NIST Special Publication 800
NIST SP 800-227 ipd

Recommendations for Key-Encapsulation Mechanisms

Initial Public Draft

Gorjan Alagic
Elaine Barker
Lily Chen
Dustin Moody
Angela Robinson
Hamilton Silberg
Noah Waller

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-227.ipd

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Special Publication 800
NIST SP 800-227 ipd

Recommendations for Key-Encapsulation Mechanisms

Initial Public Draft

Gorjan Alagic
Elaine Barker
Lily Chen
Dustin Moody
Angela Robinson
Hamilton Silberg
Noah Waller
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-227.ipd

January 2025

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Charles H. Romine, performing the non-exclusive functions and duties of the Under Secretary of Commerce for Standards and Technology and Director, National Institute of Standards and Technology

Certain commercial equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

NIST Technical Series Policies

Copyright, Use, and Licensing Statements

NIST Technical Series Publication Identifier Syntax

Publication History

Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added in the final publication.]

How to cite this NIST Technical Series Publication:

Alagic G, Barker EB, Chen L, Moody D, Robinson A, Silberg H, Waller N (2025) Recommendations for Key-Encapsulation Mechanisms. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-227 ipd. https://doi.org/10.6028/NIST.SP.800-227.ipd

Author ORCID iDs

Gorjan Alagic: 0000-0002-0107-6037

Elaine Barker: 0000-0003-0454-0461

Lily Chen: 0000-0003-2726-4279

Dustin Moody: 0000-0002-4868-6684

Angela Robinson: 0000-0002-1209-0379

Hamilton Silberg: 0009-0004-4178-8954

Noah Waller: 0000-0002-6979-9725

Public Comment Period
January 7, 2025 - March 7, 2025

Submit Comments
sp800-227-comments@nist.gov

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/227/ipd, including related content, potential updates, and document history.

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

A key-encapsulation mechanism (KEM) is a set of algorithms that can be used by two parties under certain conditions to securely establish a shared secret key over a public channel. A shared secret key that is established using a KEM can then be used with symmetric-key cryptographic algorithms to perform essential tasks in secure communications, such as encryption and authentication. This document describes the basic definitions, properties, and applications of KEMs. It also provides recommendations for implementing and using KEMs in a secure manner.

Keywords

cryptography; encryption; key-encapsulation mechanism; key establishment; public-key cryptography.

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

1. assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or
2. assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:
   (a) under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
   (b) without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: sp800-227-comments@nist.gov

Table of Contents

List of Tables

List of Figures

1. Introduction

1.1. Background

A key-establishment scheme is a set of algorithms that can be used to securely establish a shared secret key between two or more parties. Such a shared secret key can then be used to perform tasks that are suitable for symmetric-key cryptography, such as efficient confidential communication.

Many widely-deployed key-establishment schemes — including those specified in NIST Special Publication (SP) 800-56Ar3 [1] and SP 800-56Br2 [2] — are vulnerable to cryptographic attacks that make use of a large-scale, cryptanalytically-relevant quantum computer. In 2016, NIST initiated a process to select and standardize post-quantum key-establishment schemes (i.e., key-establishment schemes that would not be vulnerable to attacks even by cryptanalytically-relevant quantum computers). In response, NIST received feedback from the cryptographic community that the post-quantum key-establishment schemes best suited for standardization and widespread deployment are key-encapsulation mechanisms (KEMs). The first KEM standard that resulted from this NIST post-quantum cryptography (PQC) standardization process was ML-KEM, which is specified in Federal Information Procession Standards (FIPS) 203 [3].

At the time of standardization of ML-KEM, NIST had not provided extensive guidance on the basic definitions, properties, and applications of KEMs. This recommendation is meant to provide this guidance, supplement the current and future standardization of KEMs, and provide recommendations for implementing and using KEMs in a secure manner.

1.2. Scope and Purpose

In combination with the appropriate FIPS or SPs that specify a particular KEM, this recommendation is intended to provide the necessary information for implementing that KEM in FIPS 140-validated modules. This recommendation also provides guidance for vendors who wish to securely combine keying material produced via quantum-vulnerable methods with keying material produced via post-quantum methods.

This recommendation does not discuss how or when to migrate from quantum-vulnerable key-establishment procedures to post-quantum KEMs (see [4]). This recomendation does not provide a specification for any particular KEM; such specifications will be provided in other FIPS and/or SPs, such as the specification of ML-KEM in FIPS 203 [3].

This recommendation includes purely explanatory and educational material to aid in the general understanding of KEMs. While NIST SPs typically only include material that pertains to what is **approved**, this SP describes KEMs both generally and with respect to what is **approved**. Specific requirements will be clearly noted with "**shall**" and "**must**" statements.

2. Definitions and Requirements

2.1. Definitions

approved FIPS-approved and/or NIST-recommended. An algorithm or technique that is either 1) specified in a FIPS or NIST recommendation, 2) adopted in a FIPS or NIST recommendation, or 3) specified in a list of NIST-**approved** security functions.

(KEM) ciphertext A bit string that is produced by the encapsulation algorithm and used as an input to the decapsulation algorithm.

computationally-bounded For a bit security strength λ, an adversarial algorithm is computationally-bounded if it is allowed at most 2^{λ} basic operations.

cryptanalytically-relevant quantum computer A device capable of using quantum algorithms to break a cryptosystem that is secure against classical (i.e., non-quantum) computers.

decapsulation The process of applying the Decaps algorithm of a KEM. This algorithm accepts a KEM ciphertext and the decapsulation key as input and produces a shared secret key as output.

decapsulation key A cryptographic key produced by a KEM during key generation and used during decapsulation.

efficient (cryptographic) algorithm An algorithm whose running time is practical for the relevant security strength. At a minimum, such an algorithm runs in time polynomial in the bit security strength λ.

encapsulation The process of applying the Encaps algorithm of a KEM. This algorithm accepts the encapsulation key as input, requires private randomness, and produces a shared secret key and an associated ciphertext as output.

encapsulation key A cryptographic key produced by a KEM during key generation and used by the encapsulation algorithm.

hash function A function on arbitrarily-long bit strings in which the length of the output is fixed.

identifier A bit string that is associated with a person, device, or organization. It may be an identifying name or something more abstract (e.g., a string consisting of an IP address).

key agreement A (pair-wise) key-establishment procedure in which the resultant secret keying material is a function of information contributed by both participants so that neither party can predetermine the value of the secret keying material independent of the contributions of the other party. Contrast with key transport.

key confirmation A procedure that provides assurance to one party (the key-confirmation recipient) that another party (the key-confirmation provider) possesses the correct secret keying material and/or shared secret from which that secret keying material is derived.

key-confirmation provider The party that provides assurance to the other party (the recipient) that the two parties have indeed established a shared secret key or shared keying material.

key-confirmation recipient The party that receives assurance from the other party (the provider) that the two parties have indeed established a shared secret key or shared keying material.

key-derivation method A method used to derive keying material from an initial shared secret(s) and possibly other information.

key-derivation key A key used as an input to a key-derivation function to derive additional keying material.

key-encapsulation mechanism (KEM) A set of three cryptographic algorithms: KeyGen (key generation), Encaps (encapsulation), and Decaps (decapsulation). These algorithms can be used by two parties to securely establish a shared secret key over a public channel.

key establishment A procedure that results in secret keying material that is shared among different parties. Key agreement, KEM, and key transport are all types of key establishment.

keying material A bit string such that any non-overlapping, contiguous segments of the string with required lengths can be used as secret keys, secret initialization vectors, and other secret parameters.

key pair A public key and its corresponding private key.

key transport A (pair-wise) key-establishment procedure whereby one party (the sender) selects a value for the secret keying material and then securely distributes the value to another party (the receiver). Contrast with key agreement.

message authentication code (MAC) A family of symmetric-key cryptographic algorithms acting on input data of arbitrary length to produce an output value of a specified length (called the MAC of the input data). The MAC can be employed to provide authentication of the origin of the input data and/or data integrity protection.

message authentication code (MAC) tag Data obtained from the output of a MAC algorithm (possibly by truncation) that can be used by an entity to securely verify the integrity and origination of the information used as input to the MAC algorithm.

must Indicates a requirement of this SP that might not be testable by a CMVP testing lab.

negligible A quantity is negligible for bit security strength λ if it is smaller than $2^{-\lambda}$.

party An individual (person), organization, device, or process. In this recommendation, there are typically two parties (e.g., Party A and Party B or Alice and Bob) that jointly perform the key-establishment process using a KEM.

pseudorandom A process (or data produced by a process) is said to be pseudorandom when the outcome is deterministic yet also appears random to computationally-bounded adversaries as long as the internal action of the process is hidden from observation. For cryptographic purposes, “effectively random” means “computationally indistinguishable from random within the limits of the intended security strength.”

public channel A communication channel between two honest parties that can be observed and compromised by third parties.

post-quantum algorithm A cryptographic algorithm that is believed to be secure even against adversaries who possess a cryptanalytically-relevant quantum computer.

quantum-vulnerable algorithm A cryptographic algorithm that is believed to be secure against adversaries who possess only a classical computer but is known to be insecure against adversaries who possess a cryptanalytically-relevant quantum computer.

shared secret A secret value that has been computed during a key-establishment scheme, is known by all participating parties, and is used as input to a key-derivation method to produce keying material.

shared secret key A shared secret that can be used directly as keying material, or as a symmetric key.

security strength A number associated with the amount of work that is required to break a cryptographic algorithm or system.

shall Used to indicate a requirement of this SP that will be tested by a CMVP testing lab.

should Used to indicate a strong recommendation but not a requirement of this SP. Ignoring the recommendation could lead to undesirable results.

side-channel attack An attack enabled by the leakage of information from a deployed cryptosystem. Characteristics that could be exploited in a side-channel attack include timing, power consumption, and electromagnetic and acoustic emissions.

symmetric-key algorithm A cryptographic algorithm that uses the same secret key for an operation and its complement (e.g., encryption and decryption). Also called a secret-key algorithm.

2.2. Requirements

Conforming implementations of **approved** KEMs are required to satisfy all of the below.

Requirements that are testable by a CMVP validation lab (i.e., **shall** statements):

RS1 (Section 4.1) KEM implementations **shall** comply with the specific NIST FIPS or SP that concretely specifies the algorithms of the relevant KEM. For example, implementations of ML-KEM **shall** comply with FIPS 203 [3]. *(Note: the CMVP will perform random input-output tests in an attempt to ascertain whether this requirement is satisfied. Ensuring full functional equivalence to the specification via testing is not possible; see also the "**must**" requirement **RM1** below.)*

RS2 (Section 4.1) KEM implementations **shall** comply with the guidance given in FIPS 140-3 [5] and associated implementation guidance.

RS3 (Section 4.1) KEM implementations **shall** use **approved** components with security strengths that are chosen appropriately for each KEM parameter set.

RS4 (Section 4.1) Random bits **shall** be generated using **approved** techniques, as described in the latest revisions of SP 800-90A, SP 800-90B, and SP 800-90C [6–8].

RS5 (Section 4.2) Except for random seeds and data that can be easily computed from public information, all intermediate values used in any given KEM algorithm (i.e., KeyGen, Encaps, and Decaps) **shall** be destroyed before the algorithm terminates.

RS6 (Section 5.4.1) When a nonce is used by the decapsulator during key confirmation (as specified herein), a nonce with a bit length (at least) equal to the targeted security strength of the KEM key-establishment process **shall** be used (see Appendix A.3).

RS7 (Section 5.4.1) For key confirmation, the MAC algorithm and KC_Key used **shall** have security strengths equal to or greater than the security strength of the KEM and parameter set used.

RS8 (Section 5.4.2) The KC_Key **shall** only be used for key confirmation and destroyed after use.

RS9 (Section 5.5.1) In multi-algorithm key-establishment schemes, shared secrets **shall** be combined via an approved key-combiner, as described in Section 5.5.2.

RS10 (Appendix A.1) When key confirmation requires the use of a MAC, it **shall** be an approved MAC algorithm (i.e., HMAC, AES-CMAC, or KMAC).

