* The session key for content encryption is then wrapped as described in [RFC3394] using AES-256 as algorithm and the KEK as key.

* The PKESK package's algorithm-specific parts are made up of the ML-KEM ciphertext, the ECC ciphertext, and the wrapped session key.

5.2.1. Fixed information

For the composite KEM schemes defined in Table 1 the following procedure, justified in Section 10.4, MUST be used to derive a string to use as binding between the KEK and the communication parties.

```
//   Input:
//   algID     - the algorithm ID encoded as octet

fixedInfo = algID
```

5.2.2. Key combiner

For the composite KEM schemes defined in Table 1 the following procedure MUST be used to compute the KEK that wraps a session key. The construction is a one-step key derivation function compliant to [SP800-56C] Section 4, based on KMAC256 [SP800-185]. It is given by the following algorithm, which computes the key encryption key KEK that is used to wrap, i.e., encrypt, the session key.

```
//   multiKeyCombine(eccKeyShare, eccCipherText,
//                   mlkemKeyShare, mlkemCipherText,
//                   fixedInfo, oBits)
//
//   Input:
//   eccKeyShare     - the ECC key share encoded as an octet string
//   eccCipherText   - the ECC ciphertext encoded as an octet string
//   mlkemKeyShare   - the ML-KEM key share encoded as an octet string
//   mlkemCipherText - the ML-KEM ciphertext encoded as an octet string
//   fixedInfo       - the fixed information octet string
//   oBits           - the size of the output keying material in bits
//
//   Constants:
//   domSeparation       - the UTF-8 encoding of the string
//                         "OpenPGPCompositeKeyDerivationFunction"
//   counter             - the 4 byte value 00 00 00 01
//   customizationString - the UTF-8 encoding of the string "KDF"

eccData = eccKeyShare || eccCipherText
mlkemData = mlkemKeyShare || mlkemCipherText
encData = counter || eccData || mlkemData || fixedInfo

KEK = KMAC256(domSeparation, encData, oBits, customizationString)
return KEK
```

Here, the parameters to KMAC256 appear in the order as specified in [SP800-186], Section 4, i.e., the key K, main input data X, requested output length L, and optional customization string S in that order.

Note that the values eccKeyShare defined in Section 5.1.1 and mlkemKeyShare defined in Section 5.1.2 already use the relative ciphertext in the derivation. The ciphertext is by design included again in the key combiner to provide a robust security proof.

The value of domSeparation is the UTF-8 encoding of the string "OpenPGPCompositeKeyDerivationFunction" and MUST be the following octet sequence:

```
domSeparation := 4F 70 65 6E 50 47 50 43 6F 6D 70 6F 73 69 74 65
                 4B 65 79 44 65 72 69 76 61 74 69 6F 6E 46 75 6E
                 63 74 69 6F 6E
```

The value of counter MUST be set to the following octet sequence:

```
counter :=  00 00 00 01
```

The value of fixedInfo MUST be set according to Section 5.2.1.

The value of customizationString is the UTF-8 encoding of the string "KDF" and MUST be set to the following octet sequence:

```
customizationString := 4B 44 46
```

5.2.3. Key generation procedure

The implementation MUST independently generate the ML-KEM and the ECC component keys. ML-KEM key generation follows the specification [FIPS-203] and the artifacts are encoded as fixed-length octet strings as defined in Section 5.1.2. For ECC this is done following the relative specification in [RFC7748], [SP800-186], or [RFC5639], and encoding the outputs as fixed-length octet strings in the format specified in Table 5, Table 6, or Table 7.

5.2.4. Encryption procedure

The procedure to perform public-key encryption with a ML-KEM + ECC composite scheme is as follows:

1. Take the recipient’s authenticated public-key packet pkComposite and sessionKey as input

2. Parse the algorithm ID from pkComposite

3. Extract the eccPublicKey and mlkemPublicKey component from the algorithm specific data encoded in pkComposite with the format specified in Section 5.3.2.

4. Instantiate the ECC-KEM and the ML-KEM depending on the algorithm ID according to Table 9

5. Compute (eccCipherText, eccKeyShare) := ECC-KEM.Encaps(eccPublicKey)

6. Compute (mlkemCipherText, mlkemKeyShare) := ML-KEM.Encaps(mlkemPublicKey)

7. Compute fixedInfo as specified in Section 5.2.1

8. Compute KEK := multiKeyCombine(eccKeyShare, eccCipherText, mlkemKeyShare, mlkemCipherText, fixedInfo, oBits=256) as defined in Section 5.2.2

9. Compute C := AESKeyWrap(KEK, sessionKey) with AES-256 as per [RFC3394] that includes a 64 bit integrity check

10. Output the algorithm specific part of the PKESK as eccCipherText || mlkemCipherText (|| symAlgId) || len(C) || C, where both symAlgId and len(C) are single octet fields and symAlgId denotes the symmetric algorithm ID used and is present only for a v3 PKESK

5.2.5. Decryption procedure

The procedure to perform public-key decryption with a ML-KEM + ECC composite scheme is as follows:

1. Take the matching PKESK and own secret key packet as input

2. From the PKESK extract the algorithm ID and the encryptedKey, i.e., the wrapped session key

3. Check that the own and the extracted algorithm ID match

4. Parse the eccSecretKey and mlkemSecretKey from the algorithm specific data of the own secret key encoded in the format specified in Section 5.3.2

5. Instantiate the ECC-KEM and the ML-KEM depending on the algorithm ID according to Table 9

6. Parse eccCipherText, mlkemCipherText, and C from encryptedKey encoded as eccCipherText || mlkemCipherText (|| symAlgId) || len(C) || C as specified in Section 5.3.1, where symAlgId is present only in the case of a v3 PKESK.

7. Compute (eccKeyShare) := ECC-KEM.Decaps(eccCipherText, eccSecretKey, eccPublicKey)

8. Compute (mlkemKeyShare) := ML-KEM.Decaps(mlkemCipherText, mlkemSecretKey)

9. Compute fixedInfo as specified in Section 5.2.1

10. Compute KEK := multiKeyCombine(eccKeyShare, eccCipherText, mlkemKeyShare, mlkemCipherText, fixedInfo, oBits=256) as defined in Section 5.2.2

11. Compute sessionKey := AESKeyUnwrap(KEK, C) with AES-256 as per [RFC3394], aborting if the 64 bit integrity check fails

12. Output sessionKey

5.3. Packet specifications

5.3.1. Public-Key Encrypted Session Key Packets (Tag 1)

The algorithm-specific fields consists of the output of the encryption procedure described in Section 5.2.4:

* A fixed-length octet string representing an ECC ephemeral public key in the format associated with the curve as specified in Section 5.1.1.

* A fixed-length octet string of the ML-KEM ciphertext, whose length depends on the algorithm ID as specified in Table 8.

* A one-octet size of the following fields.

* Only in the case of a v3 PKESK packet: a one-octet symmetric algorithm identifier.

* The wrapped session key represented as an octet string.

Note that like in the case of the algorithms X25519 and X448 specified in [I-D.ietf-openpgp-crypto-refresh], for the ML-KEM composite schemes, in the case of a v3 PKESK packet, the symmetric algorithm identifier is not encrypted. Instead, it is placed in plaintext after the mlkemCipherText and before the length octet preceding the wrapped session key. In the case of v3 PKESK packets for ML-KEM composite schemes, the symmetric algorithm used MUST be AES-128, AES-192 or AES-256 (algorithm ID 7, 8 or 9).

In the case of a v3 PKESK, a receiving implementation MUST check if the length of the unwrapped symmetric key matches the symmetric algorithm identifier, and abort if this is not the case.

5.3.2. Key Material Packets

The algorithm-specific public key is this series of values:

* A fixed-length octet string representing an EC point public key, in the point format associated with the curve specified in Section 5.1.1.

* A fixed-length octet string containing the ML-KEM public key, whose length depends on the algorithm ID as specified in Table 8.

The algorithm-specific secret key is these two values:

* A fixed-length octet string of the encoded secret scalar, whose encoding and length depend on the algorithm ID as specified in Section 5.1.1.

* A fixed-length octet string containing the ML-KEM secret key, whose length depends on the algorithm ID as specified in Table 8.

6. Composite Signature Schemes

6.1. Building blocks

6.1.1. EdDSA-Based signatures

To sign and verify with EdDSA the following operations are defined:

(eddsaSignature) <- EdDSA.Sign(eddsaSecretKey, dataDigest)

and

(verified) <- EdDSA.Verify(eddsaPublicKey, eddsaSignature, dataDigest)

The public and secret key, as well as the signature MUST be encoded according to [RFC8032] as fixed-length octet strings. The following table describes the EdDSA parameters and artifact lengths:

Algorithm ID reference	Curve	Field size	Public key	Secret key	Signature
TBD (107 for testing)	Ed25519	32	32	32	64
TBD (108 for testing)	Ed448	57	57	57	114

Table 10: EdDSA parameters and artifact lengths in octets

6.1.2. ECDSA-Based signatures

To sign and verify with ECDSA the following operations are defined:

```
(ecdsaSignatureR, ecdsaSignatureS) <- ECDSA.Sign(ecdsaSecretKey,
                                                 dataDigest)
```

and

```
(verified) <- ECDSA.Verify(ecdsaPublicKey, ecdsaSignatureR,
                           ecdsaSignatureS, dataDigest)
```

The public keys MUST be encoded in SEC1 format as defined in section Section 2.1.1. The secret key, as well as both values R and S of the signature MUST each be encoded as a big-endian integer in a fixed-length octet string of the specified size.

The following table describes the ECDSA parameters and artifact lengths:

Algorithm ID reference	Curve	Field size	Public key	Secret key	Signature value R	Signature value S
TBD (ML-DSA-65 + ECDSA-NIST-P-256)	NIST P-256	32	65	32	32	32
TBD (ML-DSA-87 + ECDSA-NIST-P-384)	NIST P-384	48	97	48	48	48
TBD (ML-DSA-65 + ECDSA-brainpoolP256r1)	brainpoolP256r1	32	65	32	32	32
TBD (ML-DSA-87 + ECDSA-brainpoolP384r1)	brainpoolP384r1	48	97	48	48	48

Table 11: ECDSA parameters and artifact lengths in octets

6.1.3. ML-DSA signatures

For ML-DSA signature generation the default hedged version of ML-DSA.Sign given in [FIPS-204] is used. That is, to sign with ML-DSA the following operation is defined:

(mldsaSignature) <- ML-DSA.Sign(mldsaSecretKey, dataDigest)

For ML-DSA signature verification the algorithm ML-DSA.Verify given in [FIPS-204] is used. That is, to verify with ML-DSA the following operation is defined:

(verified) <- ML-DSA.Verify(mldsaPublicKey, dataDigest, mldsaSignature)

ML-DSA has the parametrization with the corresponding artifact lengths in octets as given in Table 12. All artifacts are encoded as defined in [FIPS-204].

Algorithm ID reference	ML-DSA	Public key	Secret key	Signature value
TBD	ML-DSA-65	1952	4032	3293
TBD	ML-DSA-87	2592	4896	4595

Table 12: ML-DSA parameters and artifact lengths in octets

6.2. Composite Signature Schemes with ML-DSA

6.2.1. Signature data digest

Signature data (i.e. the data to be signed) is digested prior to signing operations, see [I-D.ietf-openpgp-crypto-refresh] Section 5.2.4. Composite ML-DSA + ECC signatures MUST use the associated hash algorithm as specified in Table 13 for the signature data digest. Signatures using other hash algorithms MUST be considered invalid.

An implementation supporting a specific ML-DSA + ECC algorithm MUST also support the matching hash algorithm.

Algorithm ID reference	Hash function	Hash function ID reference
TBD (ML-DSA-65 IDs)	SHA3-256	12
TBD (ML-DSA-87 IDs)	SHA3-512	14

Table 13: Binding between ML-DSA and signature data digest

6.2.2. Key generation procedure

The implementation MUST independently generate the ML-DSA and the ECC component keys. ML-DSA key generation follows the specification [FIPS-204] and the artifacts are encoded as fixed-length octet strings as defined in Section 6.1.3. For ECC this is done following the relative specification in [RFC7748], [SP800-186], or [RFC5639], and encoding the artifacts as specified in Section 6.1.1 or Section 6.1.2 as fixed-length octet strings.

