

HPKE  
Internet-Draft  
Obsoletes: 9180 (if approved)  
Intended status: Standards Track  
Expires: 3 September 2026

R. Barnes  
Cisco  
K. Bhargavan  
B. Lipp  
Inria  
C. Wood  
2 March 2026

## Hybrid Public Key Encryption draft-ietf-hpke-hpke-03

### Abstract

This document describes a scheme for hybrid public key encryption (HPKE). This scheme provides a variant of public key encryption of arbitrary-sized plaintexts for a recipient public key. It also includes a variant that authenticates possession of a pre-shared key. HPKE works for any combination of an asymmetric Key Encapsulation Mechanism (KEM), key derivation function (KDF), and authenticated encryption with additional data (AEAD) encryption function. We provide instantiations of the scheme using widely used and efficient primitives, such as Elliptic Curve Diffie-Hellman (ECDH) key agreement, HMAC-based key derivation function (HKDF), and SHA2.

### Discussion Venues

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

Source for this draft and an issue tracker can be found at <https://github.com/hpkewg/hpke>.

### Status of This Memo

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

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

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

This Internet-Draft will expire on 3 September 2026.

## Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

## Table of Contents

1. Introduction . . . . .	4
2. Requirements Notation . . . . .	5
3. Notation . . . . .	5
4. Cryptographic Dependencies . . . . .	6
4.1. Key Encapsulation Mechanisms . . . . .	6
4.2. Key Derivation Functions . . . . .	7
4.3. AEAD Encryption Algorithm . . . . .	8
4.4. Labeled Derivation Functions . . . . .	8
4.5. DH-Based KEM (DHKEM) . . . . .	9
5. Hybrid Public Key Encryption . . . . .	11
5.1. Creating the Encryption Context . . . . .	12
5.1.1. Encryption to a Public Key . . . . .	15
5.1.2. Authentication Using a Pre-Shared Key . . . . .	16
5.2. Encryption and Decryption . . . . .	16
5.3. Secret Export . . . . .	18
6. Single-Shot APIs . . . . .	19
6.1. Encryption and Decryption . . . . .	19
6.2. Secret Export . . . . .	20
7. Algorithm Identifiers . . . . .	20
7.1. Key Encapsulation Mechanisms (KEMs) . . . . .	21
7.1.1. SerializePublicKey and DeserializePublicKey . . . . .	21
7.1.2. SerializePrivateKey and DeserializePrivateKey . . . . .	22
7.1.3. DeriveKeyPair . . . . .	22
7.1.4. Validation of Inputs and Outputs . . . . .	24
7.1.5. Future KEMs . . . . .	24
7.2. Key Derivation Functions (KDFs) . . . . .	25
7.2.1. Input Length Restrictions . . . . .	25
7.3. Authenticated Encryption with Associated Data (AEAD) Functions . . . . .	27
8. API Considerations . . . . .	27
8.1. Auxiliary Authenticated Application Information . . . . .	27
8.2. Errors . . . . .	28

9.	Security Considerations . . . . .	29
9.1.	Security Properties . . . . .	29
9.1.1.	Computational Analysis . . . . .	30
9.1.2.	Post-Quantum Security . . . . .	31
9.2.	Security Requirements on a KEM Used within HPKE . . . . .	32
9.2.1.	Encap/Decap Interface . . . . .	32
9.2.2.	KEM Key Reuse . . . . .	32
9.3.	Security Requirements on a KDF . . . . .	32
9.4.	Security Requirements on an AEAD . . . . .	32
9.5.	Pre-Shared Key Recommendations . . . . .	32
9.6.	Domain Separation . . . . .	33
9.7.	Application Embedding and Non-Goals . . . . .	34
9.7.1.	Message Order and Message Loss . . . . .	34
9.7.2.	Downgrade Prevention . . . . .	35
9.7.3.	Replay Protection . . . . .	35
9.7.4.	Forward Secrecy . . . . .	36
9.7.5.	Bad Ephemeral Randomness . . . . .	37
9.7.6.	Hiding Plaintext Length . . . . .	37
9.8.	Bidirectional Encryption . . . . .	37
9.9.	Metadata Protection . . . . .	38
10.	Message Encoding . . . . .	38
11.	IANA Considerations . . . . .	39
11.1.	KEM Identifiers . . . . .	39
11.2.	KDF Identifiers . . . . .	40
11.3.	AEAD Identifiers . . . . .	40
12.	References . . . . .	41
12.1.	Normative References . . . . .	41
12.2.	Informative References . . . . .	41
Appendix A.	Differences from RFC 9180 . . . . .	45
Appendix B.	Acknowledgements . . . . .	45
Appendix C.	Test Vectors . . . . .	46
C.1.	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM . . . . .	46
C.1.1.	Base Setup Information . . . . .	46
C.1.2.	PSK Setup Information . . . . .	49
C.2.	DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305 . . . . .	52
C.2.1.	Base Setup Information . . . . .	52
C.2.2.	PSK Setup Information . . . . .	55
C.3.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, AES-128-GCM . . . . .	58
C.3.1.	Base Setup Information . . . . .	58
C.3.2.	PSK Setup Information . . . . .	61
C.4.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA512, AES-128-GCM . . . . .	64
C.4.1.	Base Setup Information . . . . .	64
C.4.2.	PSK Setup Information . . . . .	67
C.5.	DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305 . . . . .	70
C.5.1.	Base Setup Information . . . . .	70
C.5.2.	PSK Setup Information . . . . .	73

C.6. DHKEM(P-521, HKDF-SHA512), HKDF-SHA512, AES-256-GCM . . .	76
C.6.1. Base Setup Information . . . . .	76
C.6.2. PSK Setup Information . . . . .	79
C.7. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, Export-Only	
AEAD . . . . .	82
C.7.1. Base Setup Information . . . . .	82
C.7.2. PSK Setup Information . . . . .	84
Authors' Addresses . . . . .	85

## 1. Introduction

Encryption schemes that combine asymmetric and symmetric algorithms have been specified and practiced since the early days of public key cryptography, e.g., [RFC1421]. Combining the two yields the key management advantages of asymmetric cryptography and the performance benefits of symmetric cryptography. The traditional combination has been "encrypt the symmetric key with the public key." "Hybrid" public key encryption (HPKE) schemes, specified here, take a different approach: "generate the symmetric key and its encapsulation with the public key." Specifically, encrypted messages convey a shared secret encapsulated with a public key scheme, along with one or more arbitrary-sized ciphertexts encrypted using that key. This type of public key encryption has many applications in practice, including Messaging Layer Security [RFC9420], TLS Encrypted ClientHello [I-D.ietf-tls-esni], and Oblivious HTTP [RFC9458].

Currently, there are numerous competing and non-interoperable standards and variants for hybrid encryption, mostly variants on the Elliptic Curve Integrated Encryption Scheme (ECIES), including ANSI X9.63 (ECIES) [ANSI], IEEE 1363a [IEEE1363], ISO/IEC 18033-2 [ISO], and SECG SEC 1 [SECG]. See [MAEA10] for a thorough comparison. All these existing schemes have problems, e.g., because they rely on outdated primitives, lack proofs of indistinguishable (adaptive) chosen-ciphertext attack (IND-CCA2) security, or fail to provide test vectors.

This document defines an HPKE scheme that provides a subset of the functions provided by the collection of schemes above but specified with sufficient clarity that they can be interoperably implemented. The HPKE construction defined herein is secure against (adaptive) chosen ciphertext attacks (IND-CCA2-secure) under classical assumptions about the underlying primitives [HPKEAnalysis] [ABHKLR20]. A summary of these analyses is in Section 9.1.

## 2. Requirements Notation

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. Notation

The following terms are used throughout this document to describe the operations, roles, and behaviors of HPKE:

- \* (skX, pkX): A key encapsulation mechanism (KEM) key pair used in role X, where X is one of S, R, or E as sender, recipient, and ephemeral, respectively; skX is the private key and pkX is the public key.
- \* pk(skX): The KEM public key corresponding to the KEM private key skX.
- \* Sender (S): Role of entity that sends an encrypted message.
- \* Recipient (R): Role of entity that receives an encrypted message.
- \* Ephemeral (E): Role of a fresh random value meant for one-time use.
- \* I2OSP(n, w): Convert non-negative integer n to a w-length, big-endian byte string, as described in [RFC8017].
- \* OS2IP(x): Convert byte string x to a non-negative integer, as described in [RFC8017], assuming big-endian byte order.
- \* concat(x0, ..., xN): Concatenation of byte strings. concat(0x01, 0x0203, 0x040506) = 0x010203040506.
- \* lengthPrefixed(x): The two-byte length of the byte string x, concatenated with x itself. (lengthPrefixed(x) = concat(I2OSP(len(x), 2), x)) It is an error to call this function with an x value that is more than 65535 bytes long.
- \* random(n): A pseudorandom byte string of length n bytes
- \* xor(a,b): XOR of byte strings; xor(0xF0F0, 0x1234) = 0xE2C4. It is an error to call this function with two arguments of unequal length.

## 4. Cryptographic Dependencies

HPKE variants rely on the following primitives:

- \* A Key Encapsulation Mechanism (KEM); see Section 4.1
- \* A Key Derivation Function (KDF); see Section 4.2
- \* An Authenticated Encryption with Associated Data (AEAD); see Section 4.3

A `_ciphersuite_` is a triple (KEM, KDF, AEAD) containing a choice of algorithm for each primitive.

A set of algorithm identifiers for concrete instantiations of these primitives is provided in Section 7. Algorithm identifier values are two bytes long.

### 4.1. Key Encapsulation Mechanisms

A key encapsulation mechanism (KEM) provides the following functions:

`GenerateKeyPair()`: Randomized algorithm to generate a key pair (skX, pkX).

`DeriveKeyPair(ikm)`: Deterministic algorithm to derive a key pair (skX, pkX) from the byte string ikm, where ikm is an arbitrary-length byte string (within the bounds in Table 4). The ikm input SHOULD have at least Nsk bytes of entropy.

`SerializePublicKey(pkX)`: Produce a byte string of length Npk encoding the public key pkX.

`DeserializePublicKey(pkXm)`: Parse a byte string of length Npk to recover a public key. This function can raise a `DeserializeError` error upon pkXm deserialization failure.

`Encap(pkR)`: Randomized algorithm to generate an ephemeral, fixed-length shared secret and a fixed-length encapsulation of that secret (also known as the KEM ciphertext) that can be decapsulated by the holder of the private key corresponding to pkR. This function can raise an `EncapError` on encapsulation failure.

`Decap(enc, skR)`: Deterministic algorithm using the private key skR to recover the shared secret) from the encapsulated secret enc. This function can raise a `DecapError` on decapsulation failure.

The notation  $\text{pk}(\text{skX})$ , depending on its use and the KEM and its implementation, is either the computation of the public key using the private key, or just syntax expressing the retrieval of the public key, assuming it is stored along with the private key object.

Each KEM is parameterized by the following constants (all measured in bytes):

Nsecret: The length of a KEM shared secret produced by this KEM.

Nenc: The length of an encapsulated secret produced by this KEM.

Npk: The length of an encoded public key for this KEM.

Nsk: The length of an encoded private key for this KEM.

Beyond the above, a KEM MAY also expose the following functions, whose behavior is detailed in Section 7.1.2:

SerializePrivateKey(skX): Produce a byte string of length Nsk encoding the private key skX.

DeserializePrivateKey(skXm): Parse a byte string of length Nsk to recover a private key. This function can raise a DeserializeError error upon skXm deserialization failure.

Senders and recipients MUST validate KEM inputs and outputs as described in Section 7.1.

#### 4.2. Key Derivation Functions

A key derivation function (KDF) is either a one-stage KDF, with a single Derive function, or a two-stage KDF, with Extract and Expand functions.

A one-stage KDF provides the function:

Derive(ikm, L): Derive an L-byte value from the input keying material ikm.

A two-stage KDF provides the functions:

Extract(salt, ikm): Extract a pseudorandom key of fixed length  $N_h$  bytes from input keying material ikm and an optional byte string salt.

Expand(prk, info, L): Expand a pseudorandom key prk using optional string info into L bytes of output keying material.

The  $N_h$  parameter indicates security strength of KDF, in bytes. For a two-stage KDF,  $N_h$  is the output size of the Extract() function.

Certain functions have a different structure depending on whether a one-stage or two-stage KDF is being used. For clarity, such functions will be described twice in this document, once with the suffix `_OneStage` and once with the suffix `_TwoStage`, representing the versions of the function to be used with a one-stage or two-stage KDF, respectively. For example, the `Foo` function would be invoked by calling `Foo_OneStage` when using a one-stage KDF, and by calling `Foo_TwoStage` when using a two-stage KDF.

#### 4.3. AEAD Encryption Algorithm

An AEAD encryption algorithm [RFC5116] provides the functions:

`Seal(key, nonce, aad, pt)`: Encrypt and authenticate plaintext `pt` with associated data `aad` using symmetric key `key` and nonce `nonce`, yielding ciphertext and tag `ct`. This function can raise a `MessageLimitReachedError` upon failure.

`Open(key, nonce, aad, ct)`: Decrypt ciphertext and tag `ct` using associated data `aad` with symmetric key `key` and nonce `nonce`, returning plaintext message `pt`. This function can raise an `OpenError` or `MessageLimitReachedError` upon failure.

AEAD functions have the following parameters (all measured in bytes):

`Nk`: The length a key for this algorithm.

`Nn`: The length of a nonce for this algorithm.

`Nt`: The length of the authentication tag for this algorithm.

#### 4.4. Labeled Derivation Functions

The following functions are defined to facilitate domain separation of KDF calls as well as context binding:

```
# For use with one-stage KDFs
def LabeledDerive(ikm, label, context, L):
    labeled_ikm = concat(
        ikm,
        "HPKE-v1",
        suite_id,
        lengthPrefixed(label),
        I2OSP(L, 2),
        context,
    )
    return Derive(labeled_ikm, L)
```

```
# For use with two-stage KDFs
def LabeledExtract(salt, label, ikm):
    labeled_ikm = concat("HPKE-v1", suite_id, label, ikm)
    return Extract(salt, labeled_ikm)

def LabeledExpand(prk, label, info, L):
    labeled_info = concat(I2OSP(L, 2), "HPKE-v1", suite_id,
                          label, info)
    return Expand(prk, labeled_info, L)
```

The value of `suite_id` depends on where the KDF is used; it is assumed implicit from the implementation and not passed as a parameter. If used inside a KEM algorithm, `suite_id` MUST start with "KEM" and identify this KEM algorithm; if used in the remainder of HPKE, it MUST start with "HPKE" and identify the entire ciphersuite in use. See sections Section 4.5 and Section 5.1 for details.

#### 4.5. DH-Based KEM (DHKEM)

Suppose we are given a KDF, and a Diffie-Hellman (DH) group providing the following operations:

`DH(skX, pkY)`: Perform a non-interactive Diffie-Hellman exchange using the private key `skX` and public key `pkY` to produce a Diffie-Hellman shared secret of length `Ndh`. This function can raise a `ValidationError` as described in Section 7.1.4.

A DH-based KEM is parameterized by the constants (in bytes):

`Ndh`: The length of the shared secret produced by `DH()`.  
`Nsk`: The length of a Diffie-Hellman private key.