RS11 (Appendix A.1) When a MAC tag is used for key confirmation, an entity **shall** compute the MAC tag on received or derived data using a MAC algorithm with a $MacKey$ that is determined from a shared secret key.

Requirements that are not testable by a validation lab (i.e., **must** statements):

RM1 (Section 4.1). Implementations **must** correctly implement the mathematical functionality of the target KEM. *(Note: the CMVP will perform random input-output tests in an attempt to ascertain whether this requirement is satisfied. Ensuring full functional equivalence to the specification is not possible.)*

RM2 (Section 5.2) In applications of KEMs, a parameter set with application-appropriate security strength **must** be selected (see [9, Section 2.2]).

3. Overview of Key-Encapsulation Mechanisms

This section gives a high-level overview of key-encapsulation mechanisms (KEMs). It considers a KEM to be a collection of mathematical functions, together with data that specify parameters. Section 4 describes how to implement a KEM as a collection of computer programs. Section 5 describes how to deploy KEMs in applications.

3.1. Introduction

Modern symmetric-key cryptography provides a wide range of useful functionalities, including secure and highly efficient computation and communication. Before symmetric-key cryptography can be used, the participating parties need to establish a shared (i.e., symmetric) secret key. One approach to establishing such a key is over a public communication channel. Any algorithmic method that establishes a shared secret key over a public channel is called a *key-establishment scheme*. A general key-establishment scheme can require multiple rounds of communication and involve any number of parties.

A KEM is a specific type of key-establishment scheme. Typical key establishment via a KEM involves two parties (here referred to as Alice and Bob) and consists of the following three stages (see Figure 1):

1. (Key Generation) Alice generates a (private) decapsulation key and a (public) encapsulation key.
2. (Encapsulation) Bob uses Alice's encapsulation key to generate a shared secret key and an associated ciphertext. The ciphertext is sent to Alice.
3. (Decapsulation) Alice uses the ciphertext and her decapsulation key to compute another copy of the shared secret key.

Security of KEMs. When a KEM is used as in Figure 1, the result should be a shared secret key that is random, unknown to adversaries, and identical for Alice and Bob. Ensuring that security holds in practice is a complex task that relies on three conditions:

1. *Theoretical security*: Selecting a KEM that (as a collection of mathematical functions) is well-defined, correct, and satisfies an application-appropriate mathematical notion of security (see Sections 3.2 and 3.3)
2. *Implementation security*: Implementing the selected KEM in a real-world algorithm (e.g., a collection of routines) in a secure manner (see Section 4)
3. *Deployment security*: Deploying the implemented KEM in a manner that is secure for the relevant application and using the shared secret key in a secure manner (see Section 5.2)

Each of these three conditions are essential for security. For example, a KEM that is theoretically secure (i.e., it satisfies condition 1) but is implemented without side-channel

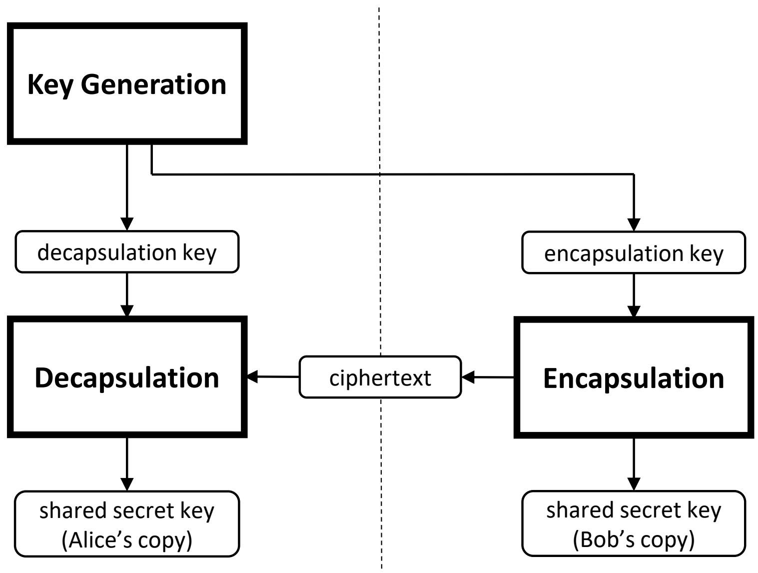

Fig. 1. Outline of key establishment using a KEM

countermeasures (so that it does not satisfy condition 2) or is deployed on a device with physical vulnerabilities (so that it does not satisfy condition 3) is likely to be insecure in practice.

History and development. KEMs were first introduced by Cramer and Shoup [10, 11] as a building block for constructing highly efficient public-key encryption (PKE) schemes. Their approach combines a Key Encapsulation Mechanism with a Data Encryption Mechanism (DEM); a DEM is simply a symmetric-key encryption scheme. The KEM is used to generate a shared secret key, while the DEM is used to encrypt an arbitrarily long stream of messages under that key. This is commonly referred to as the KEM/DEM paradigm (see the HPKE example in Section 6.2.1). This approach to constructing highly efficient public-key encryption has been the subject of several standards [1, 2, 10, 12–15]. Most recently, KEMs have attracted significant attention due to all of the post-quantum key-establishment candidates in the NIST PQC standardization process being KEMs. This ongoing process has produced one new KEM standard — ML-KEM in FIPS 203 [3] — with more KEM standards likely to follow.

3.2. Basic Definitions and Examples

This section establishes the basic definitions and properties of KEMs. Note that probabilistic algorithms require randomness, while deterministic algorithms do not.

Definition 1. *A **KEM** denoted by* Π *consists of the following four components:*

1. $\Pi.\mathsf{ParamSets}$ *(parameters): A collection of parameter sets*

2. Π.KeyGen *(key-generation algorithm): An efficient probabilistic algorithm that accepts a parameter set* $p \in \Pi$.ParamSets *as input and produces an encapsulation key* ek *and a decapsulation key* dk *as output*

3. Π.Encaps *(encapsulation algorithm): An efficient probabilistic algorithm that accepts a parameter set* $p \in \Pi$.ParamSets *and an encapsulation key* ek *as input and produces a shared secret key* K *and a ciphertext* c *as output*

4. Π.Decaps *(decapsulation algorithm): An efficient deterministic algorithm that accepts a parameter set* $p \in \Pi$.ParamSets*, a decapsulation key* dk*, and a ciphertext* c *as input and produces a shared secret key* K' *as output*

As this section views KEMs purely as mathematical objects, the labels p, ek, dk, c, K, and K' in Definition 1 are viewed as abstract variables that represent, for example, numbers or bit strings. In implementations, these variables will be represented with concrete data types (see Section 4).

In general, Definition 1 only requires some very basic properties from the four components that make up a KEM (see Example 1 below). In order to be useful and secure, a KEM should fulfill a number of additional properties. The first such property is *correctness* of the KEM algorithm. Correctness ensures that, in an ideal setting, the process in Figure 1 almost always produces the same shared secret key value for both parties.

Definition 2. *The key-encapsulation correctness experiment for a KEM* Π *and parameter set* $p \in \Pi$.ParamSets *consists of the following three steps:*

$$\text{1. } (\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p) \quad \textit{(perform key generation)} \tag{1}$$
$$\text{2. } (K, c) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}) \quad \textit{(perform encapsulation)} \tag{2}$$
$$\text{3. } K' \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c) \quad \textit{(perform decapsulation)} \tag{3}$$

The KEM Π *is* ***correct*** *if, for all* $p \in \Pi$.ParamSets*, the correctness experiment for* p *results in* $K = K'$ *with all but negligible probability.*

When Π.KeyGen and Π.Encaps are invoked in the correctness experiment, it is implied that their randomness is generated internally and uniformly at random. If one wishes to explicitly refer to the randomness used by these algorithms, then the following expressions can be used:

$$\text{Key generation (using randomness } r\text{):} \quad (\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p; r) \tag{4}$$
$$\text{Encapsulation (using randomness } s\text{):} \quad (K, c) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}; s) \tag{5}$$

These expressions can, for example, refer to the process of re-expanding a key pair $(\mathsf{ek}, \mathsf{dk})$ by running KeyGen using a stored seed r.

The following two simple but instructive examples show abstract KEMs that satisfy Definition 1 and Definition 2.

Example 1: Simple but insecure. As the following example shows, a correct and efficient KEM can still be completely insecure. Define a KEM DoNotUse as follows:

- DoNotUse.ParamSets: Contains a single, empty parameter set
- DoNotUse.KeyGen: On randomness r, outputs dk $:= r$ and ek $:= r$
- DoNotUse.Encaps: On input ek and randomness s, outputs $K := s$ and $c := s$
- DoNotUse.Decaps: On input dk and c, outputs $K' := c$

While DoNotUse is obviously a correct KEM since K' always equals K, it is also completely insecure since the shared secret key K is transmitted in plaintext. This shows that a KEM needs to satisfy additional properties in order to be secure (see Section 3.3).

Example 2: key transport using PKE. The following is a simple construction of a KEM from any public-key encryption scheme. A public-key encryption scheme PKE consists of a collection PKE.ParamSets of parameter sets and three algorithms: key generation PKE.KeyGen (that accepts a parameter set), encryption PKE.Encrypt (that accepts a parameter set, an encryption key, and a plaintext), and decryption PKE.Decrypt (that accepts a parameter set, a decryption key, and a ciphertext). One can construct a KEM KEMfromPKE from the public-key encryption scheme PKE as follows:

- KEMfromPKE.ParamSets = PKE.ParamSets
- KEMfromPKE.KeyGen = PKE.KeyGen
- KEMfromPKE.Encaps: On input p, ek and randomness s, output key $K := s$ and ciphertext $c \leftarrow$ PKE.Encrypt(p, ek, s).
- KEMfromPKE.Decaps: On input p, dk and c, output key $K' :=$ PKE.Decrypt(p, dk, c).

The efficiency, correctness, and security properties of KEMfromPKE depend on the respective properties of PKE.

Approved examples. Section 6.1 briefly discusses three additional examples of KEMs, each of which is an **approved** algorithm.

1. In Section 6.1.1, ECDH-KEM is a KEM based on ECDH key exchange.
2. In Section 6.1.2, RSASVE-KEM is RSA key transport.
3. In Section 6.1.3, ML-KEM is a lattice-based post-quantum KEM.

ECDH-KEM and RSASVE-KEM are based on NIST-standardized key-establishment schemes that can easily be viewed as KEMs. ML-KEM is the first key-establishment scheme to be standardized by NIST directly as a KEM.

A remark on key transport and key agreement. There are various ways to categorize two-party key-establishment schemes. One particular categorization distinguishes between *key agreement* and *key transport*. In key agreement (e.g., a Diffie-Hellman key exchange), both parties contribute information that influences the final shared secret key. In key transport (e.g., RSA-OAEP [2]), one party selects the key and then transmits it (in some form) to the other party.

Depending on the internal structure of the encapsulation function, a KEM could be viewed as either a key-agreement scheme or a key-transport scheme. For example, the shared secret key in ML-KEM [16] is a function of both the randomness provided by Bob and the (randomly generated) encapsulation key of Alice. Therefore, ML-KEM could be viewed as a key agreement scheme. However, as the example KEMfromPKE shows, the encapsulation operation in a KEM might simply consist of Bob generating the shared secret key and then encrypting it; this is precisely key transport. If an application requires a particular type of key establishment (either key agreement or key transport), this can be achieved using any KEM by taking appropriate additional steps using standard symmetric-key cryptography techniques.

3.3. Theoretical Security of KEMs

This section discusses the theoretical security of KEMs. Section 4 discusses KEM implementation security, and Section 5.2 discusses the secure deployment of KEMs.

Semantic security. Informally speaking, a secure key-establishment procedure produces a shared secret key K that is uniformly random and unknown to adversaries. This property should hold despite the fact that adversaries can freely observe the messages transmitted by Alice and Bob. In the case of KEMs, the encapsulation key ek and ciphertext c should reveal no information about the underlying shared secret key K or the decapsulation key dk. Moreover, even adversaries who somehow learn some partial information (e.g., if the first half of K is accidentally leaked) should not be able to combine that information with ek and c to learn more (e.g., the last bit of K). This informal notion of security can be rigorously formalized, and the resulting definition is called *semantic security* [17].