6.2.3. Signature Generation

To sign a message M with ML-DSA + EdDSA the following sequence of operations has to be performed:

1. Generate dataDigest according to [I-D.ietf-openpgp-crypto-refresh] Section 5.2.4
2. Create the EdDSA signature over dataDigest with EdDSA.Sign() from Section 6.1.1
3. Create the ML-DSA signature over dataDigest with ML-DSA.Sign() from Section 6.1.3
4. Encode the EdDSA and ML-DSA signatures according to the packet structure given in Section 6.3.1.

To sign a message M with ML-DSA + ECDSA the following sequence of operations has to be performed:

1. Generate dataDigest according to [I-D.ietf-openpgp-crypto-refresh] Section 5.2.4
2. Create the ECDSA signature over dataDigest with ECDSA.Sign() from Section 6.1.2
3. Create the ML-DSA signature over dataDigest with ML-DSA.Sign() from Section 6.1.3
4. Encode the ECDSA and ML-DSA signatures according to the packet structure given in Section 6.3.1.

6.2.4. Signature Verification

To verify a ML-DSA + EdDSA signature the following sequence of operations has to be performed:

1. Verify the EdDSA signature with EdDSA.Verify() from Section 6.1.1
2. Verify the ML-DSA signature with ML-DSA.Verify() from Section 6.1.3

To verify a ML-DSA + ECDSA signature the following sequence of operations has to be performed:

1. Verify the ECDSA signature with ECDSA.Verify() from Section 6.1.2

2. Verify the ML-DSA signature with ML-DSA.Verify() from Section 6.1.3

As specified in Section 4.3 an implementation MUST validate both signatures, i.e. EdDSA/ECDSA and ML-DSA, successfully to state that a composite ML-DSA + ECC signature is valid.

6.3. Packet Specifications

6.3.1. Signature Packet (Tag 2)

The composite ML-DSA + ECC schemes MUST be used only with v6 signatures, as defined in [I-D.ietf-openpgp-crypto-refresh].

The algorithm-specific v6 signature parameters for ML-DSA + EdDSA signatures consists of:

* A fixed-length octet string representing the EdDSA signature, whose length depends on the algorithm ID as specified in Table 10.

* A fixed-length octet string of the ML-DSA signature value, whose length depends on the algorithm ID as specified in Table 12.

The algorithm-specific v6 signature parameters for ML-DSA + ECDSA signatures consists of:

* A fixed-length octet string of the big-endian encoded ECDSA value R, whose length depends on the algorithm ID as specified in Table 11.

* A fixed-length octet string of the big-endian encoded ECDSA value S, whose length depends on the algorithm ID as specified in Table 11.

* A fixed-length octet string of the ML-DSA signature value, whose length depends on the algorithm ID as specified in Table 12.

6.3.2. Key Material Packets

The composite ML-DSA + ECC schemes MUST be used only with v6 keys, as defined in [I-D.ietf-openpgp-crypto-refresh].

The algorithm-specific public key for ML-DSA + EdDSA keys is this series of values:

* A fixed-length octet string representing the EdDSA public key, whose length depends on the algorithm ID as specified in Table 10.

* A fixed-length octet string containing the ML-DSA public key, whose length depends on the algorithm ID as specified in Table 12.

The algorithm-specific secret key for ML-DSA + EdDSA keys is this series of values:

* A fixed-length octet string representing the EdDSA secret key, whose length depends on the algorithm ID as specified in Table 10.

* A fixed-length octet string containing the ML-DSA secret key, whose length depends on the algorithm ID as specified in Table 12.

The algorithm-specific public key for ML-DSA + ECDSA keys is this series of values:

* A fixed-length octet string representing the ECDSA public key in SEC1 format, as specified in section Section 2.1.1 and with length specified in Table 11.

* A fixed-length octet string containing the ML-DSA public key, whose length depends on the algorithm ID as specified in Table 12.

The algorithm-specific secret key for ML-DSA + ECDSA keys is this series of values:

* A fixed-length octet string representing the ECDSA secret key as a big-endian encoded integer, whose length depends on the algorithm used as specified in Table 11.

* A fixed-length octet string containing the ML-DSA secret key, whose length depends on the algorithm ID as specified in Table 12.

7. SLH-DSA

7.1. The SLH-DSA Algorithms

The following table describes the SLH-DSA parameters and artifact lengths:

Parameter ID reference	Parameter name suffix	SLH-DSA public key	SLH-DSA secret key	SLH-DSA signature
1	128s	32	64	7856
2	128f	32	64	17088
3	192s	48	96	16224
4	192f	48	96	35664
5	256s	64	128	29792
6	256f	64	128	49856

Table 14: SLH-DSA parameters and artifact lengths in octets. The values equally apply to the parameter IDs of SLH-DSA-SHA2 and SLH-DSA-SHAKE.

7.1.1. Signature Data Digest

Signature data (i.e. the data to be signed) is digested prior to signing operations, see [I-D.ietf-openpgp-crypto-refresh] Section 5.2.4. SLH-DSA signatures MUST use the associated hash algorithm as specified in Table 15 for the signature data digest. Signatures using other hash algorithms MUST be considered invalid.

An implementation supporting a specific SLH-DSA algorithm and parameter MUST also support the matching hash algorithm.

Algorithm ID reference	Parameter ID reference	Hash function	Hash function ID reference
TBD (109 for testing)	1, 2	SHA-256	8
TBD (109 for testing)	3, 4, 5, 6	SHA-512	10
TBD (SLH-DSA-SHAKE)	1, 2	SHA3-256	12
TBD (SLH-DSA-SHAKE)	3, 4, 5, 6	SHA3-512	14

Table 15: Binding between SLH-DSA and signature data digest

7.1.2. Key generation

SLH-DSA key generation is performed via the algorithm SLH-DSA.KeyGen as specified in [FIPS-205], and the artifacts are encoded as fixed-length octet strings as defined in Section 7.1.

7.1.3. Signature Generation

SLH-DSA signature generation is performed via the algorithm SLH-DSA.Sign as specified in [FIPS-205]. The variable opt_rand is set to PK.seed. See also Section 10.5.

An implementation MUST set the Parameter ID in the signature equal to the issuing secret key Parameter ID.

7.1.4. Signature Verification

SLH-DSA signature verification is performed via the algorithm SLH-DSA.Verify as specified in [FIPS-205].

An implementation MUST check that the Parameter ID in the signature and in the key match when verifying.

7.2. Packet specifications

7.2.1. Signature Packet (Tag 2)

The SLH-DSA scheme MUST be used only with v6 signatures, as defined in [I-D.ietf-openpgp-crypto-refresh] Section 5.2.3.

The algorithm-specific v6 Signature parameters consists of:

* A one-octet value specifying the SLH-DSA parameter ID defined in Table 3 and Table 4. The values 0x00 and 0xFF are reserved for future extensions.

* A fixed-length octet string of the SLH-DSA signature value, whose length depends on the parameter ID in the format specified in Table 14.

7.2.2. Key Material Packets

The SLH-DSA scheme MUST be used only with v6 keys, as defined in [I-D.ietf-openpgp-crypto-refresh].

The algorithm-specific public key is this series of values:

* A one-octet value specifying the SLH-DSA parameter ID defined in Table 3 and Table 4. The values 0x00 and 0xFF are reserved for future extensions.

* A fixed-length octet string containing the SLH-DSA public key, whose length depends on the parameter ID as specified in Table 14.

The algorithm-specific secret key is this value:

* A fixed-length octet string containing the SLH-DSA secret key, whose length depends on the parameter ID as specified in Table 11.

8. Notes on Algorithms

8.1. Symmetric Algorithms for SEIPD Packets

Implementations MUST implement AES-256. An implementation SHOULD use AES-256 in the case of a v1 SEIPD packet, or AES-256 with any available AEAD mode in the case of a v2 SEIPD packet, if all recipients indicate support for it (explicitly or implicitly).

A v4 or v6 certificate that contains a PQ(/T) key SHOULD include AES-256 in the "Preferred Symmetric Ciphers for v1 SEIPD" subpacket. A v6 certificate that contains a PQ(/T) key SHOULD include the pair AES-256 with OCB in the "Preferred AEAD Ciphersuites" subpacket.

If AES-256 is not explicitly in the list of the "Preferred Symmetric Ciphers for v1 SEIPD" subpacket, and if the certificate contains a PQ/T key, it is implicitly at the end of the list. This is justified since AES-256 is mandatory to implement. If AES-128 is also implictly added to the list, it is added after AES-256.

If the pair AES-256 with OCB is not explicitly in the list of the "Preferred AEAD Ciphersuites" subpacket, and if the certificate contains a PQ/T key, it is implicitly at the end of the list. This is justified since AES-256 and OCB are mandatory to implement. If the pair AES-128 with OCB is also implictly added to the list, it is added after the pair AES-256 with OCB.

9. Migration Considerations

The post-quantum KEM algorithms defined in Table 1 and the signature algorithms defined in Table 2 are a set of new public key algorithms that extend the algorithm selection of [I-D.ietf-openpgp-crypto-refresh]. During the transition period, the post-quantum algorithms will not be supported by all clients. Therefore various migration considerations must be taken into account, in particular backwards compatibility to existing

implementations that have not yet been updated to support the post-quantum algorithms.

9.1. Key preference

Implementations SHOULD prefer PQ(/T) keys when multiple options are available.

For instance, if encrypting for a recipient for which both a valid PQ/T and a valid ECC certificate are available, the implementation SHOULD choose the PQ/T certificate. In case a certificate has both a PQ/T and an ECC encryption-capable valid subkey, the PQ/T subkey SHOULD be preferred.

An implementation MAY sign with both a PQ(/T) and an ECC key using multiple signatures over the same data as described in Section 4.4. Signing only with PQ(/T) key material is not backwards compatible.

Note that the confidentiality of a message is not post-quantum secure when encrypting to multiple recipients if at least one recipient does not support PQ/T encryption schemes. An implementation SHOULD NOT abort the encryption process in this case to allow for a smooth transition to post-quantum cryptography.

9.2. Key generation strategies

It is RECOMMENDED to generate fresh secrets when generating PQ(/T) keys. Note that reusing key material from existing ECC keys in PQ(/T) keys does not provide backwards compatibility.

An OpenPGP certificate is composed of a certification-capable primary key and one or more subkeys for signature, encryption, and authentication. Two migration strategies are recommended:

1. Generate two independent certificates, one for PQ(/T)-capable implementations, and one for legacy implementations. Implementations not understanding PQ(/T) certificates can use the legacy certificate, while PQ(/T)-capable implementations will prefer the newer certificate. This allows having an older v4 or v6 certificate for compatibility and a v6 PQ(/T) certificate, at a greater complexity in key distribution.

2. Attach PQ(/T) encryption subkeys to an existing traditional OpenPGP certificate. In the case of a v6 certificate, also PQ(/T) signature keys may be attached. Implementations understanding PQ(/T) will be able to parse and use the subkeys, while PQ(/T)-incapable implementations can gracefully ignore them. This simplifies key distribution, as only one certificate needs to be communicated and verified, but leaves the primary key vulnerable to quantum computer attacks.

10. Security Considerations

10.1. Security Aspects of Composite Signatures

When multiple signatures are applied to a message, the question of the protocol's resistance against signature stripping attacks naturally arises. In a signature stripping attack, an adversary removes one or more of the transmitted signatures such that only a subset of the signatures originally applied by the sender remain in the message that reaches the recipient. This amounts to a downgrade attack that potentially reduces the value of the signature. It should be noted that the composite signature schemes specified in this draft are not subject to a signature stripping vulnerability. This is due to the fact that in any OpenPGP signature, the hashed meta data includes the signature algorithm ID, as specified in [I-D.ietf-openpgp-crypto-refresh], Section 5.2.4. As a consequence, a component signature taken out of the context of a specific composite algorithm is not a valid signature for any message.