Then we can construct a KEM that implements the interface defined in Section 4.1 called `DHKEM(Group, KDF)` in the following way, where `Group` denotes the Diffie-Hellman group and `KDF` denotes the KDF. The function parameters `pkR` and `pkS` are deserialized public keys, and `enc` is a serialized public key. Since encapsulated shared secrets are Diffie-Hellman public keys in this KEM algorithm, we use `SerializePublicKey()` and `DeserializePublicKey()` to encode and decode them, respectively. `Npk` and `Nenc` are both equal to `Ndh`. `GenerateKeyPair()` produces a key pair for the Diffie-Hellman group in use. Section 7.1.3 contains the `DeriveKeyPair()` function specification for DHKEMs defined in this document.

```
# For use with one-stage KDFs
def ExtractAndExpand_OneStage(dh, kem_context):
    return LabeledDerive(dh, "shared_secret", kem_context, Nsecret)

# For use with two-stage KDFs
def ExtractAndExpand_TwoStage(dh, kem_context):
    eae_prk = LabeledExtract("", "eae_prk", dh)
    shared_secret = LabeledExpand(eae_prk, "shared_secret",
                                  kem_context, Nsecret)

    return shared_secret

def Encap(pkR):
    skE, pkE = GenerateKeyPair()
    dh = DH(skE, pkR)
    enc = SerializePublicKey(pkE)

    pkRm = SerializePublicKey(pkR)
    kem_context = concat(enc, pkRm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret, enc

def Decap(enc, skR):
    pkE = DeserializePublicKey(enc)
    dh = DH(skR, pkE)

    pkRm = SerializePublicKey(pk(skR))
    kem_context = concat(enc, pkRm)

    shared_secret = ExtractAndExpand(dh, kem_context)
    return shared_secret
```

The implicit `suite_id` value used within `LabeledExtract`, `LabeledExpand`, and `LabeledDerive` is defined as follows, where `kem_id` is defined in Section 7.1:

```
suite_id = concat("KEM", I2OSP(kem_id, 2))
```

The KDF used in DHKEM can be equal to or different from the KDF used in the remainder of HPKE, depending on the chosen variant. Implementations MUST make sure to use the constants (`Nh`) and function calls (`LabeledExtract`, `LabeledExpand`, and `LabeledDerive`) of the appropriate KDF when implementing DHKEM. See Section 9.3 for a comment on the choice of a KDF for the remainder of HPKE, and Section 9.6 for the rationale of the labels.

For the variants of DHKEM defined in this document, the size `Nsecret` of the KEM shared secret is equal to the output length of the hash function underlying the KDF. For P-256, P-384, and P-521, the size `Ndh` of the Diffie-Hellman shared secret is equal to 32, 48, and 66, respectively, corresponding to the x-coordinate of the resulting elliptic curve point [IEEE1363]. For X25519 and X448, the size `Ndh` is equal to 32 and 56, respectively (see Section 5 of [RFC7748]).

## 5. Hybrid Public Key Encryption

In this section, we define a few HPKE variants. All variants take a recipient public key and a sequence of plaintexts `pt` and produce an encapsulated secret `enc` and a sequence of ciphertexts `ct`. These outputs are constructed so that only the holder of `skR` can decapsulate the key from `enc` and decrypt the ciphertexts. All the algorithms also take an `info` parameter that can be used to influence the generation of keys (e.g., to fold in identity information) and an `aad` parameter that provides additional authenticated data to the AEAD algorithm in use.

In addition to the base case of encrypting to a public key, we include a variant that authenticates possession of a pre-shared key. The authenticated variant contributes additional keying material to the encryption operation. The following one-byte values will be used to distinguish between modes:

Mode	Value
mode_base	0x00
mode_psk	0x01
RESERVED	0x02
RESERVED	0x03

Table 1: HPKE Modes

(The values 0x02 and 0x03 were used in [RFC9180] to reflect additional variants which have been removed from this specification.)

Both variants follow the same basic two-step pattern:

1. Set up an encryption context that is shared between the sender and the recipient.

## 2. Use that context to encrypt or decrypt content.

A `_context_` is an implementation-specific structure that encodes the AEAD algorithm and key in use, and manages the nonces used so that the same nonce is not used with multiple plaintexts. It also has an interface for exporting secret values, as described in Section 5.3. See Section 5.2 for a description of this structure and its interfaces. HPKE decryption fails when the underlying AEAD decryption fails.

The constructions described here presume that the relevant non-private parameters (`enc`, `psk_id`, etc.) are transported between the sender and the recipient by some application making use of HPKE. Moreover, a recipient with more than one public key needs some way of determining which of its public keys was used for the encapsulation operation. As an example, applications may send this information alongside a ciphertext from the sender to the recipient. Specification of such a mechanism is left to the application. See Section 10 for more details.

The procedures described in this section are laid out in a Python-like pseudocode. The algorithms in use are left implicit.

### 5.1. Creating the Encryption Context

The variants of HPKE defined in this document share a common key schedule that translates the protocol inputs into an encryption context. The key schedule inputs are as follows:

- \* `mode` - A one-byte value indicating the HPKE mode, defined in Table 1.
- \* `shared_secret` - A KEM shared secret generated for this transaction.
- \* `info` - Application-supplied information (optional; default value "").
- \* `psk` - A pre-shared key (PSK) held by both the sender and the recipient (optional; default value "").
- \* `psk_id` - An identifier for the PSK (optional; default value "").

Senders and recipients MUST validate KEM inputs and outputs as described in Section 7.1.

The `info` parameter used by HPKE is not related to the optional string `info` used by the `LabeledExpand()` or `Expand()` functions detailed in Section 4.

The `psk` and `psk_id` parameters **MUST** appear together or not at all. That is, if a non-default value is provided for one of them, then the other **MUST** be set to a non-default value. This requirement is encoded in `VerifyPSKInputs()` below.

The `psk`, `psk_id`, and `info` parameters have maximum lengths that depend on the KDF itself, on the definition of `LabeledExtract()`, and on the constant labels used together with them. See Section 7.2.1 for precise limits on these lengths.

The `key`, `base_nonce`, and `exporter_secret` computed by the key schedule have the property that they are only known to the holder of the recipient private key, and the entity that used the KEM to generate `shared_secret` and `enc`.

The HPKE algorithm identifiers, i.e., the KEM `kem_id`, KDF `kdf_id`, and AEAD `aead_id` 2-byte code points, as defined in Table 2, Table 3, and Table 5, respectively, are assumed implicit from the implementation and not passed as parameters. The implicit `suite_id` value used within `LabeledExtract`, `LabeledExpand`, and `LabeledDerive` is defined based on them as follows:

```
suite_id = concat(
    "HPKE",
    I2OSP(kem_id, 2),
    I2OSP(kdf_id, 2),
    I2OSP(aead_id, 2)
)

default_psk = ""
default_psk_id = ""

def VerifyPSKInputs(mode, psk, psk_id):
    got_psk = (psk != default_psk)
    got_psk_id = (psk_id != default_psk_id)
    if got_psk != got_psk_id:
        raise Exception("Inconsistent PSK inputs")

    if got_psk and mode == mode_base:
        raise Exception("PSK input provided when not needed")
    if (not got_psk) and mode == mode_psk:
        raise Exception("Missing required PSK input")

# For use with a one-stage KDF
```

```
def CombineSecrets_OneStage(mode, shared_secret, info, psk, psk_id):
    secrets = concat(
        lengthPrefixed(psk),
        lengthPrefixed(shared_secret),
    )
    context = concat(
        mode,
        lengthPrefixed(psk_id),
        lengthPrefixed(info),
    )

    secret = LabeledDerive(secrets, "secret", context, Nk + Nn + Nh)

    key = secret[:Nk]
    base_nonce = secret[Nk:(Nk + Nn)]
    exporter_secret = secret[(Nk + Nn):]

    return (key, base_nonce, exporter_secret)

# For use with a two-stage KDF
def CombineSecrets_TwoStage(mode, shared_secret, info, psk, psk_id):
    psk_id_hash = LabeledExtract("", "psk_id_hash", psk_id)
    info_hash = LabeledExtract("", "info_hash", info)
    key_schedule_context = concat(mode, psk_id_hash, info_hash)

    secret = LabeledExtract(shared_secret, "secret", psk)

    key = LabeledExpand(secret, "key", key_schedule_context, Nk)
    base_nonce = LabeledExpand(secret, "base_nonce",
                                key_schedule_context, Nn)
    exporter_secret = LabeledExpand(secret, "exp",
                                    key_schedule_context, Nh)

    return (key, base_nonce, exporter_secret)

def KeySchedule<ROLE>(mode, shared_secret, info, psk, psk_id):
    VerifyPSKInputs(mode, psk, psk_id)

    key, base_nonce, exporter_secret =
        CombineSecrets(mode, shared_secret, info, psk, psk_id)

    return Context<ROLE>(key, base_nonce, 0, exporter_secret)
```

The ROLE template parameter is either S or R, depending on the role of sender or recipient, respectively. The third parameter in the Context<ROLE> refers to the sequence number, that is initialised with a 0 value. See Section 5.2 for a discussion of the key schedule output, including the role-specific Context structure and its Application Programming Interface (API), and the usage of the sequence number.

Note that the key\_schedule\_context construction in KeySchedule() is equivalent to serializing a structure of the following form in the TLS presentation syntax:

```
struct {  
    uint8 mode;  
    opaque psk_id_hash[Nh];  
    opaque info_hash[Nh];  
} KeyScheduleContext;
```

#### 5.1.1. Encryption to a Public Key

The most basic function of an HPKE scheme is to enable encryption to the holder of a given KEM private key. The SetupBaseS() and SetupBaseR() procedures establish contexts that can be used to encrypt and decrypt, respectively, for a given private key.

The KEM shared secret is combined via the KDF with information describing the key exchange, as well as the explicit info parameter provided by the caller.

The parameter pkR is a public key, and enc is an encapsulated KEM shared secret.

```
def SetupBaseS(pkR, info):  
    shared_secret, enc = Encap(pkR)  
    return enc, KeyScheduleS(mode_base, shared_secret, info,  
                             default_psk, default_psk_id)  
  
def SetupBaseR(enc, skR, info):  
    shared_secret = Decap(enc, skR)  
    return KeyScheduleR(mode_base, shared_secret, info,  
                       default_psk, default_psk_id)
```

### 5.1.2. Authentication Using a Pre-Shared Key

This variant extends the base mechanism by allowing the recipient to authenticate that the sender possessed a given PSK. The PSK also improves confidentiality guarantees in certain adversary models, as described in more detail in Section 9.1. We assume that both parties have been provisioned with both the PSK value `psk` and another byte string `psk_id` that is used to identify which PSK should be used.

The primary difference from the base case is that the `psk` and `psk_id` values are used as `ikm` inputs to the KDF (instead of using the empty string).

The PSK MUST have at least 32 bytes of entropy and SHOULD be of length `Nh` bytes or longer. See Section 9.5 for a more detailed discussion.

```
def SetupPSKS(pkR, info, psk, psk_id):
    shared_secret, enc = Encap(pkR)
    return enc, KeyScheduleS(mode_psk, shared_secret, info, psk, psk_id)

def SetupPSKR(enc, skR, info, psk, psk_id):
    shared_secret = Decap(enc, skR)
    return KeySchedulerR(mode_psk, shared_secret, info, psk, psk_id)
```

### 5.2. Encryption and Decryption

HPKE allows multiple encryption operations to be done based on a given setup transaction. Since the public key operations involved in setup are typically more expensive than symmetric encryption or decryption, this allows applications to amortize the cost of the public key operations, reducing the overall overhead.

In order to avoid nonce reuse, however, this encryption must be stateful. Each of the setup procedures above produces a role-specific context object that stores the AEAD and secret export parameters. The AEAD parameters consist of:

- \* The AEAD algorithm in use
- \* A secret key
- \* A base nonce `base_nonce`
- \* A sequence number (initially 0)

The secret export parameters consist of:

- \* The HPKE ciphersuite in use and
- \* An `exporter_secret` used for the secret export interface (see Section 5.3)

All these parameters except the AEAD sequence number are constant. The sequence number provides nonce uniqueness: The nonce used for each encryption or decryption operation is the result of XORing `base_nonce` with the current sequence number, encoded as a big-endian integer of the same length as `base_nonce`. Implementations MAY use a sequence number that is shorter than the nonce length (padding on the left with zero), but MUST raise an error if the sequence number overflows. The AEAD algorithm produces ciphertext that is `Nt` bytes longer than the plaintext. `Nt` = 16 for AEAD algorithms defined in this document.

Additionally, each AEAD algorithm has a maximum plaintext length (`P_MAX`) as specified in [RFC5116]. For AES-128-GCM and AES-256-GCM, `P_MAX` is  $2^{36} - 31$  bytes. For ChaCha20Poly1305, `P_MAX` is  $2^{38} - 64$  bytes. Implementations MUST NOT encrypt plaintexts larger than `P_MAX`. Exceeding either the sequence number limit or `P_MAX` for the AEAD in use results in loss of confidentiality and integrity guarantees.

Encryption is unidirectional from sender to recipient. The sender's context can encrypt a plaintext `pt` with associated data `aad` as follows:

```
def ContextS.Seal(aad, pt):
    ct = Seal(self.key, self.ComputeNonce(self.seq), aad, pt)
    self.IncrementSeq()
    return ct
```

The recipient's context can decrypt a ciphertext `ct` with associated data `aad` as follows:

```
def ContextR.Open(aad, ct):
    pt = Open(self.key, self.ComputeNonce(self.seq), aad, ct)
    if pt == OpenError:
        raise OpenError
    self.IncrementSeq()
    return pt
```

Each encryption or decryption operation increments the sequence number for the context in use. The per-message nonce and sequence number increment details are as follows:

```
def Context<ROLE>.ComputeNonce(seq):
    seq_bytes = I2OSP(seq, Nn)
    return xor(self.base_nonce, seq_bytes)

def Context<ROLE>.IncrementSeq():
    if self.seq >= (1 << (8*Nn)) - 1:
        raise MessageLimitReachedError
    self.seq += 1
```

The sender's context MUST NOT be used for decryption. Similarly, the recipient's context MUST NOT be used for encryption. Higher-level protocols reusing the HPKE key exchange for more general purposes can derive separate keying material as needed using the secret export interface; see Section 5.3 and Section 9.8 for more details.

It is up to the application to ensure that encryptions and decryptions are done in the proper sequence, so that encryption and decryption nonces align. If ContextS.Seal() or ContextR.Open() would cause the seq parameter to overflow, then the implementation MUST fail with an error. (In the pseudocode above, Context<ROLE>.IncrementSeq() fails with an error when seq overflows, which causes ContextS.Seal() and ContextR.Open() to fail accordingly.) Note that the internal Seal() and Open() calls inside correspond to the context's AEAD algorithm.

In parallel or concurrent environments, the ComputeNonce(), Seal(), and IncrementSeq() operations MUST be done as an atomic unit, e.g., by holding a lock on ContextS for the duration of ContextS.Seal(). If other calls to Seal() are interleaved before IncrementSeq() completes, this will result in multiple encryptions with the same sequence number, and thus the same nonce.

### 5.3. Secret Export

HPKE provides an interface for exporting secrets from the encryption context using a variable-length pseudorandom function (PRF). This interface takes as input a context string exporter\_context and a desired length L in bytes, and produces a secret derived from the internal exporter secret using the corresponding KDF Expand function. For the KDFs defined in this specification, L has a maximum value of  $255 \cdot N_h$ . Future specifications that define new KDFs MUST specify a bound for L.

The exporter\_context parameter has a maximum length that depends on the KDF itself, on the definition of LabeledExpand(), and on the constant labels used together with them. See Section 7.2.1 for precise limits on this length.

```
# For use with a one-stage KDF
def Context.Export_OneStage(exporter_context, L):
    return LabeledDerive(self.exporter_secret, "sec",
                        exporter_context, L)
```

```
# For use with a two-stage KDF
def Context.Export_TwoStage(exporter_context, L):
    return LabeledExpand(self.exporter_secret, "sec",
                        exporter_context, L)
```

Applications that do not use the encryption API in Section 5.2 can use the export-only AEAD ID 0xFFFF when computing the key schedule. Such applications can avoid computing the key and base\_nonce values in the key schedule, as they are not used by the Export interface described above.

Unlike the similar TLS 1.3 exporter interface (see Section 7.5 of [RFC8446]), the HPKE export interface does not provide replay protection. While the resulting secret will only be known to the sender and recipient, a replayed encapsulated secret enc will produce an identical context, and thus the same exported secrets. In particular, applications MUST NOT use exported secrets unless it is safe for the same exported values to be used multiple times. For example, applications MUST NOT use an exported secret to derive a (key, nonce) pair for AEAD encryption (as suggested in Section 9.8 of [RFC9180]), since reuse of a (key, nonce) pair harms security in most AEAD algorithms. In such cases, applications SHOULD incorporate a fresh recipient-provided nonce when deriving values from an export context, as discussed in Section 4.4 of [RFC9458] and Section 9.8.