Passive adversaries and IND-CPA. The formal definition of semantic security for KEMs is somewhat complex and unwieldy. Thankfully, it has an equivalent definition that is simple to describe and easy to work with. It is defined in terms of an imaginary "ciphertext indistinguishability" experiment (see Figure 2). In this experiment, an adversary is given an encapsulation key ek, a ciphertext c, and either the true shared secret key underlying c or a freshly generated random string. The adversary's goal is to distinguish these two scenarios, and they are free to use ek to generate their own encapsulations to help them in this task. This experiment is called "indistinguishable ciphertexts under chosen plaintext attack" (IND-CPA).

Challenger:		**Adversary:**
$(\mathsf{ek},\mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p)$		
$(K_0, c) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek})$		
$K_1 \leftarrow \{0,1\}^{\lvert K_0 \rvert}$		
$b \leftarrow \{0,1\}$		
	$\xrightarrow{\mathsf{ek},\ c,\ K_b}$	
	$\xleftarrow{\ b'\ }$	
output **WIN** iff $b = b'$.		

Fig. 2. The IND-CPA security experiment for a KEM Π

Definition 3 (IND-CPA, informal)**.** *A KEM* Π *has indistinguishable ciphertexts (or is IND-CPA) if, for every computationally-bounded adversary* $\mathcal{A}$*, the probability that* $\mathcal{A}$ *wins the experiment* IND-CPA$[\Pi]$ *is negligibly close to* $1/2$*.*

In the IND-CPA experiment, the adversary is free to study the encapsulation key ek and the ciphertext c in order to identify whether K_b is the true key. However, the adversary is not capable of actively interfering with the challenger's use of the decapsulation key. As a result, IND-CPA only captures security against *passive* adversaries (i.e., eavesdroppers).

Challenger:		**Adversary:**
$(\mathsf{ek},\mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p)$		
$(K_0, c) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek})$		
$K_1 \leftarrow \{0,1\}^{\lvert K_0 \rvert}$		
$b \leftarrow \{0,1\}$		
	$\xrightarrow{\mathsf{ek},\ c,\ K_b}$	
		$\leftrightarrows \boxed{\Pi.\mathsf{Decaps}(\mathsf{dk}, \star)}$
	$\xleftarrow{\ b'\ }$	
output **WIN** iff $b = b'$.		

Fig. 3. The IND-CCA security experiment for a KEM Π

Active adversaries and IND-CCA. Real-world experience indicates that adversaries can sometimes actively interfere with key-establishment processes and use this ability to uncover the shared secret key. For example, an active adversary may be able to convince an honest user to decapsulate some ciphertexts of the adversary's choosing. In such a scenario, it is natural to ask whether *other* ciphertexts are still protected. In this setting, IND-CPA security is insufficient. Instead, one must consider security against so-called chosen-ciphertext attacks (CCA).

The IND-CCA$[\Pi]$ experiment for a KEM Π is described in Figure 3. It is similar to the IND-CPA experiment, except that the adversary is now also granted "black-box oracle access" to the decapsulation function $c \mapsto \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c)$. This means that the adversary is allowed to submit ciphertexts c^* that they generate and get the response $K^* \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c^*)$. The only restriction is that they cannot submit the actual ciphertext c produced by the challenger since that would make the game trivial to win for any KEM.

Definition 4 (IND-CCA, informal)**.** *A KEM Π is IND-CCA if, for every efficient adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the experiment* IND-CCA$[\Pi]$ *is negligibly close to $1/2$.*

Note that ML-KEM, the first post-quantum KEM standardized by NIST, is believed to satisfy IND-CCA security [3].

4. Requirements for Secure KEM Implementations

As discussed in Section 3.1, a KEM (as a mathematical object) should satisfy both correctness (Definition 2) and an appropriate notion of security (Definition 3 or Definition 4). In order for such a KEM to be used in real-world applications, it needs to be implemented in actual code as part of a cryptographic module. The quality of the resulting implementation has a dramatic impact on usability and security in real-world applications.

The following subsections detail some requirements for cryptographic modules that implement a KEM. While adherence to these requirements is required for conforming implementations of **approved** KEMs, it does not guarantee that a given implementation will be secure.

For a discussion of requirements for applications that make use of a KEM cryptographic module, see Section 5.2.

4.1. Compliance to NIST Standards and Validation

Conforming implementations of **approved** KEMs are required to comply with the requirements outlined in this section, as well as all other applicable NIST standards. In addition, such implementations are required to use only **approved** cryptographic elements, and to pass FIPS-140 validation.

Implementing according to NIST standards. Implementations **shall** comply with a specific NIST FIPS or SP that concretely specifies the algorithms of the relevant KEM. For example, a conforming implementation of ML-KEM **shall** comply with FIPS 203 [3]. Each FIPS or SP that specifies a KEM will have special requirements for the particular scheme in question. These requirements will include specifications for all algorithms and parameter sets of the relevant KEM. In particular, concrete data types will be specified for the parameter sets, keys, ciphertexts, and shared secret keys (recalling Definition 1) of the relevant KEM.

The requirements in any FIPS or SP that standardizes a particular KEM are in addition to the general requirements described in this section. Any implementations **shall** follow the guidance given in FIPS 140-3 [5] and associated implementation guidance.

Approved cryptographic elements. KEMs commonly make use of other cryptographic elements (see Appendix A), such as random bit generators (RBGs) and hash functions. KEM implementations **shall** use **approved** cryptographic elements with security strengths that are appropriately chosen for each KEM parameter set. In particular, random bits **shall** be generated using **approved** techniques, as described in the latest revisions of SP 800-90A, SP 800-90B, and SP 800-90C [6–8].

Testing and validation. Mistakes in implementations can easily lead to security vulnerabilities or a loss of usability. Therefore, it is crucial that implementations are validated for conformance to the appropriate cryptographic specifications and FIPS 140 by the Cryptographic Algorithm Validation Program (CAVP) and Cryptographic Module Validation Program (CMVP).

It is important to note that validation testing typically only tests that a given implementation correctly computes the desired output for a small number of (often randomly sampled) inputs. This means that validation testing does not guarantee correct functioning on all inputs—in fact, this is often impossible to ensure. Nonetheless, implementations **must** correctly implement the mathematical functionality of the target KEM.

As validation only tests input-output behavior, implementations need not follow the exact step-by-step algorithmic specifications in the NIST standard specifying the relevant KEM. Any implementation that produces the correct output for every input will pass validation.

Requiring equivalence only at the level of input-output functionality (e.g., rather than in terms of step-by-step behavior) is desirable, as different implementations can then be optimized for different goals. For example, some implementations will focus on maximizing efficiency, while other implementations will employ numerous side-channel and leakage protection techniques.

4.2. Managing Cryptographic Data

KEM implementations need to manage all cryptographic data appropriately. This applies to data used during the execution of the three KEM algorithms as well as data-at-rest. As a cryptographic module has no control over data that exists outside the module (e.g., while in transit from one module to another), such data is not discussed here. However, a cryptographic module can exert control over what data it outputs to the outside world (e.g., by ensuring correct implementations of all functions, as discussed above). It can also exert control over what data it accepts from the outside world (e.g., by performing appropriate input-checking and importing, as discussed below).

In general, data needs to be destroyed as soon as it is no longer needed. Some examples include destroying intermediate computation values at the end of an algorithm, destroying randomness generated by RBGs after encapsulation, and destroying keys after all relevant communication sessions are completed.

Input checking. The correct and secure operation of cryptographic operations depends crucially on the validity of the provided inputs. Even relatively benign faults, such as an input that is too long or too short, can have serious security consequences. KEM implementations need to perform input checking in an appropriate manner for all KEM algorithms (i.e., KeyGen, Encaps, and Decaps). The exact form of the required input checking is described in the FIPS or SP that specifies the relevant KEM.

Sometimes, an input will not need to be checked. Instead, the implementer can acquire assurance that the input was validly generated or has already been checked, as in the following cases:

1. If the cryptographic module generated an input internally using an algorithm that ensures validity and stored that input in a manner that prevents modification, then the module is not required to check that input. For example, if the module generated a decapsulation key dk via KeyGen and then stored dk in a manner that prevents modification, then the module can later invoke Decaps directly on dk without performing any input checking.

2. If the cryptographic module checks an input once and stores that input in a manner that prevents modification, then the module is not required to check that input again. For example, if the module performed input-checking on a given encapsulation key ek and stored it in a manner that prevents modification, then the module may invoke Encaps directly on ek (even repeatedly) without performing any further input checking.

3. If the cryptographic module imports the relevant input from a trusted third party (TTP) and the TTP can provide assurance that the input does not need input-checking, then the module is not required to check the input.

Intermediate values. All intermediate values used in any given KEM algorithm (i.e., KeyGen, Encaps, Decaps) **shall** be destroyed before the algorithm terminates. However, there are two exceptions to this rule:

1. A random seed used for key generation may be stored for the purpose of recomputing the same key pair at a later time.

2. Data that can be easily computed from public information (e.g., from the encapsulation key) may be stored to improve efficiency.

When values are stored under either of these exceptions, the storage needs to be performed according to the rules for data-at-rest.

The outputs of an algorithm are not considered to be intermediate values and will thus not be immediately destroyed in typical situations. The format in which outputs and inputs are stored depends on the implementation (see discussion of data formats below.)

Data at rest. A cryptographic module that implements a KEM needs to maintain certain data-at-rest. This can include both private data (e.g., seeds and decapsulation keys) and public data (e.g., encapsulation keys). In general, private data needs to be stored within the cryptographic module in a manner that is secure and protected against both leakage and unauthorized modification. Private data needs to be destroyed as soon as it is no longer needed. The import and export of private data (e.g., seeds, decapsulation keys, shared secret keys) need to be performed in a secure manner. In general, public data stored within the cryptographic module needs to be stored in a manner that is secure and protected against unauthorized modification [5, 18].

Data formats, import and export. FIPS validation tests input and output behavior of the relevant KEM algorithms using a specific data format. Typically, this format is byte arrays containing the relevant inputs and outputs as described in the FIPS or SP specifying the relevant KEM. This format is required for testing, but is not to be viewed as a requirement for internal storage, data import, or data export. A given cryptographic module may choose to store, import, or export data (whether sensitive or not) using other formats. The desired format can vary significantly depending on the application. For example, some applications might call for storing keys using only a short seed, while other applications might call for storing keys in an expanded format that allows for faster computations. In any case, storage, import, and export of sensitive data needs to be performed securely, regardless of the chosen data format.

4.3. Additional Requirements

The following are additional requirements for cryptographic modules implementing **approved** KEMs.

Failures and aborts. Each of the KEM algorithms (i.e., KeyGen, Encaps, Decaps) and any algorithms of their cryptographic elements (e.g., DRBGs or hash functions) can potentially fail or abort. This could be a result of normal KEM operations (e.g., decapsulating a ciphertext that was corrupted by the environment during transmission), a hardware or software failure (e.g., a failed DRBG execution due to a memory fault), or an adversarial attack. Implementers need to take precautions to ensure that the cryptographic module handles failures and aborts appropriately. In particular, leaking information about failures and aborts outside of the perimeter of the cryptographic module **should** be avoided.

Side-channel protection. Cryptographic modules for KEMs **should** be designed with appropriate countermeasures against side-channel attacks. This includes protecting against timing attacks with constant-time implementations and protecting memory from leakage. Universal guidance is unlikely to be helpful as exposure to side-channel attacks varies significantly with the desired application, and countermeasures are often costly.

5. Using KEMs Securely in Applications

This section describes how to deploy a KEM in real-world applications in a manner that is useful and secure, assuming that the KEM under discussion satisfies an appropriate notion of theoretical security (see Section 3.3) and has been securely implemented in a cryptographic module (see Section 4).