Furthermore, it is also not possible to craft a new signature for a message that was signed twice with a composite algorithm by interchanging (i.e., remixing) the component signatures, which would classify as a weak existential forgery. This is due to the fact that each v6 signatures also includes a random salt at the start of the hashed meta data, as also specified in the aforementioned reference.

10.2. Hashing in ECC-KEM

Our construction of the ECC-KEMs, in particular the inclusion of eccCipherText in the final hashing step in encapsulation and decapsulation that produces the eccKeyShare, is standard and known as hashed ElGamal key encapsulation, a hashed variant of ElGamal encryption. It ensures IND-CCA2 security in the random oracle model under some Diffie-Hellman intractability assumptions [CS03]. The additional inclusion of eccPublicKey follows the security advice in Section 6.1 of [RFC7748].

10.3. Key combiner

For the key combination in Section 5.2.2 this specification limits itself to the use of KMAC. The sponge construction used by KMAC was proven to be indifferentiable from a random oracle [BDPA08]. This means, that in contrast to SHA2, which uses a Merkle-Damgard construction, no HMAC-based construction is required for key combination. Except for a domain separation it is sufficient to simply process the concatenation of any number of key shares when using a sponge-based construction like KMAC. The construction using KMAC ensures a standardized domain separation. In this case, the processed message is then the concatenation of any number of key shares.

More precisely, for a given capacity c the indifferentiability proof shows that assuming there are no weaknesses found in the Keccak permutation, an attacker has to make an expected number of 2^(c/2) calls to the permutation to tell KMAC from a random oracle. For a random oracle, a difference in only a single bit gives an unrelated, uniformly random output. Hence, to be able to distinguish a key K, derived from shared keys K1 and K2 (and ciphertexts C1 and C2) as

```
K = KMAC(domainSeparation, counter || K1 || C1 || K2 || C2 || fixedInfo,
         outputBits, customization)
```

from a random bit string, an adversary has to know (or correctly guess) both key shares K1 and K2, entirely.

The proposed construction in Section 5.2.2 preserves IND-CCA2 of any of its ingredient KEMs, i.e. the newly formed combined KEM is IND-CCA2 secure as long as at least one of the ingredient KEMs is. Indeed, the above stated indifferentiability from a random oracle qualifies Keccak as a split-key pseudorandom function as defined in [GHP18]. That is, Keccak behaves like a random function if at least one input shared secret is picked uniformly at random. Our construction can thus be seen as an instantiation of the IND-CCA2 preserving Example 3 in Figure 1 of [GHP18], up to some reordering of input shared secrets and ciphertexts. In the random oracle setting, the reordering does not influence the arguments in [GHP18].

10.4. Domain separation and binding

The domSeparation information defined in Section 5.2.2 provides the domain separation for the key combiner construction. This ensures that the input keying material is used to generate a KEK for a specific purpose or context.

The fixedInfo defined in Section 5.2.1 binds the derived KEK to the chosen algorithm and communication parties. The algorithm ID identifies unequivocally the algorithm, the parameters for its instantiation, and the length of all artifacts, including the derived key.

This is in line with the Recommendation for ECC in section 5.5 of [SP800-56A]. Other fields included in the recommendation are not relevant for the OpenPGP protocol, since the sender is not required to have a key of their own, there are no pre-shared secrets, and all the other parameters are unequivocally defined by the algorithm ID.

Furthermore, we do not require the recipients public key into the key combiner as the public key material is already included in the component key derivation functions. Given two KEMs which we assume to be multi-user secure, we combine their outputs using a KEM-combiner:

K = H(K1, C1, K2, C2), C = (C1, C2)

Our aim is to preserve multi-user security. A common approach to this is to add the public key into the key derivation for K. However, it turns out that this is not necessary here. To break security of the combined scheme in the multi-user setting, the adversary has to distinguish a set of challenge keys

K__u = H(K1__u, C1__u, K2__u, C2*_u)

for users u in some set from random, also given ciphertexts C*_u = (C1*_u, C2*_u). For each of these K* it holds that if the adversary never makes a query

H(K1*_u, C1*_u, K2*_u, C2*_u)

they have a zero advantage over guessing.

The only multi-user advantage that the adversary could gain therefore consists of queries to H that are meaningful for two different users u1 != u2 and their associated public keys. This is only the case if

(c1*_u1, c2*_u1) = (c1*_u2, c2*_u2)

as the ciphertext values decide for which challenge the query is meaningful. This means that a ciphertext collision is needed between challenges. Assuming that the randomness used in the generation of the two challenges is uncorrelated, this is negligible.

In consequence, the ciphertexts already work sufficiently well as domain-separator.

10.5. SLH-DSA Message Randomizer

The specification of SLH-DSA [FIPS-205] prescribes an optional non-deterministic message randomizer. This is not used in this specification, as OpenPGP v6 signatures already provide a salted signature data digest of the appropriate size.

10.6. Binding hashes in signatures with signature algorithms

In order not to extend the attack surface, we bind the hash algorithm used for signature data digestion to the hash algorithm used internally by the signature algorithm.

ML-DSA internally uses a SHAKE256 digest, therefore we require SHA3 in the ML-DSA + ECC signature packet, see Section 6.2.1. Note that we bind a NIST security category 2 hash function to a signature algorithm that falls into NIST security category 3. This does not constitute a security bottleneck: because of the unpredictable random salt that is prepended to the digested data in v6 signatures, the hardness assumption is not collision resistance but second-preimage resistance.

In the case of SLH-DSA the internal hash algorithm varies based on the algorithm and parameter ID, see Section 7.1.1.

10.7. Symmetric Algorithms for SEIPD Packets

This specification mandates support for AES-256 for two reasons. First, AES-KeyWrap with AES-256 is already part of the composite KEM construction. Second, some of the PQ(/T) algorithms target the security level of AES-256.

For the same reasons, this specification further recommends the use of AES-256 if it is supported by all recipients, regardless of what the implementation would otherwise choose based on the recipients' preferences. This recommendation should be understood as a clear and simple rule for the selection of AES-256 for encryption. Implementations may also make more nuanced decisions.

11. Additional considerations

11.1. Performance Considerations for SLH-DSA

This specification introduces both ML-DSA + ECC as well as SLH-DSA as PQ(/T) signature schemes.

Generally, it can be said that ML-DSA + ECC provides a performance in terms of execution time requirements that is close to that of traditional ECC signature schemes. Regarding the size of signatures and public keys, though, ML-DSA has far greater requirements than traditional schemes like EC-based or even RSA signature schemes. Implementers may want to offer SLH-DSA for applications where a higher degree of trust in the signature scheme is required. However, SLH-DSA has performance characteristics in terms of execution time of the signature generation as well as space requirements for the signature that are even greater than those of ML-DSA + ECC signature schemes.

Pertaining to the execution time, the particularly costly operation in SLH-DSA is the signature generation. In order to achieve short signature generation times, one of the parameter sets with the name ending in the letter "f" for "fast" should be chosen. This comes at the expense of a larger signature size.

In order to minimize the space requirements of a SLH-DSA signature, a parameter set ending in "s" for "small" should be chosen. This comes at the expense of a longer signature generation time.

12. IANA Considerations

IANA is requested to add the following registries to the OpenPGP registry group at https://www.iana.org/assignments/openpgp:

* Registry name: OpenPGP SLH-DSA-SHA2 parameters

  Registration procedure: SPECIFICATION REQUIRED [RFC8126]

  The registry contains the values defined in Table 3 in this document.

* Registry name: OpenPGP SLH-DSA-SHAKE parameters

  Registration procedure: SPECIFICATION REQUIRED [RFC8126]

  The registry contains the values defined in Table 4 in this document.

Furthermore, IANA is requested to add the algorithm IDs defined in Table 16 to the existing registry OpenPGP Public Key Algorithms. The field specifications enclosed in brackets for the ML-KEM + ECDH composite algorithms denote fields that are only conditionally contained in the data structure.

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format	Reference
TBD	ML-KEM-768 + X25519	32 octets X25519 public key (Table 5), 1184 octets ML-KEM-768 public key (Table 8)	32 octets X25519 secret key (Table 5), 2400 octets ML-KEM-768 secret-key (Table 8)	N/A	32 octets X25519 ciphertext, 1088 octets ML-KEM-768 ciphertext [, 1 octet algorithm ID in case of v3 PKESK], 1 octet length field of value n, n octets wrapped session key (Section 5.3.1)	Section 5.2
TBD	ML-KEM-1024 + X448	56 octets X448 public key (Table 5), 1568 octets ML-KEM-1024 public key (Table 8)	56 octets X448 secret key (Table 5), 3168 octets ML-KEM-1024 secret-key (Table 8)	N/A	56 octets X448 ciphertext, 1568 octets ML-KEM-1024 ciphertext [, 1 octet algorithm ID in case of v3 PKESK], 1 octet length field of value n, n octets wrapped session key (Section 5.3.1)	Section 5.2
TBD	ML-DSA-65 + Ed25519	32 octets Ed25519 public key (Table	32 octets Ed25519 secret key (Table	64 octets Ed25519 signature (Table 10), 3293	N/A	Section 6.2

		10), 1952 octets ML-DSA-65 public key (Table 12)	10), 4032 octets ML-DSA-65 secret (Table 12)	octets ML-DSA-65 signature (Table 12)			
TBD	ML-DSA-87 + Ed448	57 octets Ed448 public key (Table 10), 2592 octets ML-DSA-87 public key (Table 12)	57 octets Ed448 secret key (Table 10), 4896 octets ML-DSA-87 secret (Table 12)	114 octets Ed448 signature (Table 10), 4595 octets ML-DSA-87 signature (Table 12)	N/A	Section 6.2	
TBD	SLH-DSA-SHA2	1 octet parameter ID, per parameter fixed-length octet string (Table 14)	per parameter fixed-length octet string (Table 14)	1 octet parameter ID, per parameter fixed-length octet string (Table 14)	N/A	Section 7.1	
TBD	SLH-DSA-SHAKE	1 octet parameter ID, per parameter fixed-length octet string (Table 14)	per parameter fixed-length octet string (Table 14)	1 octet parameter ID, per parameter fixed-length octet string (Table 14)	N/A	Section 7.1	

Table 16: IANA updates for registry 'OpenPGP Public Key Algorithms'

13. Changelog

13.1. draft-wussler-openpgp-pqc-01

* Shifted the algorithm IDs by 4 to align with the crypto-refresh.
* Renamed v5 packets into v6 to align with the crypto-refresh.
* Defined IND-CCA2 security for KDF and key combination.
* Added explicit key generation procedures.
* Changed the key combination KMAC salt.
* Mandated Parameter ID check in SPHINCS+ signature verification.
* Fixed key share size for Kyber-768.
* Added "Preliminaries" section.
* Fixed IANA considerations.

13.2. draft-wussler-openpgp-pqc-02

* Added the ephemeral and public key in the ECC key derivation function.
* Removed public key hash from key combiner.
* Allowed v3 PKESKs and v4 keys with PQ algorithms, limiting them to AES symmetric ciphers. for encryption with SEIPDv1, in line with the crypto-refresh.

13.3. draft-wussler-openpgp-pqc-03

* Replaced round 3 submission with NIST PQC Draft Standards FIPS 203, 204, 205.
* Added consideration about security level for hashes.