## 6. Single-Shot APIs

### 6.1. Encryption and Decryption

In many cases, applications encrypt only a single message to a recipient's public key. This section provides templates for HPKE APIs that implement stateless "single-shot" encryption and decryption using APIs specified in Section 5.1 and Section 5.2:

```
def Seal<MODE>(pkR, info, aad, pt, ...):
    enc, ctx = Setup<MODE>S(pkR, info, ...)
    ct = ctx.Seal(aad, pt)
    return enc, ct

def Open<MODE>(enc, skR, info, aad, ct, ...):
    ctx = Setup<MODE>R(enc, skR, info, ...)
    return ctx.Open(aad, ct)
```

The MODE template parameter is either Base or PSK. The optional parameters indicated by "..." depend on MODE and may be empty. For example, SetupBase() has no additional parameters. SealPSK() and OpenPSK() would be implemented as follows:

```
def SealPSK(pkR, info, aad, pt, psk, psk_id):
    enc, ctx = SetupPSKS(pkR, info, psk, psk_id)
    ct = ctx.Seal(aad, pt)
    return enc, ct

def OpenPSK(enc, skR, info, aad, ct, psk, psk_id):
    ctx = SetupPSKR(enc, skR, info, psk, psk_id)
    return ctx.Open(aad, ct)
```

## 6.2. Secret Export

Applications may also want to derive a secret known only to a given recipient. This section provides templates for HPKE APIs that implement stateless "single-shot" secret export using APIs specified in Section 5.3:

```
def SendExport<MODE>(pkR, info, exporter_context, L, ...):
    enc, ctx = Setup<MODE>S(pkR, info, ...)
    exported = ctx.Export(exporter_context, L)
    return enc, exported

def ReceiveExport<MODE>(enc, skR, info, exporter_context, L, ...):
    ctx = Setup<MODE>R(enc, skR, info, ...)
    return ctx.Export(exporter_context, L)
```

As in Section 6.1, the MODE template parameter is either Base or PSK. The optional parameters indicated by "..." depend on MODE and may be empty.

Secrets exported using this single-shot API face the same replay risks discussed in Section 5.3. Usage of exported secrets needs to be limited as described in that section.

## 7. Algorithm Identifiers

This section lists algorithm identifiers suitable for different HPKE configurations. Future specifications may introduce new KEM, KDF, and AEAD algorithm identifiers and retain the security guarantees presented in this document provided they adhere to the security requirements in Section 9.2, Section 9.3, and Section 9.4, respectively.

## 7.1. Key Encapsulation Mechanisms (KEMs)

Value	KEM	Nsecret	Nenc	Npk	Nsk	Auth	Reference
0x0000	Reserved	N/A	N/A	N/A	N/A	yes	RFC 9180
0x0010	DHKEM(P-256, HKDF-SHA256)	32	65	65	32	yes	[NISTCurves], [RFC5869]
0x0011	DHKEM(P-384, HKDF-SHA384)	48	97	97	48	yes	[NISTCurves], [RFC5869]
0x0012	DHKEM(P-521, HKDF-SHA512)	64	133	133	66	yes	[NISTCurves], [RFC5869]
0x0020	DHKEM(X25519, HKDF-SHA256)	32	32	32	32	yes	[RFC7748], [RFC5869]
0x0021	DHKEM(X448, HKDF-SHA512)	64	56	56	56	yes	[RFC7748], [RFC5869]

Table 2: KEM IDs

The Auth column indicates if the KEM algorithm provides the AuthEncap()/AuthDecap() interface defined in [RFC9180].

## 7.1.1. SerializePublicKey and DeserializePublicKey

For P-256, P-384, and P-521, the SerializePublicKey() function of the KEM performs the uncompressed Elliptic-Curve-Point-to-Octet-String conversion according to [SECG]. DeserializePublicKey() performs the uncompressed Octet-String-to-Elliptic-Curve-Point conversion.

For X25519 and X448, the SerializePublicKey() and DeserializePublicKey() functions are the identity function, since these curves already use fixed-length byte strings for public keys.

Some deserialized public keys MUST be validated before they can be used. See Section 7.1.4 for specifics.

### 7.1.2. SerializePrivateKey and DeserializePrivateKey

As per [SECG], P-256, P-384, and P-521 private keys are field elements in the scalar field of the curve being used. For this section, and for Section 7.1.3, it is assumed that implementors of ECDH over these curves use an integer representation of private keys that is compatible with the OS2IP() function.

For P-256, P-384, and P-521, the SerializePrivateKey() function of the KEM performs the Field-Element-to-Octet-String conversion according to [SECG]. If the private key is an integer outside the range  $[0, \text{order}-1]$ , where order is the order of the curve being used, the private key MUST be reduced to its representative in  $[0, \text{order}-1]$  before being serialized. DeserializePrivateKey() performs the Octet-String-to-Field-Element conversion according to [SECG].

For X25519 and X448, private keys are identical to their byte string representation.

To catch invalid keys early on, implementors of DHKEMs SHOULD check that deserialized private keys are not equivalent to 0 (mod order), where order is the order of the DH group. Note that this property is trivially true for X25519 and X448 groups, since clamped values can never be 0 (mod order).

### 7.1.3. DeriveKeyPair

The keys that DeriveKeyPair() produces have only as much entropy as the provided input keying material. For a given KEM, the ikm parameter given to DeriveKeyPair() SHOULD have length at least Nsk, and SHOULD have at least Nsk bytes of entropy.

All invocations of KDF functions (such as LabeledExtract or LabeledExpand) in any DHKEM's DeriveKeyPair() function use the DHKEM's associated KDF (as opposed to the ciphersuite's KDF).

For P-256, P-384, and P-521, the DeriveKeyPair() function of the KEM performs rejection sampling over field elements:

```
# For use with a one-stage KDF
def DeriveCandidate_OneStage(ikm, counter):
    return LabeledDerive(ikm, "candidate", I2OSP(counter, 1), Nsk)
```

```
# For use with a two-stage KDF
def DeriveCandidate_TwoStage(ikm, counter):
    # Note: dkp_prk may be derived once and cached
    dkp_prk = LabeledExtract("", "dkp_prk", ikm)
    return LabeledExpand(dkp_prk, "candidate",
                        I2OSP(counter, 1), Nsk)
```

```
def DeriveKeyPair(ikm):
    sk = 0
    counter = 0
    while sk == 0 or sk >= order:
        if counter > 255:
            raise DeriveKeyPairError
        bytes = DeriveCandidate(ikm, counter)
        bytes[0] = bytes[0] & bitmask
        sk = OS2IP(bytes)
        counter = counter + 1
    return (sk, pk(sk))
```

order is the order of the curve being used (see Section D.1.2 of [NISTCurves]), and is listed below for completeness.

P-256:  
0xffffffff00000000fffffffffffffffffbce6faada7179e84f3b9cac2fc632551

P-384:  
0xfffc7634d81f4372ddf  
581a0db248b0a77aecec196accc52973

P-521:  
0x01ff  
fa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb71e91386409

bitmask is defined to be 0xFF for P-256 and P-384, and 0x01 for P-521. The precise likelihood of DeriveKeyPair() failing with DeriveKeyPairError depends on the group being used, but it is negligibly small in all cases. See Section 8.2 for information about dealing with such failures.

For X25519 and X448, the DeriveKeyPair() function applies a KDF to the input:

```
# For use with a one-stage KDF
def DeriveKeyPair_OneStage(ikm):
    sk = LabeledDerive(ikm, "sk", "", Nsk)
    return (sk, pk(sk))

# For use with a two-stage KDF
def DeriveKeyPair_TwoStage(ikm):
    dkp_prk = LabeledExtract("", "dkp_prk", ikm)
    sk = LabeledExpand(dkp_prk, "sk", "", Nsk)
    return (sk, pk(sk))
```

The suite\_id used implicitly in LabeledExtract() and LabeledExpand() for DeriveKeyPair(ikm) is derived from the KEM identifier of the DHKEM in use (see Section 7.1), that is, based on the type of key pair been generated for that DHKEM type.

For all of the above instances of DHKEM, the GenerateKeyPair can be implemented as DeriveKeyPair(random(Nsk)).

#### 7.1.4. Validation of Inputs and Outputs

The following public keys are subject to validation if the group requires public key validation: the sender MUST validate the recipient's public key pkR; the recipient MUST validate the ephemeral public key pkE. Validation failure yields a ValidationError.

For P-256, P-384, and P-521, senders and recipients MUST perform partial public key validation on all public key inputs, as defined in Section 5.6.2.3.4 of [keyagreement]. This includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, senders and recipients MUST ensure the Diffie-Hellman shared secret is not the point at infinity.

For X25519 and X448, public keys and Diffie-Hellman outputs MUST be validated as described in [RFC7748]. In particular, recipients MUST check whether the Diffie-Hellman shared secret is the all-zero value and abort if so.

#### 7.1.5. Future KEMs

Section 9.2 lists security requirements on a KEM used within HPKE.

A KEM algorithm may support different encoding algorithms, with different output lengths, for KEM public keys. Such KEM algorithms MUST specify only one encoding algorithm whose output length is Npk.

## 7.2. Key Derivation Functions (KDFs)

Value	KDF	Nh	Two-Stage	Reference
0x0000	Reserved	N/A	N/A	RFC 9180
0x0001	HKDF-SHA256	32	Y	[RFC5869]
0x0002	HKDF-SHA384	48	Y	[RFC5869]
0x0003	HKDF-SHA512	64	Y	[RFC5869]

Table 3: KDF IDs

### 7.2.1. Input Length Restrictions

For one-stage KDFs, there is length limit of 65,535 bytes for the `psk`, `psk_id`, `info` fields. This limitation arises because these fields are all prefixed with a two-byte length when being used as KDF inputs. There is no inherent length limitation on `exporter_context`. If a one-stage KDF has an input length limit, then implementations MUST limit the length of `exporter_context` accordingly, so that the `LabeledDerive` call in `Context.Export` does not overflow the input length limit.

For two-stage KDFs, this document defines `LabeledExtract()` and `LabeledExpand()` based on the KDFs listed above. These functions add prefixes to their respective inputs `ikm` and `info` before calling the KDF's `Extract()` and `Expand()` functions. This leads to a reduction of the maximum input length that is available for the inputs `psk`, `psk_id`, `info`, `exporter_context`, `ikm`, i.e., the variable-length parameters provided by HPKE applications. The following table lists the maximum allowed lengths of these parameters for the KDFs defined in this document, as inclusive bounds in bytes:

Input	HKDF-SHA256	HKDF-SHA384	HKDF-SHA512
psk	$2^{\{61\}} - 88$	$2^{\{125\}} - 152$	$2^{\{125\}} - 152$
psk_id	$2^{\{61\}} - 93$	$2^{\{125\}} - 157$	$2^{\{125\}} - 157$
info	$2^{\{61\}} - 91$	$2^{\{125\}} - 155$	$2^{\{125\}} - 155$
exporter_context	$2^{\{61\}} - 120$	$2^{\{125\}} - 200$	$2^{\{125\}} - 216$
ikm (DeriveKeyPair)	$2^{\{61\}} - 84$	$2^{\{125\}} - 148$	$2^{\{125\}} - 148$

Table 4: Application Input Limits

This shows that the limits are only marginally smaller than the maximum input length of the underlying hash function; these limits are large and unlikely to be reached in practical applications. Future specifications that define new KDFs MUST specify bounds for these variable-length parameters.

Since the above bounds are larger than any values used in practice, it may be useful for implementations to impose a lower limit on the values they will accept (for example, to avoid dynamic allocations). Implementations MUST support info values of at least 64 bytes. Implementations SHOULD support info values of at least 16384 bytes to accommodate protocols such as Encrypted Client Hello [I-D.ietf-tls-esni]. Applications seeking maximum interoperability with resource-constrained HPKE implementations SHOULD NOT provide info values exceeding 64 bytes without confirmation that an implementation supports larger info values.

The values for psk, psk\_id, info, and ikm, which are inputs to LabeledExtract(), were computed with the following expression:

$$\text{max\_size\_hash\_input} - \text{Nb} - \text{size\_version\_label} - \text{size\_suite\_id} - \text{size\_input\_label}$$

The value for exporter\_context, which is an input to LabeledExpand(), was computed with the following expression:

$$\text{max\_size\_hash\_input} - \text{Nb} - \text{Nh} - \text{size\_version\_label} - \text{size\_suite\_id} - \text{size\_input\_label} - 2 - 1$$

In these equations, `max_size_hash_input` is the maximum input length of the underlying hash function in bytes, `Nb` is the block size of the underlying hash function in bytes, `size_version_label` is the size of "HPKE-v1" in bytes and equals 7, `size_suite_id` is the size of the `suite_id` in bytes and equals 5 for DHKEM (relevant for `ikm`) and 10 for the remainder of HPKE (relevant for `psk`, `psk_id`, `info`, and `exporter_context`), and `size_input_label` is the size in bytes of the label used as parameter to `LabeledExtract()` or `LabeledExpand()`, the maximum of which is 13 across all labels in this document.

### 7.3. Authenticated Encryption with Associated Data (AEAD) Functions

Value	AEAD	Nk	Nn	Nt	Reference
0x0000	Reserved	N/A	N/A	N/A	RFC 9180
0x0001	AES-128-GCM	16	12	16	[GCM]
0x0002	AES-256-GCM	32	12	16	[GCM]
0x0003	ChaCha20Poly1305	32	12	16	[RFC8439]
0xFFFF	Export-only	0	0	N/A	RFC 9180

Table 5: AEAD IDs

The 0xFFFF AEAD ID is reserved for applications that only use the Export interface; see Section 5.3 for more details.

## 8. API Considerations

This section documents considerations for interfaces to implementations of HPKE. This includes error handling considerations and recommendations that improve interoperability when HPKE is used in applications.

### 8.1. Auxiliary Authenticated Application Information

HPKE has two places at which applications can specify auxiliary authenticated information: (1) during context construction via the `Setup info` parameter, and (2) during Context operations, i.e., with the `aad` parameter for `Open()` and `Seal()`, and the `exporter_context` parameter for `Export()`. Application information applicable to multiple operations on a single Context should use the `Setup info` parameter. This avoids redundantly processing this information for each Context operation. In contrast, application information that

varies on a per-message basis should be specified via the Context APIs (`Seal()`, `Open()`, or `Export()`).

Applications that only use the single-shot APIs described in Section 6 should use the Setup info parameter for specifying auxiliary authenticated information. Implementations which only expose single-shot APIs should not allow applications to use both Setup info and Context aad or exporter\_context auxiliary information parameters.

## 8.2. Errors

The high-level, public HPKE APIs specified in this document are all fallible. These include the Setup functions and all encryption context functions. For example, `Decap()` can fail if the encapsulated secret enc is invalid, and `Open()` may fail if ciphertext decryption fails. The explicit errors generated throughout this specification, along with the conditions that lead to each error, are as follows:

- \* `ValidationError`: KEM input or output validation failure; Section 4.5.
- \* `DeserializeError`: Public or private key deserialization failure; Section 4.
- \* `EncapError`: `Encap()` failure; Section 4.
- \* `DecapError`: `Decap()` failure; Section 4.
- \* `OpenError`: Context AEAD `Open()` failure; Section 4 and Section 5.2.
- \* `MessageLimitReachedError`: Context AEAD sequence number overflow; Section 4 and Section 5.2.
- \* `DeriveKeyPairError`: Key pair derivation failure; Section 7.1.3.

Implicit errors may also occur. As an example, certain classes of failures, e.g., malformed recipient public keys, may not yield explicit errors. For example, for the DHKEM variant described in this specification, the `Encap()` algorithm fails when given an invalid recipient public key. However, other KEM algorithms may not have an efficient algorithm for verifying the validity of public keys. As a result, an equivalent error may not manifest until AEAD decryption at the recipient.

The errors in this document are meant as a guide for implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, future KEMs might have internal failure cases, or an implementation might run out of memory.

How these errors are expressed in an API or handled by applications is an implementation-specific detail. For example, some implementations may abort or panic upon a `DeriveKeyPairError` failure given that it only occurs with negligible probability, whereas other implementations may retry the failed `DeriveKeyPair` operation. See Section 7.1.3 for more information. As another example, some implementations of the DHKEM specified in this document may choose to transform `ValidationError` from `DH()` into an `EncapError` or `DecapError` from `Encap()` or `Decap()`, respectively, whereas others may choose to raise `ValidationError` unmodified.