5.1. How to Establish a Key With a KEM

This section describes how a KEM can be used to establish a shared secret key between two parties. The description will go into greater detail than the brief outline of Section 3.1. However, since KEMs are highly flexible and can be used in a wide range of applications and contexts, no single description can account for all variations. Sections 6.2.1, 6.2.2 and 6.2.3 provide more detailed examples of special cases of key establishment using a KEM.

For simplicity of exposition, the two parties in the key establishment process will be referred to as Alice and Bob. It is assumed that Alice and Bob are communicating over a single bidirectional channel and will only use that channel to transmit data to each other.

The key establishment process using a KEM Π proceeds as follows:

1. **Preparation.** Before key establishment can begin, a parameter set $p \in \Pi.\mathsf{ParamSets}$ needs to be selected. Depending on the application, p may be selected by Alice, by Bob, or through an interactive negotiation between Alice and Bob. (In fact, the choice of the KEM Π itself could be made at this stage.)

2. **Key generation.** Alice begins by running the key generation algorithm in her cryptographic module:
$$(\mathsf{ek}_A, \mathsf{dk}_A) \leftarrow \Pi.\mathsf{KeyGen}(p)\,. \tag{6}$$
During the execution of KeyGen, Alice's module internally generates private randomness using an appropriate RBG. Alice then transmits ek_A to Bob and keeps dk_A private.

3. **Encapsulation.** Bob receives ek_A from Alice and uses it to execute the encapsulation algorithm in his cryptographic module:
$$(K_B, c_B) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}_A)\,. \tag{7}$$
During the execution of Encaps, Bob's module internally generates private randomness using an appropriate RBG. Bob then transmits c_B to Alice and keeps K_B private.

4. **Decapsulation.** Alice receives c_B from Bob and runs the decapsulation algorithm in her module using her decapsulation key and Bob's ciphertext:
$$K_A \leftarrow \Pi.\mathsf{Decaps}(\mathsf{dk}_A, c_B)\,. \tag{8}$$
Alice keeps K_A private.

5. **Using the shared secret key.** If the appropriate conditions are satisfied (see Section 5.2), then K_A will equal K_B and can be used by Alice and Bob for any symmetric-key cryptographic protocol. A typical choice is to use $K_A = K_B$ as the key for an authenticated encryption scheme (e.g., AES-GCM [19]), thereby establishing a communication channel between Alice and Bob that satisfies both confidentiality and integrity.

Figure 4 depicts the high-level stages of this process.

Alice:		**Bob:**
1.	$\xleftrightarrow{\Pi, p}$	
2. $(\mathsf{ek}_A, \mathsf{dk}_A) \leftarrow \Pi.\mathsf{KeyGen}(p)$		
	$\xrightarrow{\mathsf{ek}_A}$	
3.		$(K_B, c_B) \leftarrow \Pi.\mathsf{Encaps}(\mathsf{ek}_A)$
	$\xleftarrow{c_B}$	
4. $K_A \leftarrow \Pi.\mathsf{Decaps}(\mathsf{dk}_A, c_B)$		
5. output: K_A		output: K_B

Fig. 4. Simple key establishment using a KEM

Additional considerations. The steps 1-5 in the key establishment process above might need to be modified depending on the security and functionality needs of the application. Some common modifications are as follows.

Static versus ephemeral. Consider an application in which Alice independently decides on a parameter set, performs key generation, and publishes the resulting encapsulation key ek_A. Alice might then accept many connections from multiple parties over a long period of time, each initiated via ek_A. Each such connection would follow stages 3-5 described above. While the other party in each connection would always encapsulate with ek_A, each ciphertext is freshly generated and only applicable to the connection between Alice and that party. In this scenario, Alice's encapsulation key is said to be *static*.

In other applications, Alice might want to use a particular key pair to establish only a single connection (e.g., as part of a protocol that ensures forward secrecy). In that case, she will perform key generation, send her encapsulation key ek_A to a specific party (Bob), and discard ek_A once the connection with Bob is established. In this scenario, Alice's encapsulation key is said to be *ephemeral*. In some applications, Alice might decide to use ek_A for multiple connections but only for a brief period of time, which is typically still considered an ephemeral setting.

Authentication. In most applications, some form of authentication and cryptographic integrity checking is required (e.g., to prevent "machine-in-the-middle" attacks). Assuring this is highly application-dependent and typically requires additional cryptographic elements, such as digital signatures and certificates. Section 6.2.2 and Section 6.2.3 provide some illustrative examples.

Key confirmation and derivation. In some applications, Alice and Bob will use K_A and K_B directly as symmetric keys as soon as the decapsulation and encapsulation stages are successfully completed, respectively. If $K_A \neq K_B$, a failure in the desired symmetric-key functionality will likely follow. For other applications, Alice and Bob might need to first post-process K_A and K_B appropriately and then use the results of that post-processing step—if successful—as their symmetric keys. This post-processing might include key confirmation steps to confirm that $K_A = K_B$ and reject them otherwise (see Section 5.4). It might also include key derivation steps that securely produce multiple symmetric keys from the initial shared secret key (see Section 5.3). In some cases, key confirmation might also involve performing additional computations during the encapsulation and decapsulation stages to reduce the number of communication rounds.

5.2. Conditions for Using KEMs Securely

This section discusses general requirements for securely using **approved** KEMs in applications. As discussed in point 1 below, the first step involves selecting an **approved** KEM that has been implemented in a validated cryptographic module (see Section 4). Deploying such a cryptographic module in applications entails a number of additional requirements that are outlined below. Adherence to these requirements does not guarantee that the relevant KEM application will be secure.

The overall requirements fall into four general categories: KEM algorithm security, device security, channel security, and key usage security. Below, each category is briefly summarized in one prescriptive statement; a more detailed description of the requirements applicable to that category then follow.

1. **KEM algorithm security:** the selected KEM Π is **approved**, appropriate for the application, and implemented and deployed in a secure manner.

   Being an **approved** KEM, Π will satisfy correctness (Definition 2) and either IND-CPA or IND-CCA security (see Section 3.3). Whenever possible, IND-CCA-secure KEMs **should** be used. For some specific applications (e.g., ephemeral key establishment), IND-CPA security might be sufficient.

   Cryptographic module implementation. The implementations of Π used by Alice and Bob need to satisfy the requirements in Section 4. Whether a given implementation is sufficiently secure is an application-dependent question. For example, an implementation might be secure enough for use on a web server in a physically secure location but have insufficient side-channel protections for use on an embedded device.

Parameter set selection. A parameter set of Π with application-appropriate security strength **must** be selected (see [9, Section 2.2]).

KEM key management. If the application calls for an ephemeral-ephemeral key exchange, each key pair is only used for a brief period of time. In any case, all KEM keys and any seeds are destroyed as soon as they are no longer needed.

2. **Device security:** the devices used to execute KEM algorithms and store any intermediate data (e.g., decapsulation keys) are appropriately secured.

   Physical protection. The devices need to be appropriately protected against attacks (see [18, Section 5]). This includes protection against leakage, physical intrusion, remote access, and corruption.

   Secure storage. The device needs to provide appropriate secure storage for sensitive data (e.g., KEM keys, seeds, shared secret keys, and any derived keys) and destroy that data when required by the cryptographic module (See Section 4.2).

3. **Channel security:** the key-establishment process that takes place over the channel used by Alice and Bob needs to satisfy an application-appropriate notion of integrity.

   Pre-established versus simultaneous. Ensuring the integrity of the key-establishment process could be achieved by first ensuring the integrity of the channel and then performing key establishment. More commonly, integrity is assured simultaneously with key establishment by augmenting the key-establishment process with additional steps and checks.

   Unilateral versus bilateral. For some applications, only Alice is assured of Bob's identity and the integrity of Bob's messages. This is commonly called a unilaterally authenticated key exchange (see Section 6.2.3). In other applications, both Alice and Bob will require assurances of the other party's identity and the integrity of their messages. This is commonly called a bilaterally authenticated key exchange.

   Secure authentication algorithms. For all applications, the cryptographic algorithms (e.g., signatures, other KEMs) and other elements (e.g., certificates) required to establish channel integrity need to be selected and deployed securely.

4. **Key usage security:** the shared secret key produced by the KEM is used appropriately and securely.

   Key processing and management. Key confirmation and key derivation steps are performed appropriately, as required by the application (see Sections 5.4 and 5.3). Each shared secret key and any derived keys are destroyed as soon as they are no longer needed (see Section 4.2).

Secure symmetric-key algorithms. The KEM shared secret key and any derived keys should only be used with appropriately secure symmetric-key cryptographic algorithms. In particular, the security of the symmetric-key algorithms used is appropriate for the security provided by the KEM so that the combined algorithm (consisting of key establishment followed by symmetric cryptography operations) fulfills the desired security properties.

5.3. Key Derivation

Certain key-establishment schemes (e.g., Diffie-Hellman key exchange) can be viewed as first generating a shared secret, and then performing a key derivation step that transforms the shared secret into a shared secret key. KEMs, on the other hand, by definition output a key that is ready to use. As a result, key derivation is not required when using KEMs. Still, some applications using KEMs will require key derivation. This is the case, for example, when the application requires that the shared secret key K is expanded in order to create a collection of keys whose total length exceeds the length of K.

As specified in SP 800-108 [20], key derivation consists of applying a *key-derivation method* (KDM) to a *key-derivation key*. A KDM is an algorithm for transforming a given key-derivation key (along with possibly some other data) into keying material (e.g., a list of keys).

An example of a key-derivation method is:

1. Concatenate the key-derivation key K with optional data z.
2. Apply a key-derivation function KDF.

The final output of key derivation is then simply $\mathsf{KDF}(K||z)$.

In SP 800-56C [21], several key-derivation methods are defined for the setting in which the input to key derivation is a shared secret for one of the key-establishment schemes specified in [1, 2] (rather than a key-derivation key).

When key derivation for a KEM Π is needed, the shared secret key output by Π (i.e., as an output of Π.Encaps or Π.Decaps) may be used as a key-derivation key supplied to an **approved** key-derivation method specified in SP 800-108 [20], SP 800-56C [21], or SP 800-133 [22]. In the case where a KDM from SP 800-56C is used, the shared secret key of the KEM is used as an input to the KDM in place of the shared secret.

A simple example of key derivation is included in the example protocol in Section 6.2.3.

5.4. Key Confirmation

Key confirmation (KC) refers to the actions taken to provide assurance to one party (the key-confirmation recipient) that another party (the key-confirmation provider) possesses matching keying material. In the case of KEMs, this confirmation is done for keying material that was produced by encapsulation and/or decapsulation.

Key confirmation **should** be used during KEM usage, as it may enhance the security properties of the overall key-establishment process. Confirming successful establishment of the shared secret key can also address potential errors in transmission or decapsulation. While this section describes an explicit process, key confirmation can be accomplished in a variety of other ways. For example, successful use of the shared secret key for authenticated encryption can act as key confirmation.

Key confirmation is typically achieved by exchanging a value that can only be calculated correctly with very high probability if the key establishment was successful. Some common protocols perform key confirmation in a manner that is integrated into the steps of the protocol. For example, bilateral key confirmation is provided during a TLS handshake protocol by the generation and verification of a MAC over all previous messages in the handshake using a symmetric MAC key that was established during the handshake.

In some circumstances, it may be appropriate to perform key confirmation by including dedicated key-confirmation steps into a key-establishment scheme. An acceptable method for providing key confirmation during a key-establishment scheme is provided below. In this method, key confirmation is provided by the KC provider calculating a MAC tag and sending it to the KC recipient for confirmation of the provider's correct calculation of the shared secret key. Unilateral key confirmation is provided when only one of the parties serves as the key-confirmation provider. If mutual key confirmation is desired (i.e., bilateral key confirmation), then the parties swap roles for the second KC process, and the new provider (i.e., the previous recipient) sends a MAC value on a different data string (i.e., MAC_Data) to the new recipient (i.e., the previous provider).

If other methods are used, this recommendation makes no statement as to their adequacy.