13.4. draft-wussler-openpgp-pqc-04

* Added Johannes Roth as author

13.5. draft-ietf-openpgp-pqc-00

* Renamed draft

13.6. draft-ietf-openpgp-pqc-01

* Mandated AES-256 as mandatory to implement.
* Added AES-256 / AES-128 with OCB implicitly to v1/v2 SEIPD preferences of "PQ(/T) certificates".
* Added a recommendation to use AES-256 when possible.
* Swapped the optional v3 PKESK algorithm identifier with length octet in order to align with X25519 and X448.
* Fixed ML-DSA private key size
* Added test vectors
* correction and completion of IANA instructions

13.7. draft-ietf-openpgp-pqc-02

* Removed git rebase artifact

14. Contributors

Stephan Ehlen (BSI)
Carl-Daniel Hailfinger (BSI)
Andreas Huelsing (TU Eindhoven)

15. References

15.1. Normative References

[I-D.ietf-openpgp-crypto-refresh]
Wouters, P., Huigens, D., Winter, J., and N. Yutaka, "OpenPGP", Work in Progress, Internet-Draft, draft-ietf-openpgp-crypto-refresh-13, 4 January 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-openpgp-crypto-refresh-13>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <https://www.rfc-editor.org/rfc/rfc3394>.

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

15.2. Informative References

[BDPA08] Bertoni, G., Daemen, J., Peters, M., and G. Assche, "On the Indifferentiability of the Sponge Construction", 2008, <https://doi.org/10.1007/978-3-540-78967-3_11>.

[CS03] Cramer, R. and V. Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack", 2003, <https://doi.org/10.1137/S0097539702403773>.

[draft-driscoll-pqt-hybrid-terminology] Driscoll, F., "Terminology for Post-Quantum Traditional Hybrid Schemes", March 2023, <https://datatracker.ietf.org/doc/html/draft-driscoll-pqt-hybrid-terminology>.

[FIPS-203] National Institute of Standards and Technology, "Module-Lattice-Based Key-Encapsulation Mechanism Standard", August 2023, <https://doi.org/10.6028/NIST.FIPS.203.ipd>.

[FIPS-204] National Institute of Standards and Technology, "Module-Lattice-Based Digital Signature Standard", August 2023, <https://doi.org/10.6028/NIST.FIPS.204.ipd>.

[FIPS-205] National Institute of Standards and Technology, "Stateless Hash-Based Digital Signature Standard", August 2023, <https://doi.org/10.6028/NIST.FIPS.205.ipd>.

[GHP18] Giacon, F., Heuer, F., and B. Poettering, "KEM Combiners", 2018, <https://doi.org/10.1007/978-3-319-76578-5_7>.

[NIST-PQC] Chen, L., Moody, D., and Y. Liu, "Post-Quantum Cryptography Standardization", December 2016, <https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization>.

[NISTIR-8413]
Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., Perlner, R., Robinson, A., Smith-Tone, D., and Y. Liu, "Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process", NIST IR 8413 , September 2022, <https://doi.org/10.6028/NIST.IR.8413-upd1>.

[RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <https://www.rfc-editor.org/rfc/rfc5639>.

[SEC1] Standards for Efficient Cryptography Group, "Standards for Efficient Cryptography 1 (SEC 1)", May 2009, <https://secg.org/sec1-v2.pdf>.

[SP800-185]
Kelsey, J., Chang, S., and R. Perlner, "SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash", NIST Special Publication 800-185 , December 2016, <https://doi.org/10.6028/NIST.SP.800-185>.

[SP800-186]
Chen, L., Moody, D., Regenscheid, A., and K. Randall, "Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters", NIST Special Publication 800-186 , February 2023, <https://doi.org/10.6028/NIST.SP.800-186>.

[SP800-56A]
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Rev. 3 , April 2018, <https://doi.org/10.6028/NIST.SP.800-56Ar3>.

[SP800-56C]
Barker, E., Chen, L., and R. Davis, "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", NIST Special Publication 800-56C Rev. 2 , August 2020, <https://doi.org/10.6028/NIST.SP.800-56Cr2>.

Appendix A. Test Vectors

To help implementing this specification a set of non-normative examples follow here. The test vectors are implemented using the Initial Public Draft (IPD) variant of the ML-DSA and ML-KEM schemes.

A.1. Sample v6 PQC Subkey Artifacts

Here is a Private Key consisting of:

* A v6 Ed25519 Private-Key packet
* A v6 direct key self-signature
* A User ID packet
* A v6 positive certification self-signature
* A v6 ML-KEM-ipd-768 + X25519 Private-Subkey packet
* A v6 subkey binding signature

The primary key has the fingerprint 523432423452540502l9ceff286e9c8e479ec88757f95354388984a02d7d0b59.

The subkey has the fingerprint 263e34b69938e753dc67ca8ee37652795135e0e16e48887103c11d7307df40ed.

```
-----BEGIN PGP PRIVATE KEY BLOCK-----

xUsGUdDGgBsAAAAgsJV1qyvdl+EenEB4IFvP5/7Ci5XJ1rk8Yh967qV1rb0A8q5N
oCO2TM6GoqWftH02oIwWpAr+kvA+4CH7N3cpPSrCrwYfGwoAAABABQJR0MaAIqEG
UjQyQjRSVAUCGc7/KG6cjkeeyIdX+VNUOImEoC19C1kCGwMCHgkDCwkHAxUKCAIW
AAUnCQIHAgAAAADhOyBW8CPDe5FreFmlonhfVhr2EPw3WFLyd6mKRhkQm3VBfw7Q
w7eermL9Cr5O7Ah0JxmIkT18jgKQr9AwWa3nm2mcbjSoib2WVzm5EiW3f3lgflfr
ySQFpSICzPl2QcAcrgjNLlBRQyB1c2VyIChUZXN0IEtleSkgPHBxYy10ZXN0LWtl
eUBleGFtcGxlLmNvbT7CmwYTGwoAAAAsBQJR0MaAIqEGUjQyQjRSVAUCGc7/KG6c
jkeeyIdX+VNUOImEoC19C1kCGQEAAAAAg2ogTEbKVVlbWsejQHkq7xo8ipM7dv6H
z2AekkJqupKVR+/oy+2j6ri+/B2K6k1v1y5quzirhs87fB5AxZC6ZoFDvC0kZOvo
14fPF07wCx0jwJVOWuRFVsVw7pQJHbNzgkIAx82LBlHQxoBpAAAEwLRbSSpvve2p