Applications using HPKE APIs should not assume that the errors here are complete, nor should they assume certain classes of errors will always manifest the same way for all ciphersuites. For example, the DHKEM specified in this document will emit a `DeserializationError` or `ValidationError` if a KEM public key is invalid. However, a new KEM might not have an efficient algorithm for determining whether or not a public key is valid. In this case, an invalid public key might instead yield an `OpenError` when trying to decrypt a ciphertext.

## 9. Security Considerations

### 9.1. Security Properties

HPKE has several security goals, depending on the mode of operation, against active and adaptive attackers that can compromise partial secrets of senders and recipients. The desired security goals are detailed below:

- \* Message secrecy: Confidentiality of the sender's messages against chosen ciphertext attacks
- \* Export key secrecy: Indistinguishability of each export secret from a uniformly random bitstring of equal length, i.e., `Context.Export` is a variable-length PRF
- \* Sender authentication: Proof of sender origin for the PSK mode

These security goals are expected to hold for any honest sender and honest recipient keys, as well as if the honest sender and honest recipient keys are the same.

HPKE mitigates malleability problems (called benign malleability [SECG]) in prior public key encryption standards based on ECIES by including all public keys in the context of the key schedule.

HPKE does not provide forward secrecy with respect to recipient compromise. In the Base mode, the secrecy properties are only expected to hold if the recipient private key  $sk_R$  is not compromised at any point in time. In the PSK mode, the secrecy properties are expected to hold if the recipient private key  $sk_R$  and the pre-shared key are not both compromised at any point in time. See Section 9.7 for more details.

Besides forward secrecy and key-compromise impersonation, which are highlighted in this section because of their particular cryptographic importance, HPKE has other non-goals that are described in Section 9.7: no tolerance of message reordering or loss, no downgrade or replay prevention, no hiding of the plaintext length, and no protection against bad ephemeral randomness. Section 9.7 suggests application-level mitigations for some of them.

#### 9.1.1. Computational Analysis

It is shown in [CS01] that a hybrid public key encryption scheme of essentially the same form as the Base mode described here is IND-CCA2-secure as long as the underlying KEM and AEAD schemes are IND-CCA2-secure. Moreover, it is shown in [HHK06] that IND-CCA2 security of the KEM and the data encapsulation mechanism are necessary conditions to achieve IND-CCA2 security for hybrid public key encryption. The main difference between the scheme proposed in [CS01] and the Base mode in this document (both named HPKE) is that we interpose some KDF calls between the KEM and the AEAD. Analyzing the HPKE Base mode instantiation in this document therefore requires verifying that the additional KDF calls do not cause the IND-CCA2 property to fail, as well as verifying the additional export key secrecy property.

A preliminary computational analysis of all HPKE modes has been done in [HPKEAnalysis], indicating asymptotic security for the case where the KEM is DHKEM, the AEAD is any IND-CPA-secure and INT-CTXT-secure scheme, and the DH group and KDF satisfy the following conditions:

- \* DH group: The gap Diffie-Hellman (GDH) problem is hard in the appropriate subgroup [GAP].
- \* Extract() and Expand(): Extract() can be modeled as a random oracle. Expand() can be modeled as a pseudorandom function, wherein the first argument is the key.

In particular, the KDFs and DH groups defined in this document (see Section 7.2 and Section 7.1) satisfy these properties when used as specified. The analysis in [HPKEAnalysis] demonstrates that under these constraints, HPKE continues to provide IND-CCA2 security, and provides the additional properties noted above. Also, the analysis confirms the expected properties hold under the different key compromise cases mentioned above. The analysis considers a sender that sends one message using the encryption context, and additionally exports two independent secrets using the secret export interface.

The table below summarizes the main results from [HPKEAnalysis]. N/A means that a property does not apply for the given mode, whereas Y means the given mode satisfies the property.

Variant	Message Sec.	Export Sec.	Sender Auth.
Base	Y	Y	N/A
PSK	Y	Y	Y

Table 6

If non-DH-based KEMs are to be used with HPKE, further analysis will be necessary to prove their security. The results from [CS01] provide some indication that any IND-CCA2-secure KEM will suffice here, but are not conclusive given the differences in the schemes.

#### 9.1.2. Post-Quantum Security

All of [CS01], [HPKEAnalysis], and [ABHKLR20] are premised on classical security models and assumptions, and do not consider adversaries capable of quantum computation. A full proof of post-quantum security would need to take appropriate security models and assumptions into account, in addition to simply using a post-quantum KEM.

In future work, the analysis from [ABHKLR20] can be extended to cover HPKE's other modes and desired security properties. The hybrid quantum-resistance property described above, which is achieved by using the PSK mode, is not proven in [HPKEAnalysis] because this analysis requires the random oracle model; in a quantum setting, this model needs adaption to, for example, the quantum random oracle model.

## 9.2. Security Requirements on a KEM Used within HPKE

A KEM used within HPKE MUST allow HPKE to satisfy its desired security properties described in Section 9.1. Section 9.6 lists requirements concerning domain separation.

In particular, the KEM shared secret MUST be a uniformly random byte string of length `Nsecret`. This means, for instance, that it would not be sufficient if the KEM shared secret is only uniformly random as an element of some set prior to its encoding as a byte string.

### 9.2.1. Encap/Decap Interface

As mentioned in Section 9, [CS01] provides some indications that if the KEM's `Encap()/Decap()` interface (which is used in the Base and PSK modes) is IND-CCA2-secure, HPKE is able to satisfy its desired security properties. An appropriate definition of IND-CCA2 security for KEMs can be found in [CS01] and [BHK09].

### 9.2.2. KEM Key Reuse

An `ikm` input to `DeriveKeyPair()` (Section 7.1.3) MUST NOT be reused elsewhere, in particular not with `DeriveKeyPair()` of a different KEM.

Since a KEM key pair belonging to a sender or recipient works with all modes, it can be used with multiple modes in parallel. HPKE is constructed to be secure in such settings due to domain separation using the `suite_id` variable. However, there is no formal proof of security at the time of writing for using multiple modes in parallel; [HPKEAnalysis] and [ABHKLR20] only analyze isolated modes.

## 9.3. Security Requirements on a KDF

The choice of the KDF for HPKE SHOULD be made based on the security level provided by the KEM and, if applicable, by the PSK. The KDF SHOULD at least have the security level of the KEM and SHOULD at least have the security level provided by the PSK.

## 9.4. Security Requirements on an AEAD

All AEADs MUST be IND-CCA2-secure, as is currently true for all AEADs listed in Section 7.3.

## 9.5. Pre-Shared Key Recommendations

In the PSK modes, the PSK MUST have at least 32 bytes of entropy and SHOULD be of length `Nh` bytes or longer. Using a PSK longer than 32 bytes but shorter than `Nh` bytes is permitted.

HPKE is specified to use HKDF as its key derivation function. HKDF is not designed to slow down dictionary attacks (see [RFC5869]). Thus, HPKE's PSK mechanism is not suitable for use with a low-entropy password as the PSK: In scenarios in which the adversary knows the KEM shared secret `shared_secret` and has access to an oracle that distinguishes between a good and a wrong PSK, it can perform PSK-recovering attacks. This oracle can be the decryption operation on a captured HPKE ciphertext or any other recipient behavior that is observably different when using a wrong PSK. The adversary knows the KEM shared secret `shared_secret` if it knows all KEM private keys of one participant. In the PSK mode, this is trivially the case if the adversary acts as the sender.

To recover a lower entropy PSK, an attacker in this scenario can trivially perform a dictionary attack. Given a set  $S$  of possible PSK values, the attacker generates an HPKE ciphertext for each value in  $S$ , and submits the resulting ciphertexts to the oracle to learn which PSK is being used by the recipient. Further, because HPKE uses AEAD schemes that are not key-committing, an attacker can mount a partitioning oracle attack [LGR20] that can recover the PSK from a set of  $S$  possible PSK values, with  $|S| = m \cdot k$ , in roughly  $m + \log k$  queries to the oracle using ciphertexts of length proportional to  $k$ , the maximum message length in blocks. (Applying the multi-collision algorithm from [LGR20] requires a small adaptation to the algorithm wherein the appropriate nonce is computed for each candidate key. This modification adds one call to HKDF per key. The number of partitioning oracle queries remains unchanged.) As a result, the PSK must therefore be chosen with sufficient entropy so that  $m + \log k$  is prohibitive for attackers (e.g.,  $2^{128}$ ). Future specifications can define new AEAD algorithms that are key-committing.

## 9.6. Domain Separation

HPKE allows combining a DHKEM variant  $\text{DHKEM}(\text{Group}, \text{KDF}')$  and a KDF such that both KDFs are instantiated by the same KDF. By design, the calls to `Extract()` and `Expand()` inside DHKEM and the remainder of HPKE use separate input domains. This justifies modeling them as independent functions even if instantiated by the same KDF. This domain separation between DHKEM and the remainder of HPKE is achieved by using prefix-free sets of `suite_id` values in `LabeledExtract()`, `LabeledExpand()`, and `LabeledDerive` (KEM... in DHKEM and HPKE... in the remainder of HPKE). Recall that a set is prefix-free if no element is a prefix of another within the set.

Separation between uses of the one-stage and two-stage KDFs is ensured by the inclusion of the `suite_id` in `LabeledExtract`, `LabeledExpand`, and `LabeledDerive`.

Future KEM instantiations MUST ensure, should `Extract()`, `Expand()`, and/or `Derive()` be used internally, that they can be modeled as functions independent from the invocations of these functions in the remainder of HPKE. One way to ensure this is by using `LabeledExtract()` / `LabeledExpand()` / `LabeledDerive()` functions with a `suite_id` as defined in Section 4, which will ensure input domain separation, as outlined above. Particular attention needs to be paid if the KEM directly invokes functions that are used internally in HPKE's `Extract()` or `Expand()`, such as `Hash()` and `HMAC()` in the case of HKDF. It MUST be ensured that inputs to these invocations cannot collide with inputs to the internal invocations of these functions inside `Extract()` or `Expand()`. In HPKE's `KeySchedule()` this is avoided by using `Extract()` instead of `Hash()` on the arbitrary-length inputs `info` and `psk_id`.

The string literal "HPKE-v1" used in `LabeledExtract()` / `LabeledExpand()` / `LabeledDerive()` ensures that any secrets derived in HPKE are bound to the scheme's name and version, even when possibly derived from the same Diffie-Hellman or KEM shared secret as in another scheme or version.

## 9.7. Application Embedding and Non-Goals

HPKE is designed to be a fairly low-level mechanism. As a result, it assumes that certain properties are provided by the application in which HPKE is embedded and leaves certain security properties to be provided by other mechanisms. Otherwise said, certain properties are out-of-scope for HPKE.

### 9.7.1. Message Order and Message Loss

The primary requirement that HPKE imposes on applications is the requirement that ciphertexts MUST be presented to `ContextR.Open()` in the same order in which they were generated by `ContextS.Seal()`. When the single-shot API is used (see Section 6), this is trivially true (since there is only ever one ciphertext). Applications that allow for multiple invocations of `Open()` / `Seal()` on the same context MUST enforce the ordering property described above.

Ordering requirements of this character are usually fulfilled by providing a sequence number in the framing of encrypted messages. Whatever information is used to determine the ordering of HPKE-encrypted messages SHOULD be included in the AAD passed to `ContextS.Seal()` and `ContextR.Open()`. The specifics of this scheme are up to the application.

HPKE is not tolerant of lost messages. Applications MUST be able to detect when a message has been lost. When an unrecoverable loss is detected, the application MUST discard any associated HPKE context.

#### 9.7.2. Downgrade Prevention

HPKE assumes that the sender and recipient agree on what algorithms to use. Depending on how these algorithms are negotiated, it may be possible for an intermediary to force the two parties to use suboptimal algorithms.

#### 9.7.3. Replay Protection

The requirement that ciphertexts be presented to the `ContextR.Open()` function in the same order they were generated by `ContextS.Seal()` provides a degree of replay protection within a stream of ciphertexts resulting from a given context. HPKE provides no other replay protection.

While a sender can guarantee the uniqueness of HPKE ciphertexts, a recipient might receive the same ciphertext multiple times. Unless the recipient takes particular care to guarantee that replay is impossible, such as tracking all enc values that are received, this can result in multiple contexts that have the same shared secret. This is particularly relevant for exported secrets.

If an attacker can cause a recipient to re-use an enc value, any exported secrets will be the same as in the initial transaction. While the exported values are still known only to the sender and recipient (not the replay attacker), such replay can allow the attacker to cause the recipient to re-use the exported values.

Consider the following scenario, in which B is using the recipient-to-sender encryption described as an example in Section 9.8 of [RFC9180]:

```
B->A: pk

A:   enc1, ctx = SetupBaseS(pk)
     ct1 = ctx.seal(aad, pt)
A->B: enc1, ct1

B:   ctx = SetupBaseR(sk, enc1)
     key, nonce = ctx.export(...)
     ct2 = AEAD.seal(key, nonce, aad2, pt2)
B->A: ct2

X->B: enc1, ct1 [replay of previously sent values]

B:   ctx = SetupBaseR(sk, enc)
     key, nonce = ctx.export(...)
     ct3 = AEAD.seal(key, nonce, aad3, pt3)
B->X: ct3
```

Figure 1: Attacker-triggered nonce reuse via replay

In this scenario, if `aad2` is different from `aad3` or `pt2` is different from `pt3` (for example, due to the use of a timestamp in either field), then the ciphertexts `ct2` and `ct3` will represent encryptions of different values with the same `(key, nonce)` pair -- a nonce reuse condition that can completely break the authenticated encryption guarantees for several AEAD algorithms, including those defined in Section 7.3.

In order to avoid such risks, applications SHOULD incorporate a fresh recipient-provided nonce when deriving values from an export context, as discussed in Section 4.4 of [RFC9458] and Section 9.8.

#### 9.7.4. Forward Secrecy

HPKE ciphertexts are not forward secret with respect to recipient compromise in any mode. This means that compromise of long-term recipient secrets allows an attacker to decrypt past ciphertexts encrypted under said secrets. This is because only long-term secrets are used on the side of the recipient.

HPKE ciphertexts are forward secret with respect to sender compromise in all modes. This is because ephemeral randomness is used on the sender's side, which is supposed to be erased directly after computation of the KEM shared secret and ciphertext.

#### 9.7.5. Bad Ephemeral Randomness

If the randomness used for KEM encapsulation is bad -- i.e., of low entropy or compromised because of a broken or subverted random number generator -- the confidentiality guarantees of HPKE degrade significantly. In Base mode, confidentiality guarantees can be lost completely; in the other modes, at least forward secrecy with respect to sender compromise can be lost completely.

Such a situation could also lead to the reuse of the same KEM shared secret and thus to the reuse of same key-nonce pairs for the AEAD. The AEADs specified in this document are not secure in case of nonce reuse. This attack vector is particularly relevant in the authenticated mode because knowledge of the ephemeral randomness is not enough to derive `shared_secret` in these modes.

One way for applications to mitigate the impacts of bad ephemeral randomness is to combine ephemeral randomness with a local long-term secret that has been generated securely, as described in [RFC8937].

#### 9.7.6. Hiding Plaintext Length

AEAD ciphertexts produced by HPKE do not hide the plaintext length. Applications requiring this level of privacy should use a suitable padding mechanism. See [I-D.ietf-tls-esni] and [RFC8467] for examples of protocol-specific padding policies.

#### 9.8. Bidirectional Encryption

As discussed in Section 5.2, HPKE encryption is unidirectional from sender to recipient. Applications that require bidirectional encryption can derive necessary keying material with the secret export interface Section 5.3. The type and length of such keying material depends on the application use case.