Key-confirmation key. The key-confirmation steps specified in this recommendation can be incorporated into any scheme using a KEM to establish a shared secret key. To perform key confirmation, a dedicated KC key will be determined from the shared secret key produced by the KEM. The KC provider will then use the KC key with an approved MAC algorithm to create a MAC tag on certain data and provide the tag to the KC recipient. The KC recipient will then obtain the KC key from their copy of the shared secret key produced by the KEM and use it to verify the MAC tag.

5.4.1. Creating the MAC Data

During key confirmation, the KC provider creates a message with a $MacTag$ that is computed on MAC_Data that contains context-specific information. The MAC_Data is formatted as follows:

$$\mathsf{MAC\_Data} = \mathsf{KC\_Step\_Label} \,\|\, \mathsf{ID}_P \,\|\, \mathsf{ID}_R \,\|\, \mathsf{Eph}_P \,\|\, \mathsf{Eph}_R \,\|\, \mathsf{Extra}_P \,\|\, \mathsf{Extra}_R$$

- KC_Step_Label is a six-byte character string that indicates that the MAC_Data is used for key confirmation, whether the MAC_Data is used for the first or second key-confirmation message, and the party serving as the KC provider, either the encapsulator (E) or decapsulator (D). The four valid options are "KC_1_E", "KC_2_E", "KC_1_D", or "KC_2_D". As an example, "KC_1_D" indicates that the decapsulator (D) is the KC provider and sends the first KC message. "KC_2_E" could then be used by the encapsulator (E) to provide bilateral key confirmation.
- ID_P and ID_R are the identifiers used to label the KC provider and recipient, respectively.
- Eph_P and Eph_R are ephemeral data provided by the KC provider and recipient, respectively. The encapsulator's ephemeral data is the ciphertext. The decapsulator's ephemeral data is encapsulation key ek if ek is ephemeral; otherwise, the decapsulator's ephemeral data **shall** be a nonce with a bit length that is at least equal to the targeted security strength of the KEM key-establishment process (see Appendix A.3).

  When a nonce is used during key confirmation, it needs to be provided to the encapsulator before they can complete MAC_Data for $MacTag$ generation or verification.
- Extra_P and Extra_R are optional additional data provided by the KC provider and recipient, respectively. This could include additional identifiers, values computed during the key-establishment process, or any other information that the party wants to include. This information can be known ahead of time by both parties or transmitted during key confirmation.

The MAC algorithm and KC_Key used **shall** have security strengths that are equal to or greater than the security strength of the KEM and parameter set used. See Appendix A.1 for permitted MAC algorithms and further details.

5.4.2. Obtaining the Key-Confirmation Key

In order to create and validate the MAC tag for the created MAC_Data, the parties create a dedicated key-confirmation key, or KC_Key. This can be either a section of the KEM shared secret key or part of the derived keying material from the KEM shared secret key when using a derivation function (see Section 5.3). The KC_Key **shall** only be used for key confirmation and destroyed after use.

When a derivation function is used. After computing the plaintext shared secret value and applying the key-derivation method to obtain the derived keying material Derived_Keying_Material, the key-confirmation provider uses agreed-upon bit lengths to parse Derived_Keying_Material into two parts — the key-confirmation key (KC_Key) and the key(s) to subsequently protect data (Data_Key):

$$\text{Derived\_Keying\_Material} = \text{KC\_Key} \,||\, \text{Data\_Key}.$$

When a derivation function is NOT used. The key-confirmation provider parses the plaintext output of the encapsulation process into KC_Key and Data_Key:

$$\text{KEM\_plaintext\_output} = \text{KC\_Key} \,||\, \text{Data\_Key}.$$

5.4.3. Key-Confirmation Example

The key-confirmation process can be achieved in multiple ways. The provided example showcases unilateral key confirmation from the encapsulator to the decapsulator, which can be used for a client (i.e., Alice) requesting confirmation of successful key establishment from the server (i.e., Bob). Figure 5 shows this process.

Alice (Decapsulator, Client):		**Bob (Encapsulator, Server):**
1. $(\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p)$		
	$\xrightarrow{\mathsf{ek}, \mathsf{ID}_A, \mathsf{Extra}_A}$	
2.		$(c, K_{B0}) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek})$ $K_{Bkc} \| \| K_{B1} \leftarrow \mathsf{KDF}(K_{B0})$
3.		Construct MAC_Data $\mathsf{tag} \leftarrow \mathtt{MAC}(K_{Bkc}, \text{MAC\_Data})$
	$\xleftarrow{c, \mathsf{tag}, \mathsf{ID}_B, \mathsf{Extra}_B}$	
4. $K_{A0} \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c)$ $K_{Akc} \| \| K_{A1} \leftarrow \mathsf{KDF}(K_{A0})$		
5. Construct MAC_Data if $\mathtt{MAC}.\mathsf{Ver}(K_{Akc}, \text{MAC\_Data}, \mathsf{tag})$ rejects, **abort**.		
6. result: K_{A1}		result: K_{B1}

Fig. 5. Key-confirmation example with an ephemeral key pair

1. The decapsulating party (i.e., Alice) begins by generating a set of ephemeral keys $(\mathsf{ek}, \mathsf{dk})$ for KEM Π under the agreed parameter set p. Alice then sends ek, Alice's identifying string (ID_A), and any extra data Extra_A to include in the key confirmation to Bob.
2. The encapsulating party (i.e., Bob) performs encapsulation with the received ek to generate ciphertext c and initial key K_{B0}. Bob then derives two keys from K_{B0}: a key-confirmation key K_{Bkc} to perform key confirmation and additional key material K_{B1}.
3. Bob constructs MAC_Data using the following in order:

- The constant string ”KC_1_E”, which indicates that the encapsulator (i.e., Bob) is providing key confirmation and that this is the first KC message
- ID_B, which is Bob's identifier string
- ID_A, which is Alice's identifier string
- Ciphertext c, which is the KC provider's (Bob's) ephemeral value
- Encapsulation key ek, which is the KC recipient's (Alice's) ephemeral value
- Extra_B, which refers to any extra data that Bob (the KC provider) would like to include
- Extra_A, which refers to any extra data provided by Alice (the KC recipient)

Bob calculates the MAC tag tag using K_{Bkc} on $\mathsf{MAC\_Data}$ and sends the following to Alice: 1) ciphertext c, 2) the generated tag tag, 3) and any extra data (Extra_B) that Bob included in the $\mathsf{MAC\_Data}$.

4. Alice performs decapsulation on the received ciphertext c using the previously generated decapsulation key dk to calculate initial key K_{A0}. Alice then derives two keys from K_{A0} similarly to Bob (in step 2) with key-confirmation key K_{Akc} and other keying material K_{A1}.
5. Alice constructs $\mathsf{MAC\_Data}$ as Bob did in step 3 and verifies the received tag for the $\mathsf{MAC\_Data}$ using key K_{Akc}. Alice aborts if the tag is rejected or continues if it is verified.
6. Alice now has additional assurance that K_{A1} matches K_{B1}. Alice and Bob destroy the key-confirmation keys K_{Akc} and K_{Bkc} and can proceed to use K_{A1} and K_{B1} as planned.

5.5. Multi-algorithm KEMs and PQ/T Hybrids

Combining multiple key-establishment schemes into a single key-establishment scheme can be advantageous for some applications, e.g., during the migration to post-quantum cryptography. The discussions of such schemes in this document will adhere to the terminology established in [23].

A *multi-algorithm key-establishment scheme* combines shared secrets that are generated using two or more key-establishment schemes. The underlying schemes are called the *components* of the overall scheme. In general, it is not necessary that the multi-algorithm scheme has the same interface as its components. In this document, for example, multi-algorithm schemes will always be KEMs, while their components need not be.

A well-designed multi-algorithm scheme will be secure if *at least one* of the component schemes is secure. This may provide some protection against vulnerabilities that are discovered in one of component schemes after deployment. The migration to post-quantum key-establishment techniques, for example, might initially include multi-algorithm solutions that combine one new post-quantum algorithm with one tried-and-tested but quantum-vulnerable (or *traditional*) algorithm. This is sometimes referred to as hybrid PQ/T (post-quantum / traditional) key establishment. For example, X-Wing is a hybrid PQ/T KEM built from two components: ML-KEM (a lattice-based post-quantum KEM) and X25519 (a traditional Diffie-Hellman-style key exchange) [24].

This section outlines approved approaches for multi-algorithm key establishment. Such an approach proceeds in two stages, as follows.

1. **Establish shared secrets.** All component key establishment schemes are run (typically in parallel), resulting in Alice and Bob sharing a collection of shared secrets, one for each component scheme.
2. **Combine shared secrets.** Alice and Bob individually use a *key combiner* to combine their individual shared secrets into a single shared secret each. Approved key combiners are described in Section 5.5.2.

For simplicity, the exposition below focuses on a particular case: constructing a single KEM from two component KEMs. Since both the components and the multi-algorithm scheme in this case are of the same type (i.e., KEMs), the result is called a *composite KEM*. Note that most key-establishment schemes of interest can easily be adapted into KEMs (see, e.g., ECDH-KEM in Section 6.1.1 and RSA-KEM in Section 6.1.2). Moreover, the hybrid PQ/T application typically calls for two component schemes: one post-quantum scheme, and one traditional scheme. The two-algorithm composite KEM described below is easily adapted to other cases, such as combining more than two schemes, or combining KEMs with non-KEMs.

5.5.1. Constructing a Composite KEM

Given two KEMs Π_1 and Π_2, one can construct a composite KEM $\mathcal{C}[\Pi_1, \Pi_2]$ via the following sequence of steps:

1. **Choose parameter sets.** Choose a collection $\mathcal{C}[\Pi_1, \Pi_2]$.ParamSets of parameter sets. Each parameter set will be a pair $p = (p_1, p_2)$, where $p_1 \in \Pi_1$.ParamSets and $p_2 \in \Pi_2$.ParamSets.
2. **Select a key combiner.** Choose a key combiner algorithm `KeyCombine`. The inputs to `KeyCombine` consist of a pair of shared secret keys (one from Π_1 and one from Π_2), as well as a pair of ciphertexts, a pair of encapsulation keys, and a parameter set; the output is a single shared secret key. Section 5.5.2 discusses NIST-approved key combiners.
3. **Construct a composite key-generation algorithm.** When a parameter set $p = (p_1, p_2)$ is input, the algorithm $\mathcal{C}[\Pi_1, \Pi_2]$.KeyGen will perform:

   1. $(\mathsf{ek}_1, \mathsf{dk}_1) \leftarrow \Pi_1.\mathsf{KeyGen}(p_1)$.
   2. $(\mathsf{ek}_2, \mathsf{dk}_2) \leftarrow \Pi_2.\mathsf{KeyGen}(p_2)$.
   3. Output composite encapsulation key $\mathsf{ek}_1 \| \mathsf{ek}_2$.
   4. Output composite decapsulation key $\mathsf{dk}_1 \| \mathsf{dk}_2$.

4. **Construct a composite encapsulation algorithm.** When a parameter set $p = (p_1, p_2)$ and encapsulation key $\mathsf{ek}_1 \| \mathsf{ek}_2$ are input, the algorithm $\mathcal{C}[\Pi_1, \Pi_2].\mathsf{Encaps}$ will perform:
   1. $(K_1, c_1) \leftarrow \Pi_1.\mathsf{Encaps}(p_1, \mathsf{ek}_1)$.
   2. $(K_2, c_2) \leftarrow \Pi_2.\mathsf{Encaps}(p_2, \mathsf{ek}_2)$.
   3. Output combined shared secret key

$$K \leftarrow \texttt{KeyCombine}(K_1, K_2, c_1, c_2, \mathsf{ek}_1, \mathsf{ek}_2, p)\,. \tag{9}$$

   4. Output composite ciphertext $c := c_1 \| c_2$.

5. **Construct a composite decapsulation algorithm.** When a parameter set $p = (p_1, p_2)$, decapsulation key $\mathsf{dk}_1 \| \mathsf{dk}_2$, and ciphertext $c_1 \| c_2$ are input, the algorithm $\mathcal{C}[\Pi_1, \Pi_2].\mathsf{Decaps}$ will perform:
   1. $K_1' \leftarrow \Pi_1.\mathsf{Decaps}(p_1, \mathsf{dk}_1, c_1)$.
   2. $K_2' \leftarrow \Pi_2.\mathsf{Decaps}(p_2, \mathsf{dk}_2, c_2)$.
   3. Output combined shared secret key

$$K' \leftarrow \texttt{KeyCombine}(K_1', K_2', c_1, c_2, \mathsf{ek}_1, \mathsf{ek}_2, p)\,. \tag{10}$$

Note that, since the inputs to `KeyCombine` include the composite encapsulation key, the decapsulating party must retain a copy of that key (or maintain the ability to re-create it) after performing key generation.