Ih3hHweqq2VdRo+7Zf7whYHyXM/UifsniwMKSrubvsmLgCyiEwMip3ZlTSxIFDaF
EMVtVvCSJ7XFZ0WslTJnZ/CENPgxbVgn6CC2b8UEb8olS3AxlSiqJSRP0OrOJdfP
WJI1A+p7Vmw1CZQq2oVPUlE96SVUrFxfk7XCYpcTpIQb+mFB4ULCesat5tud7Tau
UJpMKssUf0I74EUjahoR46pPReKzlSqfvhpgXSASZpBg8IZBY7VbgTnLInGTTnEr
rScVlDnAwcdYvuZMQYO5EjS6LOxn1aVfU+iH+Rir2AyFzsYl6ICHciPAsKKa+Sk7
UPFBrIRG1qgn7FF0n5epHeiFCRNb87wSqlp0h+d8L3jPmDq4zoQPKDViasoHYXLD
7KoJTIxP2eGzjMRlg3oD9ph3ZnyOTIsx/4SDtxW3q+JU8RFoI0dZEdURwaoIITWi
tldtPUmtBuJshceEDSWopuwLzBuVTnYDpTy94ZtDBKmgPnmSmPOKZ6THucmiJGUm
```

```
WmAKkyo7kWAwYRsE2ZYqLzIJFmZFzRLIThipiZhR/9h2GemQklMJqYs25cEGx6FW
zXRv8Palm7yOAicH/ldHUOtU3oFIXthOatwSrQApJ7HHvksx59ZtLFtBgHm5eRmY
YleJsJLGCPssa7pK2hIwgLlmCLSAavFqYjuocWIYKLmw5vNXXRWIjPBbTpVXbUO5
U9F/67gggSWBJXCZlfgcluO422aN22m8aONiTgZtmjcC2elci5yRKGBbeKmFTcVs
ZbpbY6ZCKFRyzbqmMGYe0mqN6lh7R5dNiBuJZQg04mYuSzWCF3mumlJTRtlN9Miy
6LyWApJSTQdgc3awS0mjUrgU1Ia0AjMFKcxJA6iHd6iAxWMbUqxOSoTOTUlMr3lt
paNGEMGpaHwMoQs99xSI1zG9pYmfeIl6LfZSwnI4LsBvNOBiUhNUC/aYIILEm7qj
Tpw5YdI+6jSl+palLlcMDzt0LgMN8rY6UlZJBGNFSAKSNSWXdFYMByKKGSCj91TD
WPlOLvWKntSLk5eLodhgmRGqx5GZECgWS4wDARY00rl17dV53GejXrUtJaYcnam5
pKoTSaPJTuY25Kyy+oB7aHpV0vA87JaeRCsqkjcS5IQKdtceUskXNRa2f7CTrfQR
hOGk0gSA4Jx8+Fw8uGWLGJx6m1lSyWcMX5HL7hJkFhEKebYjdALGXMV1wxNiUHCI
vxCjX/AkwHEDvAN6qhULrcZlmngSbeBysOFud2a8PIS2p7RCAatO+TpFgoR+1CgV
JIdiRpM0WrMfS9iBERhtYaLH1oUjBpcV7zpgNdkT4ClfbTpgu3oPnWBogDjMXKUe
pSfFx0l1tNGRLCCFVit8xxA4Q+phutInyXUAHJiEfHIR4jxTd/FwQ3pDoKxTesY+
XsGtVJxe9oMrXSlt6uymn6zKQlQsw8odvHhp5/NWqkCh9/xQvmIlERsVVjyJ0FNF
/+HNT9KrECCj6+cujDbEN6UmRlFvlMcxFzYaTnWa1cshSVCCa1aYZddWrDdxOwMf
ObUw8TukY7A2RqcdpmpA68SLoWwNAgtFG1xWV43yC/P3XTsqTmgHRUGboDkVs9K8
1+Byg4jhKWcAksr2fFDB4wkkaZcB3uUOXuQQ2etC1aCrboS5vTeMVJVS+ssLkxle
KLZ3kH9pazHbNTKQWclexAe48RImOk1PlmN9HHMgUwgJI5H8e3a7cQw8x7Yh5wce
yAdhuwRGcT99CqtaQb0aeTz9xxh642roMy46rCQp2A/g1QbZIqqVe6lb4qkJ8YdM
dG4SrE3UzD3tuAyu3L9Ql79qxxdB4Jt7wp+dPETaoZba+aMWZ68ZxDEjQJcgyrN9
XCBNcLcU+SpjBXPK13yeCdAVGUhA1c0qB4PKVY5/e07Kc8qGgyrlJCCb05OQQKWG
mmVcJnDDIZSLM4VPd3cAgWhv5rIk/BPWQ6CGps6njH1WNaI6sTr35wcfWlMahs0w
mUPkKMG0AWwT9VBCBU7huFN7Rw2DXBdQUlQDO8WzVLXFt6sZvF+XgZ840woQ8I29
BmW55qSY2hdtMsKqkU31Nbscxa5wRsu2KSirXF3JoZkTacU/taIRmmIwGXl0zBlM
8Hp9hJOdAZAAPAYwCj8FdmD4AyDiHHDkuJsLfL80CnKck2wYbBE/BoGRKwVul1Jr
gh4KC4DS+WfKZQYam5KLAytFMUJf8TDiYYNmVr9TOVNAoCj4XKs7BQ7KZ5MMnCWi
EEsH9im2mBrHDKXLCrFK8IY54B5ae8uDKWwOuhTtlHki5CTVHHRKaorYawvMqTZ4
HCO+6Jrj8rm7YFxhxwPihVHIl10SK2Q2tX8ygidCKc1yPBh4lKyvyryPwL6i5sM4
sU5glM9bZgPKfHosk4uNdqZQ5FyIaohJ8aocQpr0JVQv8rp0UjBEDBqDeIhepohd
cp5KhA1kND4vQbfjusdVtgUorAqyAw0YSoeDLAfC5syaJqo8K06CM8y7O3VqB8Rs
ZJb8Eb7mGYdH9U8m3MTjestO5LcTAyqoBJvC4TTgp6F9dJ55HJ3rzFx19wMqGhLV
Abcw/JWJagrvYqTGozbiEcLheFNmKik4eGoG9mS1Ebhwhbmg5LD6kZXFK7hJOnkb
cTdz0ynSqlPk1oJkh8Pa1gVG4IWgEJISZWEb036BmTASRc5EYVetuBujMYQKuWeI
RrumhH3GiZBw1RIyrDYYMk37OHf0MLhahBeldJsqRoLcErOSu0T9xwmeczWoIDtZ
Q8794LDkCoY6wpYFF5Scq64HgmQaS5kSQH9UtTIgbLoBmQiDUIyrx8LoBqhOdQPR
0y60NWjSXLbs0VjxrIVMZmdlxH//gknkDLlSgSqbbAkG+7T9clLS44lVYD22N03n
Mil8pHWju6yYW3eFaylzI7jLEVZ5cLw15bd1JHEvRpOBxV8Fdn+p4RKoRrUN4EQm
1olEK4TsWY+uV2RCV4PEBQpOQxGZZxhMRa/AKnD3I1LjSlNh9SLXNbVIp69bPK9N
qS8MGBGeWBzEARhXea9mBiUisSFSZrwneYALPBXH0h4xerZWV2GH9bu12gwBmJbB
k64rwZg/dqDiCM16/C0Np0Aza4oTVsOJ6BrdZh70xFZq+Dizeg85TMywkl9Ma1BT
AsMOZ45sAEwIBhUX6Colkae023ouMgj1pnFV5Rc8cTSRcGUM1ZHW8AeLAwpKu5u+
yYuALKITAyKndmVNLEgUNoUQxW1W8JIntcVnRayVMmdn8IQ0+DFtWCfoILZvxQRv
yiVLcDGVKKolJE/Q6s4l189YkjUD6ntWbDUJlCrahU9SUT3pJVSsXF+TtcJilxOk
hBv6YUHhQsJ6xq3m253tNq5QmkwqyxR/QjvgRSNqGhHjqk9F4rOVKp++GmBdIBJm
kGDwhkFjtVuBOcsicZNOcSutJxWUOcDBx1i+5kxBg7kSNLos7GfVpV9T6If5GKvY
DIXOxiXogIdyI8Cwopr5KTtQ8UGshEbWqCfsUXSfl6kd6IUJE1vzvBKqWnSH53wv
eM+YOrjOhA8oNWJqygdhcsPsqglMjE/Z4bOMxGWDegP2mHdmfI5MizH/hIO3Fber
```

```
4lTxEWgjR1kR1RHBqgghNaK2V209Sa0G4myFx4QNJaim7AvMG5VOdgOlPL3hm0ME
qaA+eZKY84pnpMe5yaIkZSZaYAqTKjuRYDBhGwTZliovMgkWZkXNEshOGKmJmFH/
2HYZ6ZCSUwmpizblwQbHoVbNdG/w9qWbvI4CJwf+V0dQ61TegUhe2E5q3BKtACkn
sce+SzHn1m0sW0GAebl5GZhiV4mwksYI+yxrukraEjCAuWYItIBq8WpiO6hxYhgo
ubDm81ddFYiM8FtOlVdtQ7lT0X/ruCCBJYElcJmV+ByW47jbZo3babxo42JOBm2a
NwLZ6VyLnJEoYFt4qYVNxWxlultjpkIoVHLNuqYwZh7Sao3qWHtHl02IG4llCDTi
Zi5LNYIXea6aUlNG2U30yLLovJYCklJNB2BzdrBLSaNSuBTUhrQCMwUpzEkDqId3
qIDFYxtSrE5KhM5NSUyveW2lo0YQwalofAyhCz33FIjXMb2liZ94iXot9lLCcjgu
wG804GJSE1QL9pgggsSbuqNOnDlh0j7qNKX6lqUuVwwPO3QuAw3ytjpSVkkEY0VI
ApI1JZd0VgwHIooZIKP3VMNY+U4u9Yqe1IuTl4uh2GCZEarHkZkQKBZLjAMBFjTS
uXXt1XncZ6NetS0lphydqbmkqhNJo8lO5jbkrLL6gHtoelXS8Dzslp5EKyqSNxLk
hAp21x5SyRc1FrZ/sJOt9BGE4aTSBIDgnHz4XDy4ZYsYnHqbWVLJZwxfkcvuEmQW
EQp5tiN0AsZcxXDE2JQcIi/EKNf8CTAcQO8A3qqFQutxmWaeBJt4HKw4W53Zrw8
hLantEIBq075OkWChH7UKBUkh2JGkzRasx9L2IERGG1hosfWhSMGlxXvOmA12RPg
KV9tOmC7eg+dYGiAOMxcpR6lJ8XHSXW00ZEsIIVWK3zHEDhD6mG60ifJdQAcmIR8
chHiPFN38XBDekOgrFN6xj5ewa1UnF72gytdKW3q7KafrMpCVCzDyh28eGnn81aq
QKH3/FC+YiURGxVWPInQU0X/4c1P0qsQIKPr5y6MNsQ3pSZGUW+UxzEXNhpOdZrV
yyFJUIJrVphl11asN3E7Ax85tTDxO6RjsDZGpx2makDrxIuhbA0CC0UbXFZXjfIL
8/ddOypOaAdFQZugORWz0rzX4HKDiOEpZ7+6jJ8tjNCQrKgJg1wGCpAN0VnrtFrs
2l6Q0GteA6B+fwfjuRabwerw1ro7lcwOA5EiA6XO30P+pLG07ms2MCfCmwYYGwoA
AAAsBQJR0MaAIqEGUjQyQjRSVAUCGc7/KG6cjkeeyIdX+VNUOImEoC19C1kCGwwA
AAAA5kEgPwatbx3FHPIy9J9mGUEpUE03oRRPE8N4lJ2eAIMhciCEHp3BzYVGvW3O
aPYmjcu4JTREPJM6HP7yR+ZEg+Bld9lBSVmEdMJnOX2ZHOdEoRV4bm1U4aPuhrKL
/d8lkIgM
-----END PGP PRIVATE KEY BLOCK-----
```