As an example, if an application needs AEAD encryption from the recipient to the sender, it can derive a key and nonce from the corresponding HPKE context as follows:

```
def EncryptResponse(context, enc, response_aad, response_pt):
    secret = context.Export("[application] response", Nh)
    response_nonce = random(Nh)
    salt = concat(enc, response_nonce)
    prk = Extract(salt, secret)
    aead_key = Expand(prk, "key", Nk)
    aead_nonce = Expand(prk, "nonce", Nn)
    ct = Seal(aead_key, aead_nonce, response_aad, response_pt)
    return (response_nonce, ct)
```

This example mechanism differs from the example mechanism in [RFC9180] by incorporating a per-transaction random value `response_nonce`. Because HPKE does not provide replay protection, the mechanism in [RFC9180] enabled an attacker to trigger reuse of a (key, nonce) pair by replaying an HPKE message under certain application circumstances. Incorporating per-transaction entropy ensures that the key and nonce used in AEAD encryption will be distinct for every invocation of the mechanism.

In this context, HPKE's limitations with regard to sender authentication become limits on recipient authentication. In particular, in the Base mode, there is no authentication of the remote party at all.

### 9.9. Metadata Protection

The PSK mode of HPKE requires that the recipient know what key material to use for the sender. This can be signaled in applications by sending the PSK ID (`psk_id` above) and/or the sender's public key (`pkS`). However, these values themselves might be considered sensitive, since, in a given application context, they might identify the sender.

An application that wishes to protect these metadata values without requiring further provisioning of keys can use an additional instance of HPKE, using the unauthenticated Base mode. Where the application might have sent (`psk_id`, `enc`, `ciphertext`) before, it would now send (`enc2`, `ciphertext2`, `enc`, `ciphertext`), where (`enc2`, `ciphertext2`) represent the encryption of the `psk_id` value.

The cost of this approach is an additional KEM operation each for the sender and the recipient. A potential lower-cost approach (involving only symmetric operations) would be available if the nonce-protection schemes in [BNT19] could be extended to cover other metadata. However, this construction would require further analysis.

## 10. Message Encoding

This document does not specify a wire format encoding for HPKE messages. Applications that adopt HPKE must therefore specify an unambiguous encoding mechanism that includes, minimally: the encapsulated secret `enc`, `ciphertext` value(s) (and order if there are multiple), and any info values that are not implicit. One example of a non-implicit value is the recipient public key used for encapsulation, which may be needed if a recipient has more than one public key.

The AEAD interface used in this document is based on [RFC5116], which produces and consumes a single ciphertext value. As discussed in [RFC5116], this ciphertext value contains the encrypted plaintext as well as any authentication data, encoded in a manner described by the individual AEAD scheme. Some implementations are not structured in this way, instead providing a separate ciphertext and authentication tag. When such AEAD implementations are used in HPKE implementations, the HPKE implementation must combine these inputs into a single ciphertext value within `Seal()` and parse them out within `Open()`, where the parsing details are defined by the AEAD scheme. For example, with the AES-GCM schemes specified in this document, the GCM authentication tag is placed in the last `Nt` bytes of the ciphertext output.

## 11. IANA Considerations

IANA created three new registries as requested in Section 11 of [RFC9180]:

- \* HPKE KEM Identifiers
- \* HPKE KDF Identifiers
- \* HPKE AEAD Identifiers

All these registries are under "Hybrid Public Key Encryption", and administered under a Specification Required policy [RFC8126].

This document requests that entries in these registries referring to RFC 9180 be updated to refer to this document.

### 11.1. KEM Identifiers

The "HPKE KEM Identifiers" registry lists identifiers for key encapsulation algorithms defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is `0xFFFF` = 65535.

Template:

- \* Value: The two-byte identifier for the algorithm
- \* KEM: The name of the algorithm
- \* Nsecret: The length in bytes of a KEM shared secret produced by the algorithm

- \* Nenc: The length in bytes of an encoded encapsulated secret produced by the algorithm
- \* Npk: The length in bytes of an encoded public key for the algorithm
- \* Nsk: The length in bytes of an encoded private key for the algorithm
- \* Auth: A boolean indicating if this algorithm provides the AuthEncap()/AuthDecap() interface
- \* Reference: Where this algorithm is defined

Initial contents: Provided in Table 2

### 11.2. KDF Identifiers

The "HPKE KDF Identifiers" registry lists identifiers for key derivation functions defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

Template:

- \* Value: The two-byte identifier for the algorithm
- \* KDF: The name of the algorithm
- \* Nh: For two-stage KDFs, the output size of the Extract function in bytes. For one-stage KDFs, the security strength in bytes, as defined for the KDF identifier.
- \* Two-Stage: Whether the KDF provides Extract and Expand functions (Y) or only a single-stage Derive function (N). N/A for reserved entries.
- \* Reference: Where this algorithm is defined

Initial contents: Provided in Table 3

### 11.3. AEAD Identifiers

The "HPKE AEAD Identifiers" registry lists identifiers for authenticated encryption with associated data (AEAD) algorithms defined for use with HPKE. These identifiers are two-byte values, so the maximum possible value is 0xFFFF = 65535.

Template:

- \* Value: The two-byte identifier for the algorithm
- \* AEAD: The name of the algorithm
- \* Nk: The length in bytes of a key for this algorithm
- \* Nn: The length in bytes of a nonce for this algorithm
- \* Nt: The length in bytes of an authentication tag for this algorithm
- \* Reference: Where this algorithm is defined

Initial contents: Provided in Table 5

## 12. References

### 12.1. Normative References

- [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>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/rfc/rfc5116>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/rfc/rfc8017>>.
- [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>>.
- [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>>.
- [RFC9458] Thomson, M. and C. A. Wood, "Oblivious HTTP", RFC 9458, DOI 10.17487/RFC9458, January 2024, <<https://www.rfc-editor.org/rfc/rfc9458>>.

### 12.2. Informative References

- [ABHKLR20] Alwen, J., Blanchet, B., Hauck, E., Kiltz, E., Lipp, B., and D. Riepel, "Analysing the HPKE Standard", 2020, <<https://eprint.iacr.org/2020/1499>>.
- [ANSI] American National Standards Institute, "ANSI X9.63 Public Key Cryptography for the Financial Services Industry -- Key Agreement and Key Transport Using Elliptic Curve Cryptography", 2001.
- [BHK09] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz, "Subtleties in the Definition of IND-CCA: When and How Should Challenge-Decryption be Disallowed?", 2009, <<https://eprint.iacr.org/2009/418>>.
- [BNT19] Bellare, M., Ng, R., and B. Tackmann, "Nonces Are Noticed: AEAD Revisited", 2019, <[http://dx.doi.org/10.1007/978-3-030-26948-7\\_9](http://dx.doi.org/10.1007/978-3-030-26948-7_9)>.
- [CS01] Cramer, R. and V. Shoup, "Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack", 2001, <<https://eprint.iacr.org/2001/108>>.
- [FIPS202] "SHA-3 standard :: permutation-based hash and extendable-output functions", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.202, 2015, <<https://doi.org/10.6028/nist.fips.202>>.
- [GAP] Okamoto, T. and D. Pointcheval, "The Gap-Problems - a New Class of Problems for the Security of Cryptographic Schemes", ISBN 978-3-540-44586-9, 2001, <[https://link.springer.com/content/pdf/10.1007/3-540-44586-2\\_8.pdf](https://link.springer.com/content/pdf/10.1007/3-540-44586-2_8.pdf)>.
- [GCM] Dworkin, M., "Recommendation for block cipher modes of operation :: GaloisCounter Mode (GCM) and GMAC", National Institute of Standards and Technology, DOI 10.6028/nist.sp.800-38d, 2007, <<https://doi.org/10.6028/nist.sp.800-38d>>.
- [HHK06] Herranz, J., Hofheinz, D., and E. Kiltz, "Some (in)sufficient conditions for secure hybrid encryption", 2006, <<https://eprint.iacr.org/2006/265>>.
- [HPKEAnalysis] Lipp, B., "An Analysis of Hybrid Public Key Encryption", 2020, <<https://eprint.iacr.org/2020/243>>.

- [I-D.ietf-tls-esni]  
Rescorla, E., Oku, K., Sullivan, N., and C. A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-25, 14 June 2025, <<https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-25>>.
- [IEEE1363] Institute of Electrical and Electronics Engineers, "IEEE 1363a, Standard Specifications for Public Key Cryptography - Amendment 1 -- Additional Techniques", 2004.
- [IMB] Diffie, W., Van Oorschot, P., and M. Wiener, "Authentication and authenticated key exchanges", Springer Science and Business Media LLC, Designs, Codes and Cryptography vol. 2, no. 2, pp. 107-125, DOI 10.1007/bf00124891, June 1992, <<https://doi.org/10.1007/bf00124891>>.
- [ISO] International Organization for Standardization / International Electrotechnical Commission, "ISO/IEC 18033-2, Information Technology - Security Techniques - Encryption Algorithms - Part 2 -- Asymmetric Ciphers", 2006.
- [keyagreement]  
Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology, DOI 10.6028/nist.sp.800-56ar3, April 2018, <<https://doi.org/10.6028/nist.sp.800-56ar3>>.
- [LGR20] Len, J., Grubbs, P., and T. Ristenpart, "Partitioning Oracle Attacks", 2021, <<https://eprint.iacr.org/2020/1491>>.
- [MAEA10] Gayoso Martinez, V., Hernandez Alvarez, F., Hernandez Encinas, L., and C. Sanchez Avila, "A Comparison of the Standardized Versions of ECIES", 2010, <<https://ieeexplore.ieee.org/abstract/document/5604194>>.
- [NISTCurves]  
"Digital signature standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-4, 2013, <<https://doi.org/10.6028/nist.fips.186-4>>.

- [RFC1421] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures", RFC 1421, DOI 10.17487/RFC1421, February 1993, <<https://www.rfc-editor.org/rfc/rfc1421>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/rfc/rfc5869>>.
- [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>>.
- [RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/rfc/rfc8439>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC8467] Mayrhofer, A., "Padding Policies for Extension Mechanisms for DNS (EDNS(0))", RFC 8467, DOI 10.17487/RFC8467, October 2018, <<https://www.rfc-editor.org/rfc/rfc8467>>.
- [RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <<https://www.rfc-editor.org/rfc/rfc8937>>.
- [RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <<https://www.rfc-editor.org/rfc/rfc9180>>.
- [RFC9420] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., and K. Cohn-Gordon, "The Messaging Layer Security (MLS) Protocol", RFC 9420, DOI 10.17487/RFC9420, July 2023, <<https://www.rfc-editor.org/rfc/rfc9420>>.
- [SECG] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <<https://secg.org/sec1-v2.pdf>>.

[SigncryptionDZ10]

"Practical Signcryption", Springer Berlin Heidelberg,  
Information Security and Cryptography,  
DOI 10.1007/978-3-540-89411-7, ISBN ["9783540894094",  
"9783540894117"], 2010,  
<<https://doi.org/10.1007/978-3-540-89411-7>>.

[TestVectors]

"HPKE Test Vectors", 2021, <<https://github.com/cfrg/draft-irtf-cfrg-hpke/blob/5f503c564da00b0687b3de75f1dfbdfc4079ad31/test-vectors.json>>.

## Appendix A. Differences from RFC 9180

This specification is intended to be backwards-compatible with RFC 9180, in the sense that any behavior specified in both this document and RFC 9180 should specify identical behavior for any functionality that they both specify.

Within that constraint, the following list summarizes the major changes from RFC 9180:

- \* Incorporated fixes for all valid errata on RFC 9180.
- \* Updated the IANA considerations refer to existing registries.
- \* Added a framework for single-stage KDFs.
- \* Removed the Auth and AuthPSK modes.
- \* Extended the discussion of replay to cover considerations related to exported secrets.

## Appendix B. Acknowledgements

The authors would like to thank Joël Alwen, Jean-Philippe Aumasson, David Benjamin, Benjamin Beurdouche, Bruno Blanchet, Frank Denis, Stephen Farrell, Scott Fluhrer, Eduard Hauck, Scott Hollenbeck, Kevin Jacobs, Burt Kaliski, Eike Kiltz, Julia Len, John Mattsson, Christopher Patton, Doreen Riepel, Raphael Robert, Michael Rosenberg, Michael Scott, Martin Thomson, Steven Valdez, Riad Wahby, and other contributors in the CFRG for helpful feedback that greatly improved this document.

## Appendix C. Test Vectors

Each section below contains test vectors for a single HPKE ciphersuite and contains the following values:

1. Configuration information and private key material: This includes the mode, info string, HPKE ciphersuite identifiers (`kem_id`, `kdf_id`, `aead_id`), and all sender, recipient, and ephemeral key material. For each role `X`, where `X` is one of `S`, `R`, or `E`, as sender, recipient, and ephemeral, respectively, key pairs are generated as  $(skX, pkX) = \text{DeriveKeyPair}(ikmX)$ . Each key pair  $(skX, pkX)$  is written in its serialized form, where `skXm` = `SerializePrivateKey(skX)` and `pkXm` = `SerializePublicKey(pkX)`. For applicable modes, the shared PSK and PSK identifier are also included.
2. Context creation intermediate values and outputs: This includes the KEM outputs `enc` and `shared_secret` used to create the context, along with intermediate values `key_schedule_context` and `secret` computed in the `KeySchedule` function in Section 5.1. The outputs include the context values `key`, `base_nonce`, and `exporter_secret`.
3. Encryption test vectors: A fixed plaintext message is encrypted using different sequence numbers and AAD values using the context computed in (2). Each test vector lists the sequence number and corresponding nonce computed with `base_nonce`, the plaintext message `pt`, AAD `aad`, and output ciphertext `ct`.
4. Export test vectors: Several exported values of the same length with differing context parameters are computed using the context computed in (2). Each test vector lists the `exporter_context`, output length `L`, and resulting export value.

These test vectors are also available in JSON format at `[TestVectors]`.

### C.1. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM

#### C.1.1. Base Setup Information

```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
7268600d403fce431561aef583ee1613527cff655c1343f29812e66706df3234
pkEm:
37fda3567bdbd628e88668c3c8d7e97d1d1253b6d4ea6d44c150f741f1bf4431
skEm:
52c4a758a802cd8b936ecee314432798d5baf2d7e9235dc084ab1b9cfa2f736
ikmR:
6db9df30aa07dd42ee5e8181afdb977e538f5e1fec8a06223f33f7013e525037
pkRm:
3948cfe0ad1ddb695d780e59077195da6c56506b027329794ab02bca80815c4d
skRm:
4612c550263fc8ad58375df3f557aac531d26850903e55a9f23f21d8534e8ac8
enc:
37fda3567bdbd628e88668c3c8d7e97d1d1253b6d4ea6d44c150f741f1bf4431
shared_secret:
fe0e18c9f024ce43799ae393c7e8fe8fce9d218875e8227b0187c04e7d2ea1fc
key_schedule_context: 00725611c9d98c07c03f60095cd32d400d8347d45ed670
97bbad50fc56da742d07cb6cffde367bb0565ba28bb02c90744a20f5ef37f3052352
6106f637abb05449
secret:
12fff91991e93b48de37e7daddb52981084bd8aa64289c3788471d9a9712f397
key: 4531685d41d65f03dc48f6b8302c05b0
base_nonce: 56d890e5accaaf011cff4b7d
exporter_secret:
45ff1c2e220db587171952c0592d5f5ebe103f1561a2614e38f2ffd47e99e3f8
```

#### C.1.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 56d890e5accaaf011cff4b7d  
ct: f938558b5d72f1a23810b4be2ab4f84331acc02fc97bab53a52ae8218a355a9  
6d8770ac83d07bea87e13c512a

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 56d890e5accaaf011cff4b7c  
ct: af2d7e9ac9ae7e270f46ba1f975be53c09f8d875bdc8535458c2494e8a6eab25  
1c03d0c22a56b8ca42c2063b84

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 56d890e5accaaf011cff4b7f  
ct: 498dfcabd92e8acedc281e85af1cb4e3e31c7dc394a1ca20e173cb7251649158  
8d96a19ad4a683518973dcc180

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 56d890e5accaaf011cff4b79  
ct: 583bd32bc67a5994bb8ceaca813d369bca7b2a42408cddef5e22f880b631215a  
09fc0012bc69fccaa251c0246d

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 56d890e5accaaf011cff4b82  
ct: 7175db9717964058640a3a11fb9007941a5d1757fd1a6935c805c21af32505b  
f106deefec4a49ac38d71c9e0a

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 56d890e5accaaf011cff4a7d  
ct: 957f9800542b0b8891badb026d79cc54597cb2d225b54c00c5238c25d05c30e3  
fbeda97d2e0elaba483a2df9f2