General multi-algorithm schemes. The above construction can be extended in the obvious way to composite constructions that use more than two component KEMs. Extending to the case of a completely general multi-algorithm key-establishment scheme can be more complex, as the components in such a scheme can vary widely. For example, such schemes could potentially include pre-shared keys or shared secrets established via Quantum Key Distribution. Still, most multi-algorithm schemes will likely include a step in which a series of shared secrets are combined via a key combiner algorithm of a form similar to `KeyCombine` above. In those cases, an approved key-combiner discussed in Section 5.5.2 **shall** be used.

5.5.2. Approved Key Combiners

This section describes approved methods for combining shared secrets as part of a multi-algorithm key-establishment scheme. Choosing such a method amounts to selecting a key combiner `KeyCombine`. At a minimum, `KeyCombine` accepts two shared secrets as input. Optionally, `KeyCombine` can also accept additional information, such as ciphertexts, encapsulation keys, parameter sets, or other context-dependent data (see, e.g., the composite KEM in Section 5.5.1). As output, `KeyCombine` produces a shared secret key.

This section describes how cryptographic methods standardized in other NIST publications can, under an appropriate interpretation, be used as key combiners. There are two categories of such key combiners:

1. Key combiners from key derivation methods approved in SP 800-56Cr2 [21]
2. Key combiners from key combination methods approved in SP 800-133r2 [22]

Key derivation in SP800-56Cr2, in brief. SP 800-56Cr2 [21] specifies a collection of approved methods for performing key derivation. In SP 800-56Cr2, a key derivation method (KDM) is applied to a shared secret Z generated as specified in SP 800-56A [1] or SP 800-56B [2] along with some additional input, and results in keying material K:

$$K \leftarrow \mathsf{KDM}(Z, \mathtt{OtherInput})\,. \tag{11}$$

The key derivation method KDM can take one of two forms:

1. One-step key derivation. In this case, K is computed by applying a key-derivation function KDF to the concatenation of the two inputs Z and `OtherInput`.

$$K \leftarrow \mathsf{KDF}(Z \| \mathtt{OtherInput})\,. \tag{12}$$

2. Two-step key derivation. In this case, one requires two functions: Extract (which is a randomness extractor) and Expand. The process begins with applying Extract to Z, using a salt as the seed. Expand is then applied to the result along with the remaining part of `OtherInput`.

$$K \leftarrow \mathsf{Expand}(\mathsf{Extract}(\mathtt{salt}, Z), \mathtt{OtherInput})\,. \tag{13}$$

In this method, it is required that extraction is applied to the shared secret Z.

SP 800-56Cr2 describes the specific approved choices of KDF, Extract, and Expand, as well as the format and content of `OtherInput`. These details will not be discussed in this document.

As discussed in Section 5.3, this publication approves the application of SP 800-56Cr2 KDMs to the shared secret keys of approved KEMs. In particular, this means that the quantity Z in Equation (11) (and hence also in (12) and (13)) can be the shared secret key of ML-KEM.

Key combiners from SP800-56C. In both one-step and two-step key derivation, SP 800-56Cr2 allows the shared secret Z to have the form $Z = S_1 \| S_2$, where S_1 is a shared secret generated as specified in SP 800-56A [1] or SP 800-56B [2], while S_2 is a shared secret generated in some other (not necessarily approved) manner. This yields a key combiner $K \leftarrow \mathsf{KDM}(S_1 \| S_2, \texttt{OtherInput})$ for a two-algorithm key-establishment scheme. Since one is free to choose S_2 arbitrarily, one can also combine many shared secrets:

$$K \leftarrow \mathsf{KDM}(S_1 \| S_2 \| \cdots \| S_t, \texttt{OtherInput}) \tag{14}$$

This publication approves the use of the key combiner (14) for any $t > 1$, so long as at least one shared secret (i.e., S_j for some j) is a shared secret generated from the key-establishment methods of SP 800-56A [1] or SP 800-56B [2], or an approved KEM. It is important to note that, in the case where the KDM in the combiner (14) is a two-step method (i.e., using (13)), extraction is performed with all shared secrets as the input.

SP 800-56Cr2 allows `OtherInput` to contain an input that is chosen arbitrarily by the algorithm designer; this optional input is contained in a parameter called `FixedInfo` in SP 800-56Cr2. By choosing `FixedInfo` appropriately, one can also construct approved key combiners of the form (14) that, in addition to shared secrets, also receive additional inputs like encapsulation keys, ciphertexts, parameter sets, and domain separators.

As an example, consider the following simple special case. Choose KDM to be the one-step key derivation method where KDF is a hash function H (chosen from the list of hash functions approved for this purpose by SP 800-56Cr2). Set `OtherInput` to contain only the concatenation of ciphertexts, encapsulation keys, and the parameter set. Then define a key combiner algorithm `KeyCombine` simply by setting

$$\texttt{KeyCombine}(K_1, K_2, c_1, c_2, \mathsf{ek}_1, \mathsf{ek}_2, p) := H(K_1 \| K_2 \| c_1 \| c_2 \| \mathsf{ek}_1 \| \mathsf{ek}_2 \| p)\,. \tag{15}$$

One can then instantiate the composite KEM example from Section 5.5 by using this key combiner. The resulting composite KEM will have a shared secret key whose length is the output length of H.

Key combiners derived from SP 800-133r2. Section 6.3 of SP 800-133r2 [22] provides three approved methods for combining cryptographic keys that were generated in an approved way. These methods can be broadly described as concatenation, XORing, and key extraction using HMAC. Some of these methods can also be applied to just a single key. As discussed in Section 5.3, these methods are approved for key derivation for approved KEMs.

When combining multiple keys $K_1, K_2, \ldots, K_t$, the key-combination methods found in SP 800-133 [22] require every key K_j for $j \in \{1, 2, \ldots, t\}$ to be generated using approved methods. These methods can thus be used directly as key combiners for constructing multi-algorithm schemes in cases where all of the component schemes are approved, and each one produces a key.

5.5.3. Security Considerations for Composite Schemes

The typical goal of a composite KEM construction is to ensure that security will hold if *either* of the component KEMs is secure. There are some important security considerations when constructing composite KEMs.

Theoretical security. The two main security properties that KEMs can satisfy (see Section 3.3) are:

1. IND-CPA security (i.e., security against passive eavesdropping attacks)
2. IND-CCA security (i.e., security against active attacks)

A well-constructed composite KEM $\mathcal{C}[\Pi_1, \Pi_2]$ should preserve the security properties of its component KEMs Π_1 and Π_2. This crucially depends on how the composite KEM is constructed and particularly on the choice of key combiner.

An important example is the case in which the goal is active (i.e., IND-CCA) security, but only one of the two schemes Π_1 and Π_2 is itself IND-CCA (and of course, the designer of the composite scheme does not know which one it is). In this case, the choice of key combiner is particularly relevant here. As shown in [25], the straightforward key combiner

$$K \leftarrow \mathsf{KDF}(K_1 \| K_2) \tag{16}$$

that only uses the two shared secret keys K_1 (of Π_1) and K_2 (of Π_2) does not preserve IND-CCA security. So, for example, the scheme Π_2 could be so broken that $\mathcal{C}[\Pi_1, \Pi_2]$ is not IND-CCA, even if Π_1 is IND-CCA and regardless of what KDF is used.

Therefore, NIST encourages the use of key combiners that generically preserve IND-CCA security. One example of such a key-combiner is as follows [25]. Let H denote a hash function approved for one-step key-derivation in SP 800-56C [21]. Define the key combiner $\mathtt{KeyCombine}_H^{\mathtt{CCA}}$ as follows (recalling the notation of Section 5.5):

- **Inputs from** Π_1: ek_1, c_1, K_1
- **Inputs from** Π_2: ek_2, c_2, K_2
- **Output:** $H(K_1 \| K_2 \| c_1 \| c_2 \| \mathsf{ek}_1 \| \mathsf{ek}_2 \| \mathtt{domain\_separator})$

The `domain_separator` should be used to uniquely identify the composite scheme in use (e.g., Π_1, Π_2, the order of composition, the choice of key combiner and KDF)

Security in practice. While composite schemes are meant to increase security, they necessarily add a layer of additional complexity to the basic KEM framework. This additional complexity will be reflected in implementations and applications and could introduce security vulnerabilities. Moreover, adding composite schemes introduces additional choices in protocols, which could also introduce vulnerabilities (e.g., in the form of "downgrade" attacks). Implementers and users should be aware of the potential challenges in implementing and deploying composite schemes.

6. Examples

This section contains a number of examples. It does not contain any requirements or specific guidance. Instead, its purpose is to aid the reader in understanding some aspects of how KEMs are constructed and used in a manner that is consistent with NIST guidance.

6.1. Examples of KEMs

The following subsections discuss three key-encapsulation mechanisms: ECDH-KEM, RSA-KEM, and ML-KEM. While ECDH and RSA key transport are perhaps not typically described as KEMs, the discussions below will give a high-level description of how both can be naturally viewed as KEMs. The goal of these descriptions is illustrative only. As FIPS 203 already contains a complete description of ML-KEM, the relevant discussion below will simply reference the relevant parts of FIPS 203 [3].

6.1.1. A KEM From Diffie-Hellman

A KEM may be constructed from a Diffie-Hellman (DH) key-agreement scheme. The high-level idea is that, if the two parties in a DH scheme send their messages in sequential order (e.g., Alice first, then Bob), then:

1. the public message and private randomness of Alice can be viewed as an encapsulation key and a decapsulation key (respectively), and
2. the public message and private randomness of Bob can be viewed as a ciphertext and a shared secret (respectively).

For example, a KEM can be constructed from the C(1e, 1s, ECC CDH) Scheme from SP 800-56Ar3 [1] as follows:

- ECDH-KEM.ParamSets. The parameter sets are the same as those specified for ECDH in Section 5.5.1.2 of SP 800-56Ar3.
- ECDH-KEM.KeyGen. The key-generation algorithm is the same as the one specified in Section 5.6.1.2 of SP 800-56Ar3.
- ECDH-KEM.Encaps. To encapsulate, perform Party U's actions from Section 6.2.2.2 of SP 800-56Ar3. The output is the key (i.e., the derived secret keying material) along with the ciphertext (i.e., the ephemeral public key $Q_{e,U}$).
- ECDH-KEM.Decaps. To decapsulate, perform Party V's actions from Section 6.2.2.2 of SP 800-56Ar3. The output key is the derived secret keying material.

Use of this KEM would require that all assumptions for the scheme specified in SP 800-56Ar3 are met and that all necessary assurances have been obtained. In similar ways, KEMs could be constructed from the C(1e, 1s, FFC DH), C(2e, 0s, ECC CDH), and C(2e, 0s, FFC DH) schemes.

6.1.2. A KEM from RSA Secret-Value Encapsulation

As discussed in Section 3.2, any public-key encryption (PKE) scheme can be used to construct a KEM. A concrete example of this is RSA Secret-Value Encapsulation (RSASVE). The high-level idea is described as follows.

1. Alice sends an RSA public-key to Bob. (Optionally, Alice can send some other public information to Bob such as a nonce for key derivation.)

2. Bob generates a secret value and encapsulates it with the RSA public-key to produce the ciphertext. A key is derived from the secret value. The output of encapsulation is the ciphertext and derived key.

3. Alice decapsulates the ciphertext using her RSA private key to obtain the secret value that is used to derive the key.

For example, a KEM can be constructed from RSASVE from SP 800-56Br2 [2] as follows:

1. RSASVE-KEM.$\mathsf{ParamSets}$. The parameter set is the binary length of the modulus as specified as in Table 2, Section 6.3 of SP 800-56Br2, along with the exponent e.