Here is the corresponding Public Key consisting of:

* A v6 Ed25519 Public-Key packet
* A v6 direct key self-signature
* A User ID packet
* A v6 positive certification self-signature
* A v6 ML-KEM-ipd-768 + X25519 Public-Subkey packet
* A v6 subkey binding signature

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xioGUdDGgBsAAAAgsJV1qyvdl+EenEB4IFvP5/7Ci5XJ1rk8Yh967qV1rb3CrwYf
GwoAAABABQJR0MaAIqEGUjQyQjRSVAUCGc7/KG6cjkeeyIdX+VNUOImEoC19C1kC
GwMCHgkDCwkHAxUKCAIWAAUnCQIHAgAAAADhOyBW8CPDe5FreFmlonhfVhr2EPw3
WFLyd6mKRhkQm3VBfw7Qw7eermL9Cr5O7Ah0JxmIkT18jgKQr9AwWa3nm2mcbjSo
ib2WVzm5EiW3f3lgflfrySQFpSICzPl2QcAcrgjNLlBRQyB1c2VyIChUZXN0IEtl
eSkgPHBxYy10ZXN0LWtleUBleGFtcGxlLmNvbT7CmwYTGwoAAAAsBQJR0MaAIqEG
UjQyQjRSVAUCGc7/KG6cjkeeyIdX+VNUOImEoC19C1kCGQEAAAAAg2ogTEbKVVlb
WsejQHkq7xo8ipM7dv6Hz2AekkJqupKVR+/oy+2j6ri+/B2K6k1v1y5quzirhs87
fB5AxZC6ZoFDvC0kZOvo14fPF07wCx0jwJVOWuRFVsVw7pQJHbNzgkIAzsQKBlHQ
xoBpAAAEwLRbSSpvve2pIh3hHweqq2VdRo+7Zf7whYHyXM/UifsniwMKSrubvsmL
gCyiEwMip3ZlTSxIFDaFEMVtVvCSJ7XFZ0WslTJnZ/CENPgxbVgn6CC2b8UEb8ol
S3AxlSiqJSRP0OrOJdfPWJI1A+p7Vmw1CZQq2oVPUlE96SVUrFxfk7XCYpcTpIQb
+mFB4ULCesat5tud7TauUJpMKssUf0I74EUjahoR46pPReKzlSqfvhpgXSASZpBg
8IZBY7VbgTnLInGTTnErrScVlDnAwcdYvuZMQYO5EjS6LOxn1aVfU+iH+Rir2AyF
zsYl6ICHciPAsKKa+Sk7UPFBrIRG1qgn7FF0n5epHeiFCRNb87wSqlp0h+d8L3jP
mDq4zoQPKDViasoHYXLD7KoJTIxP2eGzjMRlg3oD9ph3ZnyOTIsx/4SDtxW3q+JU
8RFoI0dZEdURwaoIITWitldtPUmtBuJshceEDSWopuwLzBuVTnYDpTy94ZtDBKmg
PnmSmPOKZ6THucmiJGUmWmAKkyo7kWAwYRsE2ZYqLzIJFmZFzRLIThipiZhR/9h2
GemQklMJqYs25cEGx6FWzXRv8Palm7yOAicH/ldHUOtU3oFIXthOatwSrQApJ7HH
vksx59ZtLFtBgHm5eRmYYleJsJLGCPssa7pK2hIwgLlmCLSAavFqYjuocWIYKLmw
5vNXXRWIjPBbTpVXbUO5U9F/67gggSWBJXCZlfgcluO422aN22m8aONiTgZtmjcC
2elci5yRKGBbeKmFTcVsZbpbY6ZCKFRyzbqmMGYe0mqN6lh7R5dNiBuJZQg04mYu
SzWCF3mumlJTRtlN9Miy6LyWApJSTQdgc3awS0mjUrgU1Ia0AjMFKcxJA6iHd6iA
xWMbUqxOSoTOTUlMr3ltpaNGEMGpaHwMoQs99xSI1zG9pYmfeIl6LfZSwnI4LsBv
NOBiUhNUC/aYIILEm7qjTpw5YdI+6jSl+palLlcMDzt0LgMN8rY6UlZJBGNFSAKS
NSWXdFYMByKKGSCj91TDWPlOLvWKntSLk5eLodhgmRGqx5GZECgWS4wDARY00rl1
7dV53GejXrUtJaYcnam5pKoTSaPJTuY25Kyy+oB7aHpV0vA87JaeRCsqkjcS5IQK
dtceUskXNRa2f7CTrfQRhOGk0gSA4Jx8+Fw8uGWLGJx6m1lSyWcMX5HL7hJkFhEK
ebYjdALGXMV1wxNiUHCIvxCjX/AkwHEDvAN6qhULrcZlmngSbeBysOFud2a8PIS2
p7RCAatO+TpFgoR+1CgVJIdiRpM0WrMfS9iBERhtYaLH1oUjBpcV7zpgNdkT4Clf
bTpgu3oPnWBogDjMXKUepSfFx0l1tNGRLCCFVit8xxA4Q+phutInyXUAHJiEfHIR
4jxTd/FwQ3pDoKxTesY+XsGtVJxe9oMrXSlt6uymn6zKQlQsw8odvHhp5/NWqkCh
9/xQvmIlERsVVjyJ0FNF/+HNT9KrECCj6+cujDbEN6UmRlFvlMcxFzYaTnWa1csh
SVCCa1aYZddWrDdxOwMfObUw8TukY7A2RqcdpmpA68SLoWwNAgtFG1xWV43yC/P3
XTsqTmgHRUGboDkVs9K81+Byg4jhKWfCmwYYGwoAAAAsBQJR0MaAIqEGUjQyQjRS
VAUCGc7/KG6cjkeeyIdX+VNUOImEoC19C1kCGwwAAAAA5kEgPwatbx3FHPIy9J9m
GUEpUE03oRRPE8N4lJ2eAIMhciCEHp3BzYVGvW3OaPYmjcu4JTREPJM6HP7yR+ZE
g+Bld9lBSVmEdMJnOX2ZHOdEoRV4bm1U4aPuhrKL/d8lkIgM
-----END PGP PUBLIC KEY BLOCK-----
```

Here is an unsigned message "Testing\n" encrypted to this key:

* A v6 PKESK

* A v2 SEIPD

The hex-encoded KMAC eccKeyShare input is
4ec7dc0874ce4a3c257fec94f27f2d3c589764a5fbaf27a4b52836df53c86868.

The hex-encoded KMAC mlkemKeyShare input is
9a84cb01b6be6eecd16737fb558b5ca35899403076c7e9f0ee350195e7fbf6c4.

The hex-encoded KMAC256 output is
15a0f1eed1fb2a50a22f21e82dbce13ae91c45e3b76a9d2c61246c354a05f781.

The hex-encoded session key is
08f49fd5340b026e7ec751d82cea83a4b92d4837e785bfb66af71387f84156d0.

-----BEGIN PGP MESSAGE-----

wcPtBiEGJj40tpk451PcZ8qO43ZSeVE14OFuSIhxA8EdcwffQO1pU4rRGXPhivmf
yaE7whd94FIPayIJxbUXuq6Ei6VifzlPu9BoxvQYZa/u+exaOVT2MLxbCAceHYMw
zSuUa0BoiugafZWAnVrn4ji+mI+298c93Ij83yUjkzvBsKyJuhesTevSpJAnjiMt
m9Mmwzc8Y9tB4N/Am8jR3p8UYLH+2aH9FyT6VdqsETYiPFcz5jZqkag7bAB88KUg
heJAHU/FgHXtz013tnPyQPtuVHeJrP8jcd3IENJh9CfSg9rkhAoW72GiSGYPm+Im
I7faHJ8LMCx4Sbur9rWsZM5Gl+sG2MyjM3SrykAfL8W6s5o3LVw+06h5XnmsLUFE
8nZf99LBm/5aUzkdmq1G8hgLY+dVhTAm1DZxbNz5nDbtCsuIbJ2knS2yqxlQV8qx
5eikfWlW29TfREHJf4von2xR668vYwOibA5sTa3utBpNk2YNalZ61p9MxPW54PvO
+EIRYVTlwhSbETanOBms+yDq0NLooO+OQerT3WA1VzeT7j5rzaQM0ULAK6x/KqKt
ARZmAIufbPktMfX26tczv5sKb5rx1MJexNUhnxvn/xGDkAvbf8HrU3BpxbqPPX6/
MWGpggb08U8YQU8qLCMzfPaFajTfyyhOKnoh5MBLNVDzbWyqKLzLGQTXl13WSvxQ
ib2U0YoGZtVmshmiVijCu4x1aX4A02sYKrzeYP2WpAl1sEynKz1tkWZQKS3Jt+RP
Y702QtUrRWWul23cBsqy0uKcQ+Zk/rCh5qyQ9mZtx8EmathdnHpi0qZLAH4wEYTA
HScy59zUR5CoYnJNBDNyRyEuh/B/pkTGEisLb+i1F9M0WhrgfgYSldEzUCZ7KJUD
Bt/d50OM+M8kmkYYBmiyWBfNqMMckJHhjFAp8kd0V0FP7lVwPGywK2hGKQBF+ylh
9O3qq1knDkjyCLNIoi7kJkl5KEOq1qLfwRnqfTbivGnhMdWh658WX7Q5QCUZf6vT
+aH2SXA+HNd4VAQguWbnmRNmjW33PTdte25T36qHQ6jlCdyG4VZU8f1yuqzhbHPx
FwAaChb/B6l5SIxK8zupbT3K5NDsSrD0wBV52P377GGVF1z1wI/+0ftNTxu/mgx5
aDjNyxUpupVsqWnws7a/6ixMUqyYAI8TlVWj2vBXgIg9gvt+yp/X7iMD3V4v3z/P
c8o2nH1c9uB6bXJx4HrtenjkDA4uQ89NH35yBII5OviL4NWTmMArWSN5scBOYOMJ
KxGDvFCkgcGjE5x6srj0gzntFg3TcYQiC4+onS485xTQlud53Z7Vc4JnSn7F2j9P
AJX5s+yUgT+ob7SYvYigbp30Oe5fg0NVZzH1fJ3eTJheClb+87vjrZS4X1tVU0MG
LNCdjF20Jff31fZu16mxin+hUZVyfVRlSycfVJU7qXzZftTpr3rsNZiLlY7z7BRe
1lXSx7uZFCfhyu96hyIPxMuaBIX7OHkduinurQ58Vj92fAe/8f5jviRVqtKwZ9Ao
lZfjSn8k/CgDxzFu2gWpbhWJ4GKvoXgmWDESk4o07UGqDbhm0YO5MDM0uoAmsGae
0lQCCQIMTzhupJq1yZ3vljrXBYF37QuWDi/MAvL54hMis5UXrTAh0+FobZAXCoO0
epU7H8CIs4ZCelIyD1W+K/kv3/E3Z65WQzDOYOhXSCAOqcpjiPc=
-----END PGP MESSAGE-----

A.2. V4 PQC Subkey Artifacts

Here is a Private Key consisting of:

* A v4 Ed25519 Private-Key packet

* A User ID packet

* A v4 positive certification self-signature

* A v4 ECDH (Curve25519) Private-Subkey packet

* A v4 subkey binding signature

* A v4 ML-KEM-ipd-768 + X25519 Private-Subkey packet

* A v4 subkey binding signature

The primary key has the fingerprint
b2e9b532d55bd6287ec79e17c62adc0ddd1edd73.

The ECDH subkey has the fingerprint
95bed3c63f295e7b980b6a2b93b3233faf28c9d2.

The ML-KEM-ipd-768 + X25519 subkey has the fingerprint
bd67d98388813e88bf3490f3e440cfbaffd6f357.

-----BEGIN PGP PRIVATE KEY BLOCK-----

xVgEUdDGgBYJKwYBBAHaRw8BAQdAhoSK5cJt9N37EE1UjPqp8EXhAvOBCYikgtcg
HMUso9MAAPwIdkHSrZmM4/Res+3qv1UT7kV5OAr6VO0M2P0ZPdAFiBICzS5QUUMg
dXNlciAoVGVzdCBLZXkpIDxwcWMtdGVzdC1rZXlAZXhhbXBsZS5jb20+wo8EExYK
AEEFAlHQxoAJEMYq3A3dHt1zFiEEsum1MtVb1ih+x54XxircDd0e3XMCGwMCHgkC
GQEDCwkHAxUKCAIWAAUnCQIHAgAAooUA/jV775USotWqnMYHmrqaCWsUduO0cLxS
4U7CuItZnfMJAPwLAyXS8awEJ92Ll52fQ2ESsAkJ4f/cjdHoP9V+BZbSBsddBFHQ
xoASCisGAQQBl1UBBQEBB0Dfrrz6gEv3iM2ULhupwUD4qABPIAwaNyVYDT2euXaS
dgMBCgkAAP9Q+XMh/cX9bvDH6mbpoGjZkeYkw1NO6y5NQEDmvDnEIBN+wngEGBYK
ACoFAlHQxoAJEMYq3A3dHt1zFiEEsum1MtVb1ih+x54XxircDd0e3XMCGwwAAI/D
AP9yG1KzQlWnMNMjyvpkxWhAjyIVxbtr+4WsXUdTqMMQkgD/SeI376LSUoB6s/oL
P10oFOJ86NjwfawQvIqa0CPIkgfHzYkEUdDGgGnWzS/qVrM3Wy7ifldXrJMRIq+r
iGRtWY4Hr1s0GXm+fmMDoLIGUnUCOM0BzzdQgEAcnlVFCZQ4NmlwbChkHI5nFiIl
cGQhrqzzxOzhPJrniyRZJMb3gBMXQO6yCx66G7fHAJ73J1AcFTNWyszaIcXnazHX
OBSpnSMrvQfZIfV3tyW2Xhg6KjhDD6/TsrBiigPGGlZwcPtAh/EbkwR1xYlnU0mX
tlwrlHgWvkwlcXOgdz4VUiDGPIJRGIh6LXe1dobCUjYVZPEKmf9TN2o8oSiVRr8L
GZF1jXyqLlHloMSbJiV1m6iZH8DjTWMBYRRAOVr1Ly7MDqrwJoN0CQFnx/Hqum+2
czgxlsWLtGvADUwaPodwH9MHp4tXJ/HsOOO7z6bYdCNpAySqpSmzCaNzlXppw54n
bD+0UE70dxh0UHiGnoJQXy5mkcG2gTWpwC7bZ+nbCBgcJF/IHBbIWbYQVLDTeP+z
LKnDt/iAoJ5qgeF1wuC7pwsaQy7EUZgClZ0ivkdLyC6ZImikkaczV/VcrxeZZRqC
GqfxQ05QJOAiGFNhvBvDclXcaXYWibxQgyFlGUM1rbR8XZJzVjbihw0pfiVnustU
xYhsroqybX6iJVdAxVNiZwrMZ4VqifErJ1lYbYImF+jKQ6/zYZrrODDmLy0xZqhq
mC5jsE1owzTDzEPnibtTEWiKbShTmJmxbhtQwhW7jsvKhQXbSvr7Nsh0vXzWEGim
IkpBEXycePOnVSens94Rpa2jqgjLJhgalqocm07pNXFMeyJAhYVnHUuCsQzgrkNc
ncbVe85GzsW6S/8MzdtKD9MGy3XHlKKByeF1oxcWEnBEQZ4JhpOmIHV7TVRhHa8L
tIQ4HmCADposq1OTiAbxfYP6RtiLyemxDJaFLdaDSRSXIf5ALgxaysUxe57Qh7uA

-----END PGP PRIVATE KEY BLOCK-----