#### C.1.1.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
3853fe2b4035195a573ffc53856e77058e15d9ea064de3e59f4961d0095250ee  
  
exporter_context: 00  
L: 32  
exported_value:  
2e8f0b54673c7029649d4eb9d5e33bf1872cf76d623ff164ac185da9e88c21a5  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
e9e43065102c3836401bed8c3c3c75ae46be1639869391d62c61f1ec7af54931
```

#### C.1.2. PSK Setup Information

```
mode: 1
kem_id: 32
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
78628c354e46f3e169bd231be7b2ff1c77aa302460a26dbfa15515684c00130b
pkEm:
0ad0950d9fb9588e59690b74f1237ecdf1d775cd60be2eca57af5a4b0471c91b
skEm:
463426a9fffb42bb17dbe6044b9abd1d4e4d95f9041cef0e99d7824eef2b6f588
ikmR:
d4a09d09f575fef425905d2ab396c1449141463f698f8efdb7accfaff8995098
pkRm:
9fed7e8c17387560e92cc6462a68049657246a09bfa8ade7aef5e589672016366
skRm:
c5eb01eb457fe6c6f57577c5413b931550a162c71a03ac8d196babbd4e5ce0fd
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc:
0ad0950d9fb9588e59690b74f1237ecdf1d775cd60be2eca57af5a4b0471c91b
shared_secret:
727699f009ffe3c076315019c69648366b69171439bd7dd0807743bde76986cd
key_schedule_context: 01e78d5cf6190d275863411ff5edd0dece5d39fa48e04e
ec1ed9b71be34729d18ccb6cffde367bb0565ba28bb02c90744a20f5ef37f3052352
6106f637abb05449
secret:
3728ab0b024b383b0381e432b47cced1496d2516957a76e2a9f5c8cb947afca4
key: 15026dba546e3ae05836fc7de5a7bb26
base_nonce: 9518635eba129d5ce0914555
exporter_secret:
3d76025dbbedc49448ec3f9080a1abab6b06e91c0b11ad23c912f043a0ee7655
```

#### C.1.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 9518635eba129d5ce0914555  
ct: e52c6fed7f758d0cf7145689f21bc1be6ec9ea097fef4e959440012f4feb73fb  
611b946199e681f4cfc34db8ea

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 9518635eba129d5ce0914554  
ct: 49f3b19b28a9ea9f43e8c71204c00d4a490ee7f61387b6719db765e948123b45  
b61633ef059ba22cd62437c8ba

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 9518635eba129d5ce0914557  
ct: 257ca6a08473dc851fde45afd598cc83e326ddd0abe1ef23baa3baa4dd8cde99  
fce2cle8ce687b0b47eadladc9

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 9518635eba129d5ce0914551  
ct: a71d73a2cd8128fcccbd328b9684d70096e073b59b40b55e6419c9c68ae21069  
c847e2a70f5d8fb821ce3dfb1c

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 9518635eba129d5ce09145aa  
ct: 55f84b030b7f7197f7d7d552365b6b932df5ec1abacd30241cb4bc4ccea27bd2  
b518766adfa0fb1b71170e9392

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 9518635eba129d5ce0914455  
ct: c5bf246d4a790a12dcc9eed5eae525081e6fb541d5849e9ce8abd92a3bc15517  
76bea16b4a518f23e237c14b59

#### C.1.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
dff17af354c8b41673567db6259fd6029967b4e1aad13023c2ae5df8f4f43bf6
```

```
exporter_context: 00  
L: 32  
exported_value:  
6a847261d8207fe596befb52928463881ab493da345b10e1dcc645e3b94e2d95
```

```
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
8aff52b45a1be3a734bc7a41e20b4e055ad4c4d22104b0c20285a7c4302401cd
```

C.2. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305

C.2.1. Base Setup Information

```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
909a9b35d3dc4713a5e72a4da274b55d3d3821a37e5d099e74a647db583a904b
pkEm:
1afa08d3dec047a643885163f1180476fa7ddb54c6a8029ea33f95796bf2ac4a
skEm:
f4ec9b33b792c372c1d2c2063507b684ef925b8c75a42dbcbf57d63ccd381600
ikmR:
1ac01f181fdf9f352797655161c58b75c656a6cc2716dcb66372da835542e1df
pkRm:
4310ee97d88cc1f088a5576c77ab0cf5c3ac797f3d95139c6c84b5429c59662a
skRm:
8057991eef8f1f1af18f4a9491d16a1ce333f695d4db8e38da75975c4478e0fb
enc:
1afa08d3dec047a643885163f1180476fa7ddb54c6a8029ea33f95796bf2ac4a
shared_secret:
0bbe78490412b4bbea4812666f7916932b828bba79942424abb65244930d69a7
key_schedule_context: 00431df6cd95e11ff49d7013563baf7f11588c75a6611e
e2a4404a49306ae4cfc5b69c5718a60cc5876c358d3f7fc31ddb598503f67be58ea1
e798c0bb19eb9796
secret:
5b9cd775e64b437a2335cf499361b2e0d5e444d5cb41a8a53336d8fe402282c6
key:
ad2744de8e17f4ebba575b3f5f5a8fa1f69c2a07f6e7500bc60ca6e3e3ec1c91
base_nonce: 5c4d98150661b848853b547f
exporter_secret:
a3b010d4994890e2c6968a36f64470d3c824c8f5029942feb11e7a74b2921922
```

#### C.2.1.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 5c4d98150661b848853b547f  
ct: 1c5250d8034ec2b784ba2cfd69dbdb8af406cfe3ff938e131f0def8c8b60b4db  
21993c62ce81883d2dd1b51a28

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 5c4d98150661b848853b547e  
ct: 6b53c051e4199c518de79594e1c4ab18b96f081549d45ce015be002090bb119e  
85285337cc95ba5f59992dc98c

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 5c4d98150661b848853b547d  
ct: 71146bd6795ccc9c49ce25dda112a48f202ad220559502cef1f34271e0cb4b02  
b4f10ecac6f48c32f878fae86b

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 5c4d98150661b848853b547b  
ct: 63357a2aa291f5a4e5f27db6baa2af8cf77427c7c1a909e0b37214dd47db122b  
b153495ff0b02e9e54a50dbe16

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 5c4d98150661b848853b5480  
ct: 18ab939d63ddec9f6ac2b60d61d36a7375d2070c9b683861110757062c52b888  
0a5f6b3936da9cd6c23ef2a95c

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 5c4d98150661b848853b557f  
ct: 7a4a13e9ef23978e2c520fd4d2e757514ae160cd0cd05e556ef692370ca53076  
214c0c40d4c728d6ed9e727a5b

#### C.2.1.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
4bbd6243b8bb54cec311fac9df81841b6fd61f56538a775e7c80a9f40160606e  
  
exporter_context: 00  
L: 32  
exported_value:  
8cldf14732580e5501b00f82b10a1647b40713191b7c1240ac80e2b68808ba69  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
5acb09211139c43b3090489a9da433e8a30ee7188ba8b0a9a1ccf0c229283e53
```

#### C.2.2. PSK Setup Information

```
mode: 1
kem_id: 32
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
35706a0b09fb26fb45c39c2f5079c709c7cf98e43afa973f14d88ece7e29c2e3
pkEm:
2261299c3f40a9afc133b969a97f05e95be2c514e54f3de26cbe5644ac735b04
skEm:
0c35fdf49df7aa01cd330049332c40411ebba36e0c718ebc3edf5845795f6321
ikmR:
26b923eade72941c8a85b09986cdfa3f1296852261adedc52d58d2930269812b
pkRm:
13640af826b722fc04feaa4de2f28fbd5ecc03623b317834e7ff4120dbe73062
skRm:
77d114e0212be51cb1d76fa99dd41cfd4d0166b08caa09074430a6c59ef17879
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc:
2261299c3f40a9afc133b969a97f05e95be2c514e54f3de26cbe5644ac735b04
shared_secret:
4be079c5e77779d0215b3f689595d59e3e9b0455d55662d1f3666ec606e50ea7
key_schedule_context: 016870c4c76ca38ae43efbec0f2377d109499d7ce73f4a
9elec37f21d3d063b97cb69c5718a60cc5876c358d3f7fc31ddb598503f67be58ea1
e798c0bb19eb9796
secret:
16974354c497c9bd24c000ceed693779b604f1944975b18c442d373663f4a8cc
key:
600d2fdb0313a7e5c86a9ce9221cd95bed069862421744cfb4ab9d7203a9c019
base_nonce: 112e0465562045b7368653e7
exporter_secret:
73b506dc8b6b4269027f80b0362def5cbb57ee50eed0c2873dac9181f453c5ac
```

#### C.2.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 112e0465562045b7368653e7  
ct: 4a177f9c0d6f15cfd533fb65bf84aecdc6ab16b8b85b4cf65a370e07fcd1d78d28fb073214525276f4a89608ff

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 112e0465562045b7368653e6  
ct: 5c3cabae2f0b3e124d8d864c116fd8f20f3f56fda988c3573b40b09997fd6c769e77c8eda6cda4f947f5b704a8

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 112e0465562045b7368653e5  
ct: 14958900b44bdae9cbe5a528bf933c5c990dbb8e282e6e495adf8205d19da9eb270e3a6f1e0613ab7e757962a4

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 112e0465562045b7368653e3  
ct: c2a7bc09ddb853cf2effb6e8d058e346f7fe0fb3476528c80db6b698415c5f8c50b68a9a355609e96d2117f8d3

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 112e0465562045b736865318  
ct: 2414d0788e4bc39a59a26d7bd5d78e111c317d44c37bd5a4c2a1235f2ddc2085c487d406490e75210c958724a7

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 112e0465562045b7368652e7  
ct: c567aelc3f0f75abeldd9e4532b422600ed4a6e5b9484dafb1e43ab9f5fd662b28c00e2e81d3cde955dae7e218

#### C.2.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
813c1bfc516c99076ae0f466671f0ba5ff244a41699f7b2417e4c59d46d39f40  
  
exporter_context: 00  
L: 32  
exported_value:  
2745cf3d5bb65c333658732954ee7af49eb895ce77f8022873a62a13c94cb4e1  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
ad40e3ae14f21c99bfdebc20ae14ab86f4ca2dc9a4799d200f43a25f99fa78ae
```

C.3. DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, AES-128-GCM

C.3.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
4270e54fffd08d79d5928020af4686d8f6b7d35dbe470265f1f5aa22816ce860e
pkEm: 04a92719c6195d5085104f469a8b9814d5838ff72b60501e2c4466e5e67b32
5ac98536d7b61a1af4b78e5b7f951c0900be863c403ce65c9bfc9382657222d18c4
skEm:
4995788ef4b9d6132b249ce59a77281493eb39af373d236a1fe415cb0c2d7beb
ikmR:
668b37171f1072f3cf12ea8a236a45df23fc13b82af3609ad1e354f6ef817550
pkRm: 04fe8c19ce0905191ebc298a9245792531f26f0cece2460639e8bc39cb7f70
6a826a779b4cf969b8a0e539c7f62fb3d30ad6aa8f80e30f1d128aafd68a2ce72ea0
skRm:
f3ce7fdae57e1a310d87f1ebbde6f328be0a99cdbcadf4d6589cf29de4b8ffd2
enc: 04a92719c6195d5085104f469a8b9814d5838ff72b60501e2c4466e5e67b325
ac98536d7b61a1af4b78e5b7f951c0900be863c403ce65c9bfc9382657222d18c4
shared_secret:
c0d26aeab536609a572b07695d933b589dcf363ff9d93c93adea537aeabb8cb8
key_schedule_context: 00b88d4e6d91759e65e87c470e8b9141113e9ad5f0c8ce
efcle088c82e6980500798e486f9c9c09c9b5c753ac72d6005de254c607d1b534ed1
1d493aelcld9ac85
secret:
2eb7b6bf138f6b5aff857414a058a3f1750054a9balf72c2cf0684a6f20b10e1
key: 868c066ef58aae6dc589b6cfdd18f97e
base_nonce: 4e0bc5018beba4bf004cca59
exporter_secret:
14ad94af484a7ad3ef40e9f3be99ecc6fa9036df9d4920548424df127ee0d99f
```

#### C.3.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 4e0bc5018beba4bf004cca59  
ct: 5ad590bb8baa577f8619db35a36311226a896e7342a6d836d8b7bcd2f20b6c7f  
9076ac232e3ab2523f39513434

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 4e0bc5018beba4bf004cca58  
ct: fa6f037b47fc21826b610172ca9637e82d6e5801eb31cbd3748271affd4ecb06  
646e0329cbdf3c3cd655b28e82

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 4e0bc5018beba4bf004cca5b  
ct: 895cabfac50ce6c6eb02ffe6c048bf53b7f7be9a91fc559402cbc5b8dcaeb52b  
2ccc93e466c28fb55fed7a7fec

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 4e0bc5018beba4bf004cca5d  
ct: 8787491ee8df99bc99a246c4b3216d3d57ab5076e18fa27133f520703bc70ec9  
99dd36ce042e44f0c3169a6a8f

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 4e0bc5018beba4bf004ccaa6  
ct: 2ad71c85bf3f45c6eca301426289854b31448bcf8a8ccb1deef3ebd87f60848a  
a53c538c30a4dac71d619ee2cd

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 4e0bc5018beba4bf004ccb59  
ct: 10f179686aa2caec1758c8e554513f16472bd0a11e2a907dde0b212cbe87d74f  
367f8ffe5e41cd3e9962a6afb2

#### C.3.1.2. Exported Values

```
exporter_context:
L: 32
exported_value:
5e9bc3d236e1911d95e65b576a8a86d478fb827e8bdfe77b741b289890490d4d

exporter_context: 00
L: 32
exported_value:
6cff87658931bda83dc857e6353efe4987a201b849658d9b047aab4cf216e796

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
d8f1ea7942adbba7412c6d431c62d01371ea476b823eb697e1f6e6cae1dab85a
```

### C.3.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 1
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
2afa611d8b1a7b321c761b483b6a053579afa4f767450d3ad0f84a39fda587a6
pkEm: 04305d35563527bce037773d79a13deabed0e8e7cde61eecee403496959e89
e4d0ca701726696d1485137ccb5341b3c1c7aaee90a4a02449725e744b1193b53b5f
skEm:
57427244f6cc016cddf1c19c8973b4060aa13579b4c067fd5d93a5d74e32a90f
ikmR:
d42ef874c1913d9568c9405407c805baddaffd0898a00f1e84e154fa787b2429
pkRm: 040d97419ae99f13007a93996648b2674e5260a8ebd2b822e84899cd52d874
46ea394ca76223b76639eccdf00e1967db10ade37db4e7db476261fcc8df97c5ffd1
skRm:
438d8bcef33b89e0e9ae5eb0957c353c25a94584b0dd59c991372a75b43cb661
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04305d35563527bce037773d79a13deabed0e8e7cde61eecee403496959e89e
4d0ca701726696d1485137ccb5341b3c1c7aaee90a4a02449725e744b1193b53b5f
shared_secret:
2e783ad86albeae03b5749e0f3f5e9bb19cb7eb382f2fb2dd64c99f15ae0661b
key_schedule_context: 01b873cdf2dff4c1434988053b7a775e980dd2039ea24f
950b26b056ccedcb933198e486f9c9c09c9b5c753ac72d6005de254c607d1b534ed1
1d493aelc1d9ac85
secret:
f2f534e55931c62eeb2188c1f53450354a725183937e68c85e68d6b267504d26
key: 55d9eb9d26911d4c514a990fa8d57048
base_nonce: b595dc6b2d7e2ed23af529b1
exporter_secret:
895a723aleab809804973a53c0ee18ece29b25a7555a4808277ad2651d66d705
```

#### C.3.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: b595dc6b2d7e2ed23af529b1  
ct: 90c4deb5b75318530194e4bb62f890b019b1397bbf9d0d6eb918890e1fb2be1a  
c2603193b60a49c2126b75d0eb

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: b595dc6b2d7e2ed23af529b0  
ct: 9e223384a3620f4a75b5a52f546b7262d8826dea18db5a365feb8b997180b22d  
72dc1287f7089a1073a7102c27