2. RSASVE-KEM.KeyGen. The key generation algorithm is specified in Section 6.3 of SP 800-56Br2 (see also Appendix C.2 of FIPS 186-5).

3. RSASVE-KEM.Encaps. To encapsulate, perform RSASVE.GENERATE as specified in Section 7.2.1.2 of SP 800-56Br2. The output is the secret value (from which to derive a key) and ciphertext. With a nonce for key derivation provided by Party V, this step is the same as the operation of Party U in the KAS1-basic scheme specified in Section 8.2.2 of SP 800-56Br2.

4. RSASVE-KEM.Decaps. To decapsulate, perform RSASVE.RECOVER as specified in Section 7.2.1.3 of SP 800-56Br2. The output key is derived from the secret value output by RSASVE.RECOVER. With a nonce for key derivation (previously provided to Party U), this step is the same as the operation of Party V in the KAS1-basic scheme specified in Section 8.2.2 of SP 800-56Br2.

Use of this KEM would require that all assumptions for the scheme specified in SP 800-56Ar2 are met and that all necessary assurances have been obtained. In similar ways, KEMs could be constructed from RSA-OAEP-basic as specified in Section 9.2.3.

6.1.3. ML-KEM

ML-KEM is a high-performance, general-purpose, lattice-based key-encapsulation mechanism. It is a NIST-approved KEM and was standardized in FIPS 203 [3]. ML-KEM is based on CRYSTALS-Kyber [26], which was a candidate submitted to the NIST PQC standardization process. It is believed to satisfy IND-CCA security (Definition 4), even against adversaries in possession of a cryptanalytically-relevant quantum computer [17, 27, 28]. The asymptotic, theoretical security of ML-KEM is based on the presumed hardness of the Module Learning with Errors (MLWE) problem [29, 30].

FIPS 203 describes ML-KEM directly as a KEM in a manner that closely matches the notation of this document. Specifically, the components of ML-KEM are described in FIPS 203 as follows [3]:

- ML-KEM.ParamSets. There are three parameter sets described in Section 8 of FIPS 203: ML-KEM-512, ML-KEM-768, and ML-KEM-1024.
- ML-KEM.KeyGen. The key generation algorithm of ML-KEM is specified as Algorithm 19 in Section 7.1 of FIPS 203.
- ML-KEM.Encaps. The encapsulation algorithm of ML-KEM is specified as Algorithm 20 in Section 7.2 of FIPS 203.
- ML-KEM.Decaps. The decapsulation algorithm of ML-KEM is specified as Algorithm 21 in Section 7.3 of FIPS 203.

Note that this document treats parameter sets as an explicit input for the KEM algorithms KeyGen, Encaps, and Decaps. By contrast, the algorithms of ML-KEM as described in FIPS 203 expect the chosen parameter set to be stored in a set of global variables that are accessible to each of the algorithms of ML-KEM. This is only a difference in presentation and does not imply any particular implementation requirement.

6.2. Examples of Applications of KEMs

This section provides a high-level overview of a few example applications of KEMs.

6.2.1. Hybrid Public-Key Encryption (HPKE)

A KEM can be combined with a symmetric-key encryption scheme to yield very efficient public-key encryption. This is sometimes referred to as a hybrid PKE (HPKE), which should not be confused with "hybrid PQC." The former refers to combining a KEM with symmetric-key encryption, and the latter refers to combining a quantum-vulnerable key-establishment scheme with a quantum-resistant KEM.

The prescription for constructing an HPKE scheme is as follows. Let Π be a KEM, and let $\Xi = (\mathsf{Encrypt}, \mathsf{Decrypt})$ be a symmetric-key encryption scheme. One then constructs a PKE called $\mathtt{HPKE}$ as follows:

- $\mathtt{HPKE}.\mathsf{ParamSets} = \Pi.\mathsf{ParamSets}$
- $\mathtt{HPKE}.\mathsf{KeyGen} = \Pi.\mathsf{KeyGen}$
- $\mathtt{HPKE}.\mathsf{Encrypt}$: Using input p, ek, and message m:

1. Compute $(K, c_\Pi) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}_A)$;
2. Compute $c_\Xi \leftarrow \Xi.\mathsf{Encrypt}(K, m)$; and
3. Output (c_Π, c_Ξ).

- HPKE.Decrypt: Using input p, dk, and (c_Π, c_Ξ),
  1. Compute $K' \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c_\Pi)$; and
  2. Output $m' \leftarrow \Xi.\mathsf{Decrypt}(K', c_\Xi)$.

Here, the keys of Ξ are assumed to be the same length as the shared secret keys produced by Π. If not, appropriate key-derivation steps (see Section 5.3) can be added to HPKE.Encrypt and HPKE.Decrypt to transform the shared secret key of Π into a key that is appropriate for use with Ξ.

Figure 6 shows the procedure for sending an encrypted message m from Bob to Alice using HPKE.

$$
\begin{array}{lcl}
\underline{\textbf{Alice}\qquad} & & \underline{\textbf{Bob}\qquad} \\
(\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p) & & \\
 & \xrightarrow{\mathsf{ek},\, p} & \\
 & & (K, c_\Pi) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}) \\
 & & c_\Xi \leftarrow \Xi.\mathsf{Encrypt}(K, m) \\
 & \xleftarrow{c_\Pi,\, c_\Xi} & \\
K' \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c_\Pi) & & \\
m \leftarrow \Xi.\mathsf{Decrypt}(K', c_\Xi) & &
\end{array}
$$

Fig. 6. Sending a message using HPKE

This same procedure can also be used to perform key transport by choosing m uniformly at random.

6.2.2. Static-Ephemeral Key Establishment

Most applications of key establishment require at least one party to authenticate their identity, such as KEM key establishment with a static encapsulation key that is authenticated by a chain of certificates. A description of such a procedure is given below and depicted in Figure 7.

1. At the outset, Alice has a long-term key pair that she generated earlier via $(\mathsf{ek}, \mathsf{dk}) \leftarrow \Pi.\mathsf{KeyGen}(p)$. Here, Π is some KEM, and p is some parameter set of Π. Alice also

Alice (server)		**Bob (client)**
1. static: $(\mathsf{ek}, \mathsf{dk})$		
$\mathsf{cert}[\mathsf{ek}, p, \mathsf{Alice}]$		
	$\xrightarrow{\mathsf{cert}[\mathsf{ek}, p, \mathsf{Alice}]}$	
2.		if $\mathsf{cert}[\mathsf{ek}, p, \mathsf{Alice}]$invalid, **abort**.
		$(K_B, c) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek})$
	$\xleftarrow{c}$	
3. $K_A \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}, c)$		
4. $t \leftarrow \mathtt{MAC}(K_A, c)$		
	$\xrightarrow{t}$	
		if $\mathtt{MAC}.\mathsf{Ver}(K_B, c, t)$rejects, **abort**.
5. result: K_A		result: K_B

Fig. 7. Static-ephemeral key establishment using a KEM

has a certificate $\mathsf{cert}[\mathsf{ek}, p, \mathsf{Alice}]$ that contains ek and p and associates them both to Alice's identity.

2. When Bob wants to connect to Alice, he acquires $\mathsf{cert}[\mathsf{ek}, p, \mathsf{Alice}]$ (e.g., from Alice), verifies that the certificate is valid, and extracts ek and p from the certificate. He then performs encapsulation with ek, saves the resulting shared secret key K_B, and sends the ciphertext c to Alice.
3. Alice decapsulates c and gets a shared secret key K_A.
4. Alice and Bob perform key confirmation to ensure that key establishment was successful. Alice uses a message authentication code $\mathtt{MAC}$ to generate a tag $t \leftarrow \mathtt{MAC}(K_A, c)$ for the ciphertext c and sends t to Bob. Bob then runs $\mathtt{MAC}$ verification and aborts unless the tag t is accepted.
5. Alice and Bob can now use their shared secret keys to communicate efficiently and securely using symmetric-key cryptography.

It is assumed that if the certificate chain was valid, then only Alice was capable of performing decapsulation of ciphertexts encapsulated using ek.

6.2.3. Ephemeral Authenticated Key Establishment

This section describes an alternative approach to unilaterally authenticated key establishment using a KEM. Compared to the example in Section 6.2.2, Alice and Bob will now have the opposite roles in the protocol. Specifically, Bob is now the authenticated party (e.g., a web server), while Alice is the unauthenticated party (e.g., a browser client). KEM key generation will now be performed by the *client* (i.e., Alice), and Alice will discard the KEM key pair once the connection is established. As the server (i.e., Bob) no longer uses a static KEM encapsulation key, he will need to establish his identity through other means. In this example, that will be done via a digital signature verification key provided in a certificate and verified as part of a certificate chain.

The protocol proceeds as follows (see Figure 8.) Let Σ be a digital signature scheme with algorithms Σ.KeyGen, Σ.Sign, and Σ.Ver. Recall that KEM key pairs are denoted by ek (encaps key, public) and dk (decaps key, private). For the digital signature, key pairs are denoted by vk (verification key, public) and sk (signing key, private).

Alice (client)		**Bob (server)**
1.		static: $(\mathsf{vk}_B, \mathsf{sk}_B)$ $\mathsf{cert}[\mathsf{vk}_B, \mathsf{Bob}]$
2. $(\mathsf{ek}_A, \mathsf{dk}_A) \leftarrow \Pi.\mathsf{KeyGen}(p)$		
	$\xrightarrow{\mathsf{ek}_A,\, p}$	
3.		$(K_B, c_B) \leftarrow \Pi.\mathsf{Encaps}(p, \mathsf{ek}_A)$ $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sk}_B, \mathsf{transcript})$ $(K'_B, K''_B) \leftarrow \mathsf{KDF}(K_B)$
	$\xleftarrow{\mathsf{cert}[\mathsf{vk}_B, \mathsf{Bob}],\, \sigma,\, c_B}$	
4. if $\mathsf{cert}[\mathsf{vk}_B$, Bob$]$invalid, **abort**. if $\Sigma.\mathsf{Ver}(\mathsf{vk}_B, \sigma, \mathsf{transcript}) = \perp$, **abort**. $K_A \leftarrow \Pi.\mathsf{Decaps}(p, \mathsf{dk}_A, c_B)$ $(K'_A, K''_A) \leftarrow \mathsf{KDF}(K_A)$		
5. result: K'_A, K''_A		result: K'_B, K''_B

Fig. 8. Using a KEM for key establishment with unilateral authentication

1. The protocol begins with Alice (who will not need to authenticate herself) and Bob (who has previously generated a static digital signature key pair $(\mathsf{vk}_B, \mathsf{sk}_B)$).
2. Alice generates a KEM key pair $(\mathsf{ek}_A, \mathsf{dk}_A)$ and sends the encapsulation key ek_A and the relevant parameter set p to Bob, keeping the decapsulation key dk_A private.
3. Bob performs encapsulation using ek_A, which results in a KEM ciphertext c_B and a shared secret key K_B. Bob then uses his private signing key sk_B to sign the transcript of all communications with Alice, including what he will send in this transmission.

This transcript includes ek_A, p, vk_B, c_B, and a certificate chain $\mathsf{cert}[\mathsf{vk}_B, \mathsf{Bob}]$ that establishes that vk_B is associated with Bob's identity. He then sends the ciphertext, certificate chain, and signature to Alice. Finally, he applies a key-derivation function KDF to K_B in order to produce two symmetric keys K_B' and K_B'', destroys K_B, and keeps K_B' and K_B'' private.

4. Next, Alice performs two checks. First, she checks the validity of Bob's claimed certificate chain with the appropriate certification authority. Second, she verifies Bob's signature on the transcript. If either check fails, Alice aborts. Otherwise, she decapsulates c_B and keeps the resulting shared secret key K_A private. She also derives two keys K_A' and K_A'' via KDF applied to K_A.
5. Alice and Bob can now use the keys K_A' and K_A'' for symmetric-key cryptography. For example, they could use K_A' for encryption and K_A'' for authentication.