```
Qh5WejIJy6cDZtUYqtoLg8KDegxKSmo3hy2nsReMgc6SFU/ziHNWWQAtSjHrbFry
ruaAJAmVGKj2UoqACMQlDpZkQYF2po8byQx7TIGnXwmGisygomwjTGocO5LDqoyS
uORISmhcXbvcXtRWnQMafPAhpb6Sfm4JGic7W3/EcgmRcWiLnbnzNeBgirQgqTky
kBRMycBAzgglsq5CJOHWZOoJTvlBHXBiq3z2ddY4hzckCeqQYwCrn08qChsLHuX1
r5ZxFE+XE6+YRvwIYEKrBTDzxNppnZTMFkGhgHWXuZcSnYQAxiSbVHTkjvcEC3k8
HHGovlujZInkNlQGk2KQjCWCI2JgFvIBBcswMt8Jmr9Jpa4zvv08Zi60DJpWYonH
N+uSQ1FbxCm5tM6JJaKSjLYQxm6zfZ0Lxc0XP90SUKg4Ux+Al0y1jH7VgjWmGrP1
geoHgvP8RWlHbW+rhBWsYmAATawUdPZAg/rcODM0fzRpe5CWdnjIhRqUAjQruB4A
n2iXWu5DymzgV6ajOB/3VKxYvup0mRULOwZsqHHIzJGCxlesMIecccRUT2IWaxFJ
h4m1igg8zS9hG5/yQr3bJH2UbxX2o453u58wqvJBYOvhWDsITKAQMyhSD9iGA8Pq
AAs1utxWaATaNH1qvDxDrdiHCYadNeTVxYYb8HVRBLaWNlG1lYjvl+WGf5t9AMC5
EKxgiozRC4yyd1oYV1+fv8g5eMz2pBWB5tuvE5ootGtwzIWSkRmGUfEzZpLCIWAF
/9CWtmnPPiygQZecuzUDsTQRHnIANfWVhGZFmFh8qxc81IZKTPtgBMgW8ewE3oiJ
cac8BHo5cEiTxeVDXXqEMCOn4jQCtKU8+ogQLhF6OvVpV2A9eKlvVberhBqu2+lC
KDt2YpRjb2Bm5lqWDLUAiWa8rMTMwQmfybFp7Zi25pDPHpEwWvGGR6sjVYMfVRR8
HlOV6csYJTejEPghZih0dwEHdhUSe9Su+HNjsiunyNg042mGkOE/n1SGZ2cVkOMB
iwyDm4uo9bG8X9akvOdT4EA3Pfg5DLAOIUILJlsrBPBRvIG5bulUdOtMfkMMUHOF
O3O8FvyjpUc43tOgmvnEcCuuraqIjzokM6pHYjYjySmipMxFi8anwZIix/sUBclg
UtIMo0KBY+aTwDGoOJkERWp8zgdcplfLYYEzlCWhm1JAbabKQpummohwpUErZ9gy
1NtuMJhDxxtb8MMylwWHklpwhkFcLgW/rIkLEte15zuSiGcrOJYpUEpnP/edSdyn
YOutidNbg5tLxaZTiKYcUFcdZ1jI7ows9Ri4v4xJ6MxGJOSIPGWieDw1b9GTehS/
uCp++UJzCMQUYfHI/nYbDAyWPbx9piVkCIychNhrGurIqPMEUBCwXNF60hhGXlep
TLoldts3W0xX3ROmD8gPqEUa8pujI1oeULiL3vlfb1deXSu6evLMPoyWyRKEFCY0
+Fe1G2RX1CyjYKkW5heqWkJixHS7tItCksAgFTTCWcUammBEA8NUuEeg3jO8nVA4
aKFrfOoSrYbHqsmO1AJfCDh21iMUOVAefLTFIsavLuFO8OmOh7cXcOQWG6G5ZdUt
SQQxJJK6mKka5TZlp2GXyGzFVgdD1ddkMHMu9gB96SJjsodDQseklAldfft66xt6
UIbEVbwVCDctzPZ1o/pwKeu2GJa+D6RSEPNRrZeHF7OVq8pnnCQE/HUsrLcqPZhn
/8K5MvhphdmoUQl5fmQZoIRqRxFoDqFvJ/qETBsTwwkkgLwBzSEe4+ubbchq3Jp5
1LVmmZG5BxVe9UWzPirPqKXPu4oArjEqtRJ5MARI1NifzMvKJpGuqhMIh8UQraGf
KNJBxwN99Aq7GCYmkyYvo9wNUMJrYfa3TXh0GwBhqwxObtzAQZzGXGl9kVQEI9Q1
upTFQKgddScJIsoIdzSGfJlkVtSj6lsqmDdkHMPCa9yhk2Ikn9E/kbQ371ca2iZ2
4FAIIXHNeULH4qIc8ScjO1epIuerd9MLJdi5dwR9zAwNe8GznNGMi5HE2BJyS5gZ
8ht3/Xm88lIXRil6bnTOBYQ9RDoIHNU2EFamnBO9jUu92XBGqgRv9iKAVTmPr7i5
qMe8vEU0PeOjt2d4aHlZ/cO+NRO1YvHMQfxSGpjPQMRvUGoUAOOkndYVlGw3VzKw
7pFerytKJaozmpuGFCVLhlJYn9C6oeCfPQQjy1ydULIBe6DKUycbvAIjK3Qj4jum
I2dp8RV3JHRmcxWzngJuj7nFyfeKBecwZesMqXl3YwOgsgZSdQI4zQHPN1CAQBye
VUUJlDg2aXBsKGQcjmcWIiVwZCGurPPE7OE8mueLJFkkxveAExdA7rILHrobt8cA
nvcnUBwVM1bKzNohxedrMdc4FKmdIyu9B9kh9Xe3JbZeGDoqOEMPr9OysGKKA8Ya
VnBw+0CH8RuTBHXFiWdTSZe2XCuUeBa+TCVxc6B3PhVSIMY8glEYiHotd7V2hsJS
NhVk8QqZ/1M3ajyhKJVGvwsZkXWNfKouUeWgxJsmJXWbqJkfwONNYwFhFEA5WvUv
LswOqvAmg3QJAWfH8eq6b7ZzODGWxYu0a8ANTBo+h3Af0weni1cn8ew447vPpth0
I2kDJKqlKbMJo3OVemnDnidsP7RQTvR3GHRQeIaeglBfLmaRwbaBNanALttn6dsI
GBwkX8gcFshZthBUsNN4/7MsqcO3+ICgnmqB4XXC4LunCxpDLsRRmAKVnSK+R0vI
LpkiaKSRpzNX9VyvF5llGoIap/FDTlAk4CIYU2G8G8NyVdxpdhaJvFCDIWUZQzWt
tHxdknNWNuKHDSl+JWe6y1TFiGyuirJtfqIlV0DFU2JnCsxnhWqJ8SsnWVhtgiYX
6MpDr/Nhmus4MOYvLTFmqGqYLmOwTWjDNMPMQ+eJu1MRaIptKFOYmbFuG1DCFbuO
y8qFBdtK+vs2yHS9fNYQaKYiSkERfJx486dVJ6ez3hGlraOqCMsmGBqWqhybTuk1
cUx7IkCFhWcdS4KxDOCuQ1ydxtV7zkbOxbpL/wzN20oP0wbLdceUooHJ4XWjFxYS
```

```
cERBngmGk6YgdXtNVGEdrwu0hDgeYIAOmiyrU5OIBvF9g/pG2IvJ6bEMloUt1oNJ
FJch/kAuDFrKxTF7ntCHu4BCHlZ6MgnLpwNm1Riq2guDwoN6DEpKajeHLaexF4yB
zpIVT/OIc1ZZAC1KMetsWvKu5oAkCZUYqPZSioAIxCUOlmRBgXamjxvJDHtMgadf
CYaKzKCibCNMahw7ksOqjJK45EhKaFxdu9xe1FadAxp88CGlvpJ+bgkaJztbf8Ry
CZFxaIudufM14GCKtCCpOTKQFEzJwEDOCCWyrkIk4dZk6glO+UEdcGKrfPZ11jiH
NyQJ6pBjAKufTyoKGwse5fWvlnEUT5cTr5hG/AhgQqsFMPPE2mmdlMwWQaGAdZe5
lxKdhADGJJtUdOSO9wQLeTwccai+W6NkieQ2VAaTYpCMJYIjYmAW8gEFyzAy3wma
v0mlrjO+/TxmLrQMmlZiicc365JDUVvEKbm0zoklopKMthDGbrN9nQvFzRc/3RJQ
qDhTH4CXTLWMftWCNaYas/WB6geC8/xFaUdtb6uEFaxiYABNrBR09kCD+tw4MzR/
NGl7kJZ2eMiFGpQCNCu4HgCfaJda7kPKbOBXpqM4H/dUrFi+6nSZFQs7BmyoccjM
kYLGV6wwh5xxxFRPYhZrEUmHibWKCDzNL2Ebn/JCvdskfZRvFfajjne7nzCq8kFg
6+FYOwhMoBAzKFIP2IYDw+oACzW63FZoBNo0fWq8PEOt2IcJhp015NXFhhvwdVEE
tpY2UbWViO+X5YZ/m30EFhqD2sbN4HJ/Sv2SB7DadONGI5Sj0tnqRWZ//nA4CLZo
y1LriIK38pV3lBCLv2M9vynHoyXTFco3BqTUGUEjbDnCeAQYFgoAKgUCUdDGgAkQ
xircDd0e3XMWIQSy6bUy1VvWKH7HnhfGKtwN3R7dcwIbDAAA8PEA/16fgmhfrX12
GXFXcTGO8MKQTihxz2djD4aki7fVX+ZAAP9UT/A3jAfqvFNp+ecYkkZ8T+vnXR4P
0O22blDNAr/tDA==
=q5En
-----END PGP PRIVATE KEY BLOCK-----
```

Here is the corresponding Public Key consisting of:

* A v4 Ed25519 Public-Key packet

* A User ID packet

* A v4 positive certification self-signature

* A v4 ECDH (Curve25519) Public-Subkey packet

* A v4 subkey binding signature

* A v4 ML-KEM-ipd-768 + X25519 Public-Subkey packet

* A v4 subkey binding signature

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xjMEUdDGgBYJKwYBBAHaRw8BAQdAhoSK5cJt9N37EE1UjPqp8EXhAvOBCYikgtcg
HMUso9PNLlBRQyB1c2VyIChUZXN0IEtleSkgPHBxYy10ZXN0LWtleUBleGFtcGxl
LmNvbT7CjwQTFgoAQQUCUdDGgAkQxircDd0e3XMWIQSy6bUy1VvWKH7HnhfGKtwN
3R7dcwIbAwIeCQIZAQMLCQcDFQoIAhYABScJAgcCAACihQD+NXvvlRKi1aqcxgea
upoJaxR247RwvFLhTsK4i1md8wkA/AsDJdLxrAQn3YuXnZ9DYRKwCQnh/9yN0eg/
1X4FltIGzjgEUdDGgBIKKwYBBAGXVQEFAQEHQN+uvPqAS/eIzZQuG6nBQPioAE8g
DBo3JVgNPZ65dpJ2AwEKCcJ4BBgWCgAqBQJR0MaACRDGKtwN3R7dcxYhBLLptTLV
W9YofseeF8Yq3A3dHt1zAhsMAACPwwD/chtSs0JVpzDTI8r6ZMVoQI8iFcW7a/uF
rF1HU6jDEJIA/0niN++i0lKAerP6Cz9dKBTifOjY8H2sELyKmtAjyJIHzsQGBFHQ
xoBp1s0v6lazN1su4n5XV6yTESKvq4hkbVmOB69bNBl5vn5jA6CyBlJ1AjjNAc83
UIBAHJ5VRQmUODZpcGwoZByOZxYiJXBkIa6s88Ts4Tya54skWSTG94ATF0Dusgse
uhu3xwCe9ydQHBUzVsrM2iHF52sx1zgUqZ0jK70H2SH1d7cltl4YOio4Qw+v07Kw
YooDxhpWcHD7QIfxG5MEdcWJZ1NJl7ZcK5R4Fr5MJXFzoHc+FVIgxjyCURiIei13
tXaGwlI2FWTxCpn/UzdqPKEolUa/CxmRdY18qi5R5aDEmyYldZuomR/A401jAWEU
QDla9S8uzA6q8CaDdAkBZ8fx6rpvtnM4MZbFi7RrwA1MGj6HcB/TB6eLVyfx7Djj
u8+m2HQjaQMkqqUpswmjc5V6acOeJ2w/tFBO9HcYdFB4hp6CUF8uZpHBtoE1qcAu
22fp2wgYHCRfyBwWyFm2EFSw03j/syypw7f4gKCeaoHhdcLgu6cLGkMuxFGYApWd
Ir5HS8gumSJopJGnM1f1XK8XmWUaghqn8UNOUCTgIhhTYbwbw3JV3Gl2Fom8UIMh
ZRlDNa20fF2Sc1Y24ocNKX4lZ7rLVMWIbK6Ksm1+oiVXQMVTYmcKzGeFaonxKydZ
WG2CJhfoykOv82Ga6zgw5i8tMWaoapguY7BNaMM0w8xD54m7UxFoim0oU5iZsW4b
UMIVu47LyoUF20r6+zbIdL181hBopiJKQRF8nHjzp1Unp7PeEaWto6oIyyYYGpaq
HJtO6TVxTHsiQIWFZx1LgrEM4K5DXJ3G1XvORs7Fukv/DM3bSg/TBst1x5Sigcnh
daMXFhJwREGeCYaTpiB1e01UYR2vC7SEOB5ggA6aLKtTk4gG8X2D+kbYi8npsQyW
hS3Wg0kUlyH+QC4MWsrFMXue0Ie7gEIeVnoyCcunA2bVGKraC4PCg3oMSkpqN4ct
p7EXjIHOkhVP84hzVlkALUox62xa8q7mgCQJlRio9lKKgAjEJQ6WZEGBdqaPG8kM
e0yBp18JhorMoKJsI0xqHDuSw6qMkrjkSEpoXF273F7UVp0DGnzwIaW+kn5uCRon
O1t/xHIJkXFoi5258zXgYIq0IKk5MpAUTMnAQM4IJbKuQiTh1mTqCU75QR1wYqt8
9nXWOIc3JAnqkGMAq59PKgobCx7l9a+WcRRPlxOvmEb8CGBCqwUw88TaaZ2UzBZB
oYB1l7mXEp2EAMYkm1R05I73BAt5PBxxqL5bo2SJ5DZUBpNikIwlgiNiYBbyAQXL
MDLfCZq/SaWuM779PGYutAyaVmKJxzfrkkNRW8QpubTOiSWikoy2EMZus32dC8XN
Fz/dElCoOFMfgJdMtYx+1YI1phqz9YHqB4Lz/EVpR21vq4QVrGJgAE2sFHT2QIP6
3DgzNH80aXuQlnZ4yIUalAI0K7geAJ9ol1ruQ8ps4Femozgf91SsWL7qdJkVCzsG
bKhxyMyRgsZXrDCHnHHEVE9iFmsRSYeJtYoIPM0vYRuf8kK92yR9lG8V9qOOd7uf
MKryQWDr4Vg7CEygEDMoUg/YhgPD6gALNbrcVmgE2jR9arw8Q63YhwmGnTXk1cWG
G/B1UQS2ljZRtZWI75flhn+bfcJ4BBgWCgAqBQJR0MaACRDGKtwN3R7dcxYhBLLp
tTLVW9YofseeF8Yq3A3dHt1zAhsMAADw8QD/Xp+CaF+tfXYZcVdxMY7wwpBOKHHP
Z2MPhqSLt9Vf5kAA/1RP8DeMB+q8U2n55xiSRnxP6+ddHg/Q7bZuUM0Cv+0M
=dPFW
-----END PGP PUBLIC KEY BLOCK-----
```