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: b595dc6b2d7e2ed23af529b3  
ct: adf9f6000773035023be7d415e13f84c1cb32a24339a32eb81df02be9ddc6abc  
880dd81cceb7c1d0c7781465b2

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: b595dc6b2d7e2ed23af529b5  
ct: 1f4cc9b7013d65511b1f69c050b7bd8bbd5a5c16ece82b238fec4f30ba2400e7  
ca8ee482ac5253cffb5c3dc577

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: b595dc6b2d7e2ed23af5294e  
ct: cdc541253111ed7a424eea5134dc14fc5e8293ab3b537668b8656789628e4589  
4e5bb873c968e3b7cdccb654a4

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: b595dc6b2d7e2ed23af528b1  
ct: faf985208858b1253b97b60aec28bc18737b58d1242370e7703ec33b73a4c31  
alafee300e349adef9015bbbfd

#### C.3.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
a115a59bf4dd8dc49332d6a0093af8efca1bcbfd3627d850173f5c4a55d0c185  
  
exporter_context: 00  
L: 32  
exported_value:  
4517eaede0669b16aac7c92d5762dd459c301fa10e02237cd5aeb9be969430c4  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
164e02144d44b607a7722e58b0f4156e67c0c2874d74cf71da6ca48a4cbdc5e0
```

C.4. DHKEM(P-256, HKDF-SHA256), HKDF-SHA512, AES-128-GCM

C.4.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
4ab11a9dd78c39668f7038f921ffc0993b368171d3ddde8031501ee1e08c4c9a
pkEm: 0493ed86735bdfb978cc055c98b45695ad7ce61ce748f4dd63c525a3b8d53a
15565c6897888070070c1579db1f86aaa56deb8297e64db7e8924e72866f9a472580
skEm:
2292bf14bb6e15b8c81a0f45b7a6e93e32d830e48cca702e0affcfb4d07e1b5c
ikmR:
ea9ff7cc5b2705b188841c7ace169290ff312a9cb31467784ca92d7a2e6e1be8
pkRm: 04085aa5b665dc3826f9650ccbcc471be268c8ada866422f739e2d531d4a88
18a9466bc6b449357096232919ec4fe9070ccbacc4aac30f4a1a53efcf7af90610edd
skRm:
3ac8530ad1b01885960fab38cf3cdc4f7aef121eaa239f222623614b4079fb38
enc: 0493ed86735bdfb978cc055c98b45695ad7ce61ce748f4dd63c525a3b8d53a1
5565c6897888070070c1579db1f86aaa56deb8297e64db7e8924e72866f9a472580
shared_secret:
02f584736390fc93f5b4ad039826a3fa08e9911bd1215a3db8e8791ba533cafd
key_schedule_context: 005b8a3617af7789ee716e7911c7e77f84cdc4cc46e60f
b7e19e4059f9aeadc00585e26874d1ddde76e551a7679cd47168c466f6e1f705cc93
74c192778a34fcd5ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea
4891c9a2a87a4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1
252ef4f9
secret: 0c7acdab61693f936c4c1256c78e7be30eebfe466812f9cc49f0b58dc970
328dfc03ea359be0250a471b1635a193d2dfa8cb23c90aa2e25025b892a725353eeb
key: 090ca96e5f8aa02b69fac360da50ddf9
base_nonce: 9c995e621bf9a20c5ca45546
exporter_secret: 4a7abb2ac43e6553f129b2c5750a7e82d149a76ed56dc342d7b
ca61e26d494f4855dff0d0165f27ce57756f7f16baca006539bb8e4518987ba61048
0ac03efa8
```

#### C.4.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 9c995e621bf9a20c5ca45546  
ct: d3cf4984931484a080f74c1bb2a6782700dc1fef9abe8442e44a6f09044c8890  
7200b332003543754eb51917ba

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 9c995e621bf9a20c5ca45547  
ct: d14414555a47269dfead9fbf26abb303365e40709a4ed16eae1f2070f1ddeb  
1bdd94d9e41186f124e0acc62d

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 9c995e621bf9a20c5ca45544  
ct: 9bba13cade5c4069707ba91a61932e2cbdda2d9c7bdc33515aa01dd0e0f7e9  
d3579bf4016dec37da4aafa800

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 9c995e621bf9a20c5ca45542  
ct: a531c0655342be013bf32112951f8df1da643602f1866749519f5dcb09cc6843  
2579de305a77e6864e862a7600

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 9c995e621bf9a20c5ca455b9  
ct: be5da649469efbad0fb950366a82a73fefeda5f652ec7d3731fac6c4ffa21a70  
04d2ab8a04e13621bd3629547d

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 9c995e621bf9a20c5ca45446  
ct: 62092672f5328a0dde095e57435edf7457ace60b26ee44c9291110ec135cb0e1  
4b85594e4fea11247d937deb62

#### C.4.1.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
a32186b8946f61aeead1c093fe614945f85833b165b28c46bf271abf16b57208  
  
exporter_context: 00  
L: 32  
exported_value:  
84998b304a0ea2f11809398755f0abd5f9d2c141d1822def79dd15c194803c2a  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
93fb9411430b2cfa2cf0bed448c46922a5be9beff20e2e621df7e4655852edbc
```

#### C.4.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 3
aead_id: 1
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
c11d883d6587f911d2ddbc2a0859d5b42fb13bf2c8e89ef408a25564893856f5
pkEm: 04a307934180ad5287f95525fe5bc6244285d7273c15e061f0f2efb211c350
57f3079f6e0abae200992610b25f48b63aacfc669106dde8aa023feed301901371
skEm:
a5901ff7d6931959c2755382ea40a4869b1dec3694ed3b009dda2d77dd488f18
ikmR:
75bfc2a3a3541170a54c0b06444e358d0ee2b4fb78a401fd399a47a33723b700
pkRm: 043f5266fba0742db649e1043102b8a5afd114465156719cea90373229aabd
d84d7f45dabfc1f55664b888a7e86d594853a6cccdc9b189b57839cbb3b90b55873
skRm:
bc6f0b5e22429e5ff47d5969003f3cae0f4fec50e23602e880038364f33b8522
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04a307934180ad5287f95525fe5bc6244285d7273c15e061f0f2efb211c3505
7f3079f6e0abae200992610b25f48b63aacfc669106dde8aa023feed301901371
shared_secret:
2912aacc6eae71ff715ea50f6ef3a6637856b2a4c58ea61e0c3fc159e3bc16
key_schedule_context: 01713f73042575ceb7d132f0cc4338523f8eae95c80a74
9f7cf3eb9436ff1c612ca62c37df27ca46d2cc162445a92c5f5fdc57bcde129ca7b1
f284b0c12297c037ca221d77e229a9d11b654de7942d685069c633b2362ce3b3d8ea
4891c9a2a87a4eb7cdb289ba5e2ecbf8cd2c8498bb4a383dc021454d70d46fcbbad1
252ef4f9
secret: ff2051d2128d5f3078de867143e076262ce1d0aecafc3fff3d607f1eaff0
5345c7d5ffcb3202cdec3d1a2f7da20592a237747b6e855390cbe2109d3e6ac70c2
key: 0b910ba8d9cfa17e5f50c211cb32839a
base_nonce: 0c29e714eb52de5b7415a1b7
exporter_secret: 50c0a182b6f94b4c0bd955c4aa20df01f282cc12c43065a0812
fe4d4352790171ed2b2c4756ad7f5a730ba336c8f1edd0089d8331192058c385bae3
9c7cc8b57
```

#### C.4.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 0c29e714eb52de5b7415a1b7  
ct: 57624b6e320d4aba0afd11f548780772932f502e2ba2a8068676b2a0d3b5129a45b9faa88de39e8306da41d4cc

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 0c29e714eb52de5b7415a1b6  
ct: 159d6b4c24bacaf2f5049b7863536d8f3ffede76302dace42080820fa51925d4e1c72a64f87b14291a3057e00a

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 0c29e714eb52de5b7415a1b5  
ct: bd24140859c99bf0055075e9c460032581dd1726d52cf980d308e9b20083ca62e700b17892bcf7fa82bac751d0

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 0c29e714eb52de5b7415a1b3  
ct: 93ddd55f82e9aaaa3cfc06840575f09d80160b20538125c2549932977d1238dde8126a4a91118faf8632f62cb8

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 0c29e714eb52de5b7415a148  
ct: 377a98a3c34bf716581b05a6b3fdc257f245856384d5f2241c8840571c52f5c85c21138a4a81655edab8fe227d

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 0c29e714eb52de5b7415a0b7  
ct: cc161f5a179831d456d119d2f2c19a6817289c75d1c61cd37ac8a450acd9efba02e0ac00d128c17855931ff69a

#### C.4.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
8158bea21a6700d37022bb7802866edca30ebf2078273757b656ef7fc2e428cf  
  
exporter_context: 00  
L: 32  
exported_value:  
6a348ba6e0e72bb3ef22479214a139ef8dac57be34509a61087a12565473da8d  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
2f6d4f7a18ec48de1ef4469f596aada4afdf6d79b037ed3c07e0118f8723bffc
```

C.5. DHKEM(P-256, HKDF-SHA256), HKDF-SHA256, ChaCha20Poly1305

C.5.1. Base Setup Information

```
mode: 0
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
f1f1a3bc95416871539ecb51c3a8f0cf608afb40fbbe305c0a72819d35c33f1f
pkEm: 04c07836a0206e04e31d8ae99bfd549380b072a1b1b82e563c935c09582782
4fc1559eac6fb9e3c70cd3193968994e7fe9781aa103f5b50e934b5b2f387e381291
skEm:
7550253e1147aae48839c1f8af80d2770fb7a4c763afe7d0afa7e0f42a5b3689
ikmR:
61092f3f56994dd424405899154a9918353e3e008171517ad576b900ddb275e7
pkRm: 04a697bffde9405c992883c5c439d6cc358170b51af72812333b015621dc0f
40bad9bb726f68a5c013806a790ec716ab8669f84f6b694596c2987cf35baba2a006
skRm:
a4d1c55836aa30f9b3fbb6ac98d338c877c2867dd3a77396d13f68d3ab150d3b
enc: 04c07836a0206e04e31d8ae99bfd549380b072a1b1b82e563c935c095827824
fc1559eac6fb9e3c70cd3193968994e7fe9781aa103f5b50e934b5b2f387e381291
shared_secret:
806520f82ef0b03c823b7fc524b6b55a088f566b9751b89551c170f4113bd850
key_schedule_context: 00b738cd703db7b4106e93b4621e9a19c89c838e559642
40e5d3f331aaf8b0d58b2e986eal671b61cf45eec134dac0bae58ec6f63e790b140
0b47c33038b0269c
secret:
fe891101629aa355aad68eff3cc5170d057eca0c7573f6575e91f9783e1d4506
key:
a8f45490a92a3b04d1dbf6cf2c3939ad8bfc9bfc97c04bffe116730c9dfe3fc
base_nonce: 726b4390ed2209809f58c693
exporter_secret:
4f9bd9b3a8db7d7c3a5b9d44fdc1f6e37d5d77689ade5ec44a7242016e6aa205
```

#### C.5.1.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 726b4390ed2209809f58c693  
ct: 6469c41c5c81d3aa85432531ecf6460ec945bde1eb428cb2fedf7a29f5a685b4  
ccb0d057f03ea2952a27bb458b

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 726b4390ed2209809f58c692  
ct: f1564199f7e0e110ec9c1bcdde332177fc35c1adf6e57f8d1df24022227ffa87  
16862dbda2b1dc546c9d114374

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 726b4390ed2209809f58c691  
ct: 39de89728bcb774269f882af8dc5369e4f3d6322d986e872b3a8d074c7c18e85  
49ff3f85b6d6592ff87c3f310c

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 726b4390ed2209809f58c697  
ct: bc104a14fbede0cc79eeb826ea0476ce87b9c928c36e5e34dc9b6905d91473ec  
369a08b1a25d305dd45c6c5f80

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 726b4390ed2209809f58c66c  
ct: 8f2814a2c548b3be50259713c6724009e092d37789f6856553d61df23ebc0792  
35f710e6af3c3ca6eaba7c7c6c

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 726b4390ed2209809f58c793  
ct: b45b69d419a9be7219d8c94365b89ad6951caf4576ea4774ea40e9b7047a09d6  
537d1aa2f7c12d6ae4b729b4d0

#### C.5.1.2. Exported Values

```
exporter_context:
L: 32
exported_value:
9b13c510416ac977b553bf1741018809c246a695f45eff6d3b0356dbefe1e660

exporter_context: 00
L: 32
exported_value:
6c8b7be3a20a5684edecb4253619d9051ce8583baf850e0cb53c402bdcaf8ebb

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
477a50d804c7c51941f69b8e32fe8288386ee1a84905fe4938d58972f24ac938
```

#### C.5.2. PSK Setup Information

```
mode: 1
kem_id: 16
kdf_id: 1
aead_id: 3
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
ela4e1d50c4bfcf890f2b4c7d6b2d2aca61368eddc3c84162df2856843e1057a
pkEm: 04f336578b72ad7932fe867cc4d2d44a718a318037a0ec271163699cee653f
a805c1fec955e562663e0c2061bb96a87d78892bff0cc0bad7906c2d998ebela7246
skEm:
7d6e4e006cee68af9b3fdd583a0ee8962df9d59fab029997ee3f456cbc857904
ikmR:
ee51dec304abf993ef8fd52aacdd3b539108bbf6e491943266c1de89ec596a17
pkRm: 041eb8f4f20ab72661af369ff3231a733672fa26f385ffb959fd1bae46bfda
43ad55e2d573b880831381d9367417f554ce5b2134fbba5235b44db465feffc6189e
skRm:
12ecde2c8bc2d5d7ed2219c71f27e3943d92b344174436af833337c557c300b3
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 04f336578b72ad7932fe867cc4d2d44a718a318037a0ec271163699cee653fa
805c1fec955e562663e0c2061bb96a87d78892bff0cc0bad7906c2d998ebela7246
shared_secret:
ac4f260dce4db6bf45435d9c92c0e11cfdd93743bd3075949975974cc2b3d79e
key_schedule_context: 01622b72afcc3795841596c67ea74400ca3b029374d7d5
640bda367c5d67b3fbeb2e986ealc671b61cf45eec134dac0bae58ec6f63e790b140
0b47c33038b0269c
secret:
858c8087a1c056db5811e85802f375bb0c19b9983204a1575de4803575d23239
key:
6d61cb330b7771168c8619498e753f16198aad9566d1f1c6c70e2bc1a1a8b142
base_nonce: 0de7655fb65e1cd51a38864e
exporter_secret:
754ca00235b245e72d1f722a7718e7145bd113050a2aa3d89586d4cb7514bfdb
```

#### C.5.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 0de7655fb65e1cd51a38864e  
ct: 21433eaff24d7706f3ed5b9b2e709b07230e2b11df1f2b1fe07b3c70d5948a53  
d6fa5c8bed194020bd9df0877b

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 0de7655fb65e1cd51a38864f  
ct: c74a764b4892072ea8c2c56b9bcd46c7f1e9ca8cb0a263f8b40c2ba59ac9c857  
033f176019562218769d3e0452

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 0de7655fb65e1cd51a38864c  
ct: dc8cd68863474d6e9cbb6a659335a86a54e036249d41acf909e738c847ff2bd3  
6fe3fcacda4ededa7032c0a220

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 0de7655fb65e1cd51a38864a  
ct: cd54a8576353b1b9df366cb0cc042e46eef6f4cf01e205fe7d47e306b2fdd90f  
7185f289a26c613ca094e3be10

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 0de7655fb65e1cd51a3886b1  
ct: 6324570c9d542c70c7e70570c1d8f4c52a89484746bf0625441890ededcc80c2  
4ef2301c38bfd34d689d19f67d

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 0de7655fb65e1cd51a38874e  
ct: 1ea6326c8098ed0437a553c466550114fb2ca1412cca7de98709b9ccdf19206e  
52c3d39180e2cf62b3e9f4baf4

#### C.5.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
530bbc2f68f078dccc89cc371b4f4ade372c9472bafe4601a8432cbb934f528d  
  
exporter_context: 00  
L: 32  
exported_value:  
6e25075ddcc528c90ef9218f800ca3dfelb8ff4042de5033133adb8bd54c401d  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
6f6fbd0d1c7733f796461b3235a856cc34f676fe61ed509dfc18fa16efe6be78
```