Here is an SEIPDv1 unsigned message "Testing\n" encrypted to this key:

* A v3 PKESK

* A v1 SEIPD

The hex-encoded KMAC eccKeyShare input is
ba6634c5bab5756868dac8282054b0b30916d764e1f15841222392e5545a67c7.

The hex-encoded KMAC mlkemKeyShare input is
a6b263da0e367b39c2d44bf4c3f66015f410ee4fa674ddbba8d50cde2fc4094a.

The hex-encoded KMAC256 output is
504bc329627af248947117936bee9e87230d327d5c5f5b4db593c4b58b2d0339.

The hex-encoded session key is
b639d5feaae6c8eabcf04182322d576298193cfa9555d869cf911ffbbc5e52e7.

```
-----BEGIN PGP MESSAGE-----

wcPUA+RAz7r/1vNXafOEf/qo/UCjgP5MHXsUdzusv+Xtwa/q5/gtvWSKENMEm/ef
mwvHzlOW3jyHPr3wNkiRGNmZdEgHlRUxOF67AKfUqX1/H+RawlXrnu2O/7xDfn09
1DOeYQYsG+VQe5eDfPOh6EAqvhglNG29enkwukjQuWJ0U2+SywRlh0J49oG0lZZi
tEonE6BX+dKo7JFAAW27I4+I15zamWCm7C2qXtcjlMq+68U6isstfJYBgCFoJLRV
9ME/JlwVQ/3OA4eagh1Ysw/s0Kbl8dWc/U+pUVINfLFnZbr8UnRx5QVjo47HzV9y
IPQcl1ETnrQqg4jH5cUB0wVB/OGSHEH1l1q7OwO4TBx9seRKza3wzHgXQyQn4jJn
WxG4Uf0rWwa/dbXpoGdxye7Di0HhFJBp4aPaPEm9RwmeD06HPGyurgQS2heW+ICR
X9q6HjxKmbIToWau7sEUZQh1isRD923zmZO1Cceiefwvv8RDErBaYRRfhoDHnwlf
haUkGDH55GC/mFgMUst7XjYLTpjTtM32bHHVoYyx3edk3V4C32dRP4geZRlq8MKb
3eNvTyZHPBXVeVjB2XVenMkju2qQvjAvr7xkJk5QvqRGZ3qy5JKPHjupPKMKR6Wv
90WtP9OPf6geE4K8g8dc4yqhqEf84IpWfwyUjglPwc8G7QURyc14INSgnVKQ8nzJ
26Y0YSuBG599cwnBXIv7TLOsrmpcR+3O/OiqztEXwBSFalWqC6SScWr51+K4RHpi
0Dg3GYOdeu9V12ofWhh5eoXiQxiThsDc0cYD2LwwITb/sXJW6jisvTSQSszuDJsg
w4LJGVIgk94GwBeYCa4w+YbFD/6bC+nd9hze5iWyAt4jx2XuYWHvI95wUvRal4Y6
LcYKShZslef/9sZlrJ3O+oM8Tc+xjr7w860cw9bHBaRA3oZfDwVLRbJ63bh6dzLS
8Vj4UL0NiK9AlokOTlYTatKj2tkLaD6snZ5QaS916NniwvBiSozyCWKFPvKbIA5t
Fy7bWMk70wd7FJ5rOYsRvWOodcxxfYHLyOYCRikP5sOgelHJfIcqZZM/iMyKrUl5
QXTe5SUMFKuAAWkv1nFgJqaBGkkYUN+aL6um4ogKJKxBArhUXtz5jp1zLQe6w7Pf
XtJ3rtnLMNIGbaJjN6jlSqB1lF88ljU6yL8lsfNfGuq6eLsNz1QD9sFoZyNKGVF8
uGcX8KH07hyNCOPbn4jMZqnLfcEwT971KNeW0NNNLaVIjiYtYrEInYpnGP1fW0G7
4ic4/CGZ0Jsti39rTPdz76n14EV5+HGGNtO92nXr7HhRvlvehZO+pPN2pPvAOQgp
378jEDnhdcFIUnWjaSN4HLjxN9X3tWosa9stVatcOB5n01+QAqdstgbARXiYhzZI
QOy5bk7GzM/xaD+4iMlsMgmhdvI3EuMrvzRHx95YvU+tc8UBhVtMpcxgp5tiPtw7
wEFyPWYqTA6c+YGTZFzHNhqJ+DqLTlKP6+uYet4pCY6YLcu4z0JMG60Q4/MnuF2u
7pjPG7aRNGwr2/JFeaumSFbs5SrDPwDSOQGz9bQ6fbfGNm9Iw/Gq4EYrNx0raEVt
E1FD8+nRXcS1PribkWlu/qNd0yM+tXMXlryR6wAV3R4p9Q==
=h3Km
-----END PGP MESSAGE-----
```

Here is an SEIPDv2 unsigned message testing encrypted to this key:

* A v6 PKESK

* A v2 SEIPD

The hex-encoded KMAC eccKeyShare input is
50a74bfb94dc7677bc02f278eb4e7d5d2f1b04e34a2b5c7b8da0579f3e1e0825.

The hex-encoded KMAC mlkemKeyShare input is
161911216c93a5b7936f9a8876c446b0767c904c94786bfc79bcc505b45f5075.

The hex-encoded KMAC256 output is
ee4dacbc4efac509ad5f79640d5963af038baf512d55974c46ac71db6c1ed579.

The hex-encoded session key is
27e3c564fa7b8adb7ee1cfede3ee2cda79dd8f1a6d029ebeb7f3880c752185f6.

-----BEGIN PGP MESSAGE-----

wcPhBhUEvWfZg4iBPoi/NJDz5EDPuv/W81dp/h3ny6MDFKIjw8Jva/pK0mlJJMbX
RtJ2+idWKUL4/Evq2JBL2pky3VeCkhJIAnKdMXBvuucpNC4xVhCREpPquOsQcJE1
IyR5kdk0uOw/7T5i7i9zb3N8Mn4jObYNxem0PbD44xYfF4BcNZqWPhLSwdFp+uti
FDVcsAtaruQQnwcb0tE9KUxMHsH2QxuG6Xnm+a1v3fds/rp/DqrR1vTcwbeUBej9
AcrWhRI+KrE80WtMqqB0nvj74Jsx7xrYKyDqi7C50PG1LwTfFmO1kuIOGwEVh3AL
f9vesNee8+QWqJHMuSaqTsndts7Pq2EQ1fzdsrhYvseFEuiITj6OHnvmERjhkVRt
M36v3cA8GmcKI2YLa+fNvyagjgkoUycjNopK94/KQk8DY4JnKEbMHJ1BvUQ6mXkZ
ZeQXEt9gBHMZvTPE4ZPKSR14tki9YbjCvGcklEx1BEhONMo+03C07H/AHMtk5Ia1
BFDF3dRkOhyXLfJCNKhJL4Sq7Z07CqKYCCzEq6WgKQUlWmiJzEtZKmh61LX3FaVU
A8+iK56QBvYyFUv/T9mGIBvF6qM+1l9BUSDim81EEN3FVKuo5QJ+tEbuKOrDWMTQ
klx3iEmGFeW8/N0gxTMpDKQRqDbkhdwFoM1c1L61NeCfDvTiWWVQQg00Z/n38gGC
S4MC8szKWPPlmtPYsXbyITMLpErgbIfr3IPfwRkm1RWVPQ/T7dm0Iz8ufeVTAqv+
4+WgeILpQbFf/PmKHlZJnyj+munVaLaV9ed8g/0ncfZ0Vn9sMzGXifAjJ7vT8t/2
ojSqc1AQuhoNguI3xFUCaUEzmWzOl3ONZhgDr4WTJdvfW/8IHn2y0Kkn5buqLAO5
LOvrksFgkuy3N21OzgI+H5SeCqOGt5NKgyMAUGKEa5a6ApG/DSecXJ38yr4hTG6k
IGxfd9dlezcdByrohFsRHC93XUSlTnxTqdYmuhK8x4me6QblJFEJhoDTr3lrKTQE
gXC5ncdTLKLLINnZgFRc0oFABKGATf/708nt/vkuhyNroCY2NxGKoem3M8P5radp
vKJ7qRi+ePBmxwPhb5mWVf7G5y509GgVhKUFUWCzvyoyp1HLtekARKrxBAElXo4D
RINA468O/Sx9rWDFFkaShsXqQEhe06C3IhoG6vqagdB82LYcisCWEAKd/hZE99Ub
EWawnQor/jrHhGNQId1nZEwpEE2/cndU3Np0mhoGF/kkjyGah8wxjFRZhX2b+w3M
jmyFyOFGog6SDK1dK6+Vdki3fJFvBCx/uLmuLiNQo2MYeG3B4W30jNDlkxZ2DA14
1zPXL8S3t54ZCtqiqDO63SNW18o2SB9AJpmSYYF34LO29VJnWIK5/94tIsfwHbAg
P/JdfRmSXkUYDTwJcYR1SOp8IIWwhmybnYctHS/n1q+WnzMj0jJIe4SI9ifoJg6n
+ntRfsqR1B3fcTAKN0XK6pnNy+pJ1BCVDOYDz5RGsAa61wJdVK/mXEg7VyjMlf0t
EnPzSfwfotMPoZs9n+MuK/BO0AX8DqzY8SluBALG6Eu5OV+A0lQCCQIMeNuKCuzH
CoYERR8ds1jE9m+xLEGK7i1+zr5FSdzGowHh4xMo95Zk5JUubl5rYvcYbHgVVKKE
9mwM7/4Q5mXZ2xvsBftkujgamZM2UN9UYSA=
-----END PGP MESSAGE-----

Acknowledgments

Thanks to Daniel Huigens and Evangelos Karatsiolis for the early review and feedback on this document.

Authors' Addresses

Stavros Kousidis
BSI
Germany
Email: stavros.kousidis@bsi.bund.de

Johannes Roth
MTG AG
Germany
Email: johannes.roth@mtg.de

Falko Strenzke
MTG AG
Germany
Email: falko.strenzke@mtg.de

Aron Wussler
Proton AG
Switzerland
Email: aron@wussler.it