C.6. DHKEM(P-521, HKDF-SHA512), HKDF-SHA512, AES-256-GCM

C.6.1. Base Setup Information

```
mode: 0
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: 7f06ab8215105fc46aceeb2e3dc5028b44364f960426eb0d8e4026c2f8b5d7
e7a986688f1591abf5ab753c357a5d6f0440414b4ed4ede71317772ac98d9239f709
04
pkEm: 040138b385ca16bb0d5fa0c0665fbbd7e69e3ee29f63991d3e9b5fa740aab8
900aaeed46ed73a49055758425a0ce36507c54b29cc5b85a5cee6bae0cf1c21f2731
ece2013dc3fb7c8d21654bb161b463962ca19e8c654ff24c94dd2898de12051f1ed0
692237fb02b2f8d1dc1c73e9b366b529eb436e98a996ee522aef863dd5739d2f29b0
skEm: 014784c692da35df6ecde98ee43ac425dbdd0969c0c72b42f2e708ab9d5354
15a8569bdacfcc0a114c85b8e3f26acf4d68115f8c91a66178cddb03b7bcc5291e37
4b
ikmR: 2ad954bbe39b7122529f7dde780bff626cd97f850d0784a432784e69d86ecc
aade43b6c10a8ffdb94bf943c6da479db137914ec835a7e715e36e45e29b587bab3b
f1
pkRm: 0401b45498c1714e2dce167d3caf162e45e0642afc7ed435df7902ccae0e84
ba0f7d373f646b7738bbbdcalled91bdeae3cdcba3301f2457be452f271fa6837580
e661012af49583a62e48d44bed350c7118c0d8dc861c238c72a2bda17f64704f464b
57338e7f40b60959480c0e58e6559b190d81663ed816e523b6b6a418f66d2451ec64
skRm: 01462680369ae375e4b3791070a7458ed527842f6a98a79ff5e0d4cbde83c2
7196a3916956655523a6a2556a7af62c5cadabe2ef9da3760bb21e005202f7b24628
47
enc: 040138b385ca16bb0d5fa0c0665fbbd7e69e3ee29f63991d3e9b5fa740aab89
00aaeed46ed73a49055758425a0ce36507c54b29cc5b85a5cee6bae0cf1c21f2731e
ce2013dc3fb7c8d21654bb161b463962ca19e8c654ff24c94dd2898de12051f1ed06
92237fb02b2f8d1dc1c73e9b366b529eb436e98a996ee522aef863dd5739d2f29b0
shared_secret: 776ab421302f6eff7d7cb5cbladaea0cd50872c71c2d63c30c4f1
d5e43653336fef33b103c67e7a98add2d3b66e2fda95b5b2a667aa9dac7e59cc1d46
d30e818
key_schedule_context: 0083a27c5b2358ab4daelb2f5d8f57f10cccc822a4733
26f543f239a70aee46347324e84e02d7651a10d08fb3dda739d22d50c53fbfa8122b
aacd0f9ae5913072ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d67753
08c3d3faa75dd64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692
e85b09a4
secret: 49fd9f53b0f93732555b2054edfdc0e3101000d75df714b98ce5aa295a37
f1b18dfa86a1c37286d805d3ea09a20b72f93c21e83955a1f01eb7c5ead563d21e7
key:
751e346ce8f0ddb2305c8a2a85c70d5cf559c53093656be636b9406d4d7d1b70
base_nonce: 55ff7a7d739c69f44b25447b
exporter_secret: e4ff9dfbc732a2b9c75823763c5ccc954a2c0648fc6de80a585
81252d0ee3215388a4455e69086b50b87eb28c169a52f42e71de4ca61c920e7bd24c
95cc3f992
```

#### C.6.1.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 55ff7a7d739c69f44b25447b  
ct: 170f8beddfe949b75ef9c387e201baf4132fa7374593dfafa90768788b7b2b20  
0aafcc6d80ea4c795a7c5b841a

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 55ff7a7d739c69f44b25447a  
ct: d9ee248e220ca24ac00bbbe7e221a832e4f7fa64c4fbab3945b6f3af0c5ecd5e  
16815b328be4954a05fd352256

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 55ff7a7d739c69f44b254479  
ct: 142cf1e02d1f58d9285f2af7dcfa44f7c3f2d15c73d460c48c6e0e506a3144ba  
e35284e7e221105b61d24e1c7a

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 55ff7a7d739c69f44b25447f  
ct: 3bb3a5a07100e5a12805327bf3b152df728b1c1be75a9fd2cb2bf5eac0cca1fb  
80addb37eb2a32938c7268e3e5

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 55ff7a7d739c69f44b254484  
ct: 4f268d0930f8d50b8fd9d0f26657ba25b5cb08b308c92e33382f369c768b558e  
113ac95a4c70dd60909ad1adc7

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 55ff7a7d739c69f44b25457b  
ct: dbbfc44ae037864e75f136e8b4b4123351d480e6619ae0e0ae437f036f2f8f1e  
f677686323977a1ccbb4b4f16a

#### C.6.1.2. Exported Values

```
exporter_context:
L: 32
exported_value:
05e2e5bd9f0c30832b80a279ff211cc65eceb0d97001524085d609ead60d0412

exporter_context: 00
L: 32
exported_value:
fca69744bb537f5b7a1596dbf34eaa8d84bf2e3ee7f1a155d41bd3624aa92b63

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
f389beaac6fcf6c0d9376e20f97e364f0609a88f1bc76d7328e9104df8477013
```

### C.6.2. PSK Setup Information

```
mode: 1
kem_id: 18
kdf_id: 3
aead_id: 2
info: 4f6465206f6e2061204772656369616e2055726e
ikmE: f3ebfa9a69a924e672114fcd9e06fa9559e937f7eccce4181a2b506df53dbe
514be12f094bb28e01de19dd345b4f7ede5ad7eaa6b9c3019592ec68eaae9a14732c
e0
pkEm: 040085eff0835cc84351f32471d32aa453cdc1f6418eaaecf1c2824210eb1d
48d0768b368110fab21407c324b8bb4bec63f042cfa4d0868d19b760eb4beba1bff7
93b30036d2c614d55730bd2a40c718f9466faf4d5f8170d22b6df98dfe0c067d02b3
49ae4a142e0c03418f0a1479ff78a3db07ae2c2e89e5840f712c174ba2118e90fdcb
skEm: 012e5cfe0daf5fe2a1cd617f4c4bae7c86f1f527b3207f115e262a98cc6526
8ec88cb8645aec73b7aa0a472d0292502d1078e762646e0c093cf873243d12c39915
f6
ikmR: a2a2458705e278e574f835effecd18232f8a4c459e7550a09d44348ae5d3b1
ea9d95c51995e657ad6f7cae659f5e186126a471c017f8f5e41da9eba74d4e0473e1
79
pkRm: 04006917e049a2be7e1482759fb067ddb94e9c4f7f5976f655088dec452466
14ff924ed3b385fc2986c0ecc39d14f907bf837d7306aada59dd5889086125ecd038
ead400603394b5d81f89ebfd556a898cc1d6a027e143d199d3db845cb91c5289fb26
c5ff80832935b0e8dd08d37c6185a6f77683347e472d1edb6daa6bd7652fea628fae
skRm: 011bafd9c7a52e3e71afbdab0d2f31b03d998a0dc875dd7555c63560e142bd
e264428de03379863b4ec6138f813fa009927dc5d15f62314c56d4e7ff2b485753eb
72
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc: 040085eff0835cc84351f32471d32aa453cdc1f6418eaaecf1c2824210eb1d4
8d0768b368110fab21407c324b8bb4bec63f042cfa4d0868d19b760eb4beba1bff79
3b30036d2c614d55730bd2a40c718f9466faf4d5f8170d22b6df98dfe0c067d02b34
```

```
9ae4a142e0c03418f0a1479ff78a3db07ae2c2e89e5840f712c174ba2118e90fdcb
shared_secret: 0d52de997fdaa4797720e8b1bebd3df3d03c4cf38cc8c1398168d
36c3fc7626428c9c254dd3f9274450909c64a5b3acbe45e2d850a2fd69ac0605fe5c
8a057a5
key_schedule_context: 0124497637cf18d6fbcc16e9f652f00244c981726f293b
b7819861e85e50c94f0be30e022ab081e18e6f299fd3d3d976a4bc590f85bc7711bf
ce32eela7fb1c154ef45baa1f3a4b169e141feb957e48d03f28c837d8904c3d67753
08c3d3faa75dd64adfa44e1a1141edf9349959b8f8e5291cbdc56f62b0ed6527d692
e85b09a4
secret: 2cf425e26f65526afc0634a3dba4e28d980c1015130ce07c2ac7530d7a39
1a75e5a0db428b09f27ad4d975b4ad1e7f85800e03ffeea35e8cf3fe67b18d4a1345
key:
f764a5a4b17e5d1ffba6e699d65560497ebaea6eb0b0d9010a6d979e298a39ff
base_nonce: 479afdf3546ddba3a9841f38
exporter_secret: 5c3d4b65a13570502b93095ef196c42c8211a4a188c4590d358
63665c705bb140ecba6ce9256be3fad35b4378d41643867454612adfd0542a684b61
799bf293f
```

#### C.6.2.1. Encryptions

sequence number: 0  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d30  
nonce: 479afdf3546ddba3a9841f38  
ct: de69e9d943a5d0b70be3359a19f317bd9aca4a2ebb4332a39bcd9c97d5fe62f3  
a77702f4822c3be531aa7843a1

sequence number: 1  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d31  
nonce: 479afdf3546ddba3a9841f39  
ct: 77a16162831f90de350fea9152cfc685ecfa10acb4f7994f41aed43fa5431f23  
82d078ec88baec53943984553e

sequence number: 2  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d32  
nonce: 479afdf3546ddba3a9841f3a  
ct: f1d48d09f126b9003b4c7d3fe6779c7c92173188a2bb7465ba43d899a6398a33  
3914d2bb19fd769d53f3ec7336

sequence number: 4  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d34  
nonce: 479afdf3546ddba3a9841f3c  
ct: 829b11c082b0178082cd595be6d73742a4721b9ac05f8d2ef8a7704a53022d82  
bd0d8571f578c5c13b99eccff8

sequence number: 255  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323535  
nonce: 479afdf3546ddba3a9841fc7  
ct: a3ee291e20f37021e82df14d41f3fbe98b27c43b318a36cacd8471a3b1051ab1  
2ee055b62ded95b72a63199a3f

sequence number: 256  
pt: 4265617574792069732074727574682c20747275746820626561757479  
aad: 436f756e742d323536  
nonce: 479afdf3546ddba3a9841e38  
ct: eecc2173celac14b27ee67041e90ed50b7809926e55861a579949c07f6d26137  
bf9cf0d097f60b5fd2fbf348ec

#### C.6.2.2. Exported Values

```
exporter_context:  
L: 32  
exported_value:  
62691f0f971e34de38370bff24deb5a7d40ab628093d304be60946afcdb3a936  
  
exporter_context: 00  
L: 32  
exported_value:  
76083c6dlb6809da088584674327b39488eaf665f0731151128452e04ce81bff  
  
exporter_context: 54657374436f6e74657874  
L: 32  
exported_value:  
0c7cfc0976e25ae7680cf909ae2de1859cd9b679610a14bec40d69b91785b2f6
```

C.7. DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, Export-Only AEAD

C.7.1. Base Setup Information

```
mode: 0
kem_id: 32
kdf_id: 1
aead_id: 65535
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
55bc245ee4efda25d38f2d54d5bb6665291b99f8108a8c4b686c2b14893ea5d9
pkEm:
e5e8f9bffff6c2f29791fc351d2c25ce1299aa5eaca78a757c0b4fb4bcd830918
skEm:
095182b502f1f91f63ba584c7c3ec473d617b8b4c2cec3fad5af7fa6748165ed
ikmR:
683ae0da1d22181e74ed2e503ebf82840deb1d5e872cade20f4b458d99783e31
pkRm:
194141ca6c3c3beeb4792cd97ba0ea1faff09d98435012345766ee33aae2d7664
skRm:
33d196c830a12f9ac65d6e565a590d80f04ee9b19c83c87f2c170d972a812848
enc:
e5e8f9bffff6c2f29791fc351d2c25ce1299aa5eaca78a757c0b4fb4bcd830918
shared_secret:
e81716ce8f73141d4f25ee9098efc968c91e5b8ce52ffff59d64039e82918b66
key_schedule_context: 009bd09219212a8cf27c6bb5d54998c5240793a70ca0a8
92234bd5e082bc619b6a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbf6dd
c9c64fee26bdd292
secret:
04d64e0620aa047e9ab833b0ebcd4ff026cefbe44338fd7d1a93548102ee01af
key:
base_nonce:
exporter_secret:
79dc8e0509cf4a3364ca027e5a0138235281611ca910e435e8ed58167c72f79b
```

#### C.7.1.1.1. Exported Values

```
exporter_context:
L: 32
exported_value:
7a36221bd56d50fb51ee65edfd98d06a23c4dc87085aa5866cb7087244bd2a36

exporter_context: 00
L: 32
exported_value:
d5535b87099c6c3ce80dc112a2671c6ec8e811a2f284f948cec6dd1708ee33f0

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
ffaabc85a776136ca0c378e5d084c9140ab552b78f039d2e8775f26efff4c70e
```

## C.7.2. PSK Setup Information

```
mode: 1
kem_id: 32
kdf_id: 1
aead_id: 65535
info: 4f6465206f6e2061204772656369616e2055726e
ikmE:
c51211a8799f6b8a0021fcba673d9c4067a98ebc6794232e5b06cb9febcbddf5
pkEm:
d3805a97cbcd5f08babd21221d3e6b362a700572d14f9bbeb94ec078d051ae3d
skEm:
1d72396121a6a826549776ef1a9d2f3a2907fc6a38902fa4e401afdb0392e627
ikmR:
5e0516b1b29c0e13386529da16525210c796f7d647c37eac118023a6aa9eb89a
pkRm:
d53af36ea5f58f8868bb4a1333ed4cc47e7a63b0040eb54c77b9c8ec456da824
skRm:
98f304d4ecb312689690b113973c61ffe0aa7c13f2fbe365e48f3ed09e5a6a0c
psk:
0247fd33b913760fa1fa51e1892d9f307fbe65eb171e8132c2af18555a738b82
psk_id: 456e6e796e20447572696e206172616e204d6f726961
enc:
d3805a97cbcd5f08babd21221d3e6b362a700572d14f9bbeb94ec078d051ae3d
shared_secret:
024573db58c887decba4c57b6ed39f2c9a09c85600a8a0ecb11cac24c6aaec195
key_schedule_context: 01446fb1fe2632a0a338f0a85ed1f3a0ac475bdea2cd72
f8c713b3a46ee737379a3f4c22aa6d9a0424c2b4292fdf43b8257df93c2f6adbf6dd
c9c64fee26bdd292
secret:
638b94532e0d0bf812cf294f36b97a5bdcb0299df36e22b7bb6858e3c113080b
key:
base_nonce:
exporter_secret:
04261818aeae99d6aba5101bd35ddf3271d909a756adcef0d41389d9ed9ab153
```

## C.7.2.1. Exported Values

```
exporter_context:
L: 32
exported_value:
be6c76955334376aa23e936be013ba8bbae90ae74ed995c1c6157e6f08dd5316

exporter_context: 00
L: 32
exported_value:
1721ed2aa852f84d44ad020c2e2be4e2e6375098bf48775a533505fd56a3f416

exporter_context: 54657374436f6e74657874
L: 32
exported_value:
7c9d79876a288507b81a5a52365a7d39cc0fa3f07e34172984f96fec07c44cba
```

#### Authors' Addresses

Richard L. Barnes  
Cisco  
Email: rlb@ipv.sx

Karthik Bhargavan  
Inria  
Email: karthikeyan.bhargavan@inria.fr

Benjamin Lipp  
Inria  
Email: ietf@benjaminlipp.de

Christopher A. Wood  
Email: caw@heapingbits.net