References

[1] Barker EB, Chen L, Roginsky A, Vassilev A, Davis R (2018) Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-56Ar3. https://doi.org/10.6028/NIST.SP.800-56Ar3

[2] Barker EB, Chen L, Roginsky A, Vassilev A, Davis R, Simon S (2019) Recommendation for pair-wise key-establishment using integer factorization cryptography (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-56Br2. https://doi.org/10.6028/NIST.SP.800-56Br2

[3] National Institute of Standards and Technology (2024) Module-lattice-based key-encapsulation mechanism standard (U.S. Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publications (FIPS PUBS) NIST FIPS 203, August, 2024. https://doi.org/10.6028/NIST.FIPS.203

[4] Moody D, Perlner R, Regenscheid A, Robinson A, Cooper D (2024) Transition to post-quantum cryptography standards (National Institute of Standards and Technology),

[5] National Institute of Standards and Technology (2001) Security requirements for cryptographic modules (U.S. Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publications (FIPS PUBS) NIST FIPS 140-3, March 03, 2019. https://doi.org/10.6028/NIST.FIPS.140-3

[6] Barker EB, Kelsey J (2015) Recommendation for random number generation using deterministic random bit generators (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-90Ar1. https://doi.org/10.6028/NIST.SP.800-90Ar1

[7] Turan MS, Barker E, Kelsey J, McKay K, Baish M, Boyle M (2018) Recommendation for the entropy sources used for random bit generation (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-90B. https://doi.org/10.6028/NIST.SP.800-90B

[8] Barker EB, Kelsey J, Roginsky A, Turan MS (2024) Recommendation for random bit generator (RBG) constructions (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-90C4pd. https://doi.org/10.6028/NIST.SP.800-90C.4pd

[9] Barker E, Branstad D, Smid M (2015) A Profile for U.S. Federal Cryptographic Key Management Systems (CKMS) (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-152. https://doi.org/10.6028/NIST.SP.800-152

[10] Shoup V (2001) A proposal for an ISO standard for public key encryption, Cryptology ePrint Archive, Paper 2001/112. Available at https://eprint.iacr.org/2001/112.

[11] Cramer R, Shoup V (2003) Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing* 33(1):167–226.

[12] Herranz J, Hofheinz D, Kiltz E (2006) Some (in)sufficient conditions for secure hybrid encryption, Cryptology ePrint Archive, Paper 2006/265. Available at https://eprint.iacr.org/2006/265.

[13] Brainard J, Kaliski B, Turner S, Randall J (2010) Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS), RFC 5990. https://doi.org/10.17487/RFC5990.

[14] American National Standards Institute (2011) ANSI X9.63, Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography.

[15] American National Standards Institute (2007) ANSI X9.44, Public Key Cryptography for the Financial Services Industry: Key Establishment Using Integer Factorization Cryptography.

[16] National Institute of Standards and Technology (2023) Module-lattice-based key-encapsulation mechanism standard (U.S. Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publications (FIPS PUBS) NIST FIPS 203 ipd, August, 2023. https://doi.org/10.6028/NIST.FIPS.203.ipd

[17] Katz J, Lindell Y (2020) *Introduction to Modern Cryptography* (Chapman & Hall/CRC), 3rd Ed.

[18] Barker E (2020) Recommendation for Key Management: Part 1 – General (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-57pt1r5. https://doi.org/10.6028/NIST.SP.800-57pt1r5

[19] Dworkin M (2007) Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-38D. https://doi.org/10.6028/NIST.SP.800-38D

[20] Chen L (2022) Recommendation for key derivation using pseudorandom functions (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-108r1. https://doi.org/10.6028/NIST.SP.800-108r1

[21] Barker EB, Chen L, Davis R (2020) Recommendation for key-derivation methods in key-establishment schemes (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-56Cr2. https://doi.org/10.6028/NIST.SP.800-56Cr2

[22] Barker EB, Roginsky A, Davis R (2020) Recommendation for cryptographic key generation (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-13r2. https://doi.org/10.6028/NIST.SP.800-133r2

[23] Driscoll F, Parsons M, Hale B (2024) Terminology for Post-Quantum Traditional Hybrid Schemes (Internet Engineering Task Force), Internet-Draft draft-ietf-pquip-pqt-hybrid-terminology-05. Work in Progress. Available at https://datatracker.ietf.org/doc/draft-ietf-pquip-pqt-hybrid-terminology/05/.

[24] Barbosa M, Connolly D, Duarte JD, Kaiser A, Schwabe P, Varner K, Westerbaan B (2024) X-wing. *IACR Communications in Cryptology* 1(1). https://doi.org/10.62056/a3qj89n4e

[25] Giacon F, Heuer F, Poettering B (2018) KEM combiners. *Public-Key Cryptography – PKC 2018*, eds Abdalla M, Dahab R (Springer International Publishing, Cham), pp 190–218.

[26] Avanzi R, Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashekvsky V, Schanck JM, Schwabe P, Seiler G, Stehlé’ D (2021) CRYSTALS-Kyber Algorithm Specifications and Supporting Documentation (version 3.02). Available at https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[27] Avanzi R, Bos J, Ducas L, Kiltz E, Lepoint T, Lyubashevsky V, Schanck JM, Schwabe P, Seiler G, Stehlé D (2020) CRYSTALS-Kyber algorithm specifications and supporting documentation, Third-round submission to the NIST’s post-quantum cryptography standardization process. Available at https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[28] Almeida JB, Olmos SA, Barbosa M, Barthe G, Dupressoir F, Grégoire B, Laporte V, Léchenet JC, Low C, Oliveira T, Pacheco H, Quaresma M, Schwabe P, Strub PY (2024) Formally verifying Kyber episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt, Cryptology ePrint Archive, Paper 2024/843. Available at https://eprint.iacr.org/2024/843.

[29] Regev O (2005) On lattices, learning with errors, random linear codes, and cryptography. *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing* STOC ’05 (Association for Computing Machinery, New York, NY, USA), pp 84––93. https://doi.org/10.1145/1060590.1060603.

[30] Langlois A, Stehlé D (2015) Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography* 75(3):565–599. https://doi.org/10.1007/s10623-014-9938-4.

[31] Sönmez Turan M, Brandão L (2010) Keyed-Hash Message Authentication Code (HMAC): Specification of HMAC and Recommendations for Message Authentication (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-224 ipd. https://doi.org/10.6028/NIST.SP.800-224.ipd

[32] Dworkin M (2010) Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-38B. https://doi.org/10.6028/NIST.SP.800-38B

[33] Kelsey J, Chang SJ, Perlner R (2016) SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication (SP) NIST SP 800-185. https://doi.org/10.6028/NIST.SP.800-185

Appendix A. Cryptographic Components

Appendix A.1. Message Authentication Codes (MACs)

A message authentication code (MAC) algorithm defines a family of cryptographic functions that is parameterized by a symmetric key. It is computationally infeasible to determine the MAC of a newly formed $MacData$ output value without knowledge of the $MacKey$ value, even if one has seen the MACs corresponding to other $MacData$ values that were computed using that same $MacKey$ value.

The input to a MAC algorithm includes a symmetric key $MacKey$ and a binary data string $MacData$ that serves as the "message." That is, a MAC computation is represented as MAC($MacKey$, $MacData$). In this recommendation, a MAC algorithm is used if key confirmation is performed during key establishment (see Section 5.4).

When key confirmation requires the use of a MAC, it **shall** be an approved MAC algorithm (i.e., HMAC, AES-CMAC, or KMAC). HMAC is specified in SP 800-224 [31] and requires the use of an approved hash function. AES-CMAC is specified in SP 800-38B [32] for the AES block cipher algorithm specified in FIPS 197. KMAC is specified in SP 800-185 [33].

When a MAC tag (MacTag) is used for key confirmation, an entity **shall** compute the MAC tag on received or derived data using a MAC algorithm with a $MacKey$ that is determined from a shared secret key. The MAC tag is sent to the other entity participating in the key-establishment scheme in order to provide assurance that the shared secret key or derived keying material was correctly computed. MAC-tag computation and verification are defined in Sections A.1.3.1 and A.1.3.2.

MAC Tag Computation for Key Confirmation. Key confirmation can be performed as one or more additional steps in a KEM scheme. The computation of a MacTag is represented as follows:

$$MacTag = T_{MacTagBits}[\mathrm{MAC}(MacKey, MacData)].$$

To compute a MacTag:

1. The agreed-upon MAC algorithm (see Section A.1.3) is used with $MacKey$ to compute the MAC on $MacData$, where $MacKey$ is a symmetric key, and $MacData$ represents the input "message" data. The minimum length of $MacKey$ is specified in Table 1.

   $MacKey$ is obtained from the $Derived\_Keying\_Material$ when a KEM scheme employs key confirmation, as specified in Section 5.4.

   The output $MacOutput$ of the MAC algorithm is a bit string whose length in bits is $MacOutputBits$.

Table 1. Approved MAC algorithms for key confirmation

Mac Algorithm	MacOutputBits	Permissible KC_Key Lengths (μ bits)	Supported Security Strengths for Key Confirmation (s bits)
HMAC_SHA-256	256	$s \leq \mu \leq 512$	$128 \leq s \leq 256$
HMAC_SHA-512/256	256		
HMAC_SHA-384	384		
HMAC_SHA-512	512		
HMAC_SHA3-256	256		
HMAC_SHA3-384	384		
HMAC_SHA3-512	512		
KMAC128	$\leq 2^{2040} - 1$		$s = 128$
KMAC256			$128 \leq s \leq 256$
AES-128-CMAC	128	$\mu = 128$	$s = 128$
AES-192-CMAC	128	$\mu = 192$	$128 \leq s \leq 192$
AES-256-CMAC	128	$\mu = 256$	$128 \leq s \leq 256$

2. Those bits are input to the truncation function $T_{MacTagBits}$, which returns the leftmost (i.e., initial) bits of $MacOutput$ to be used as the value of $MacTag$. $MacTagBits$ shall be less than or equal to $MacOutputBits$. When $MacTagBits$ equals $MacOutputBits$, $T_{MacTagBits}$ acts as the identity function. The minimum value for $MacTagBits$ is 64, as specified in Section 5.4.1.

MacTag Verification for Key Confirmation. To verify a received $MacTag$ (i.e., received during key confirmation), a new MacTag $MacTag'$ is computed using the values of $MacKey$, $MacTagBits$, and $MacData$ possessed by the recipient (as specified in Section 5.4.1). $MacTag'$ is compared with the received $MacTag$. If their values are equal, then it may be inferred that the same $MacKey$, $MacTagBits$, and $MacData$ values were used in the two MacTag computations.

Appendix A.2. Random Bit Generators

When this recommendation requires the use of a randomly generated value (e.g., for obtaining the randomness use in KeyGen and Encaps), the values **shall** be generated using an approved random bit generator that supports the targeted security strength (see the SP 800-90 series of publications).

Appendix A.3. Nonces

A nonce is a time-varying value with a negligible chance of repeating (where the meaning of "negligible" may be application-specific). A decapsulator may be required to provide a public nonce that is used for key-confirmation purposes. This circumstance arises when the decapsulator's public key is static.

A nonce may be composed of one or more of the following components, though other components may also be appropriate:

1. A random bit string that is generated anew for each nonce using an approved random bit generator. A nonce containing a component of this type is called a random nonce.

2. A timestamp of sufficient resolution so that it is different each time it is used.

3. A monotonically increasing sequence number.

4 A combination of a timestamp and a monotonically increasing sequence number such that the sequence number is reset when and only when the timestamp changes. For example, a timestamp may show the date but not the time of day, so a sequence number is appended that will not repeat during a particular day.

Whenever a nonce is required for key-confirmation purposes as specified in this recommendation, it should be a random nonce containing a random bit string output from an approved random bit generator, where both the security strength supported by the instantiation of the random bit generator and the bit length of the random bit string are greater than or equal to the targeted security strength of the key-establishment scheme in which the nonce is used during key confirmation. When feasible, the bit length of the random bit string should be at least twice the targeted security strength. For details concerning the security strength supported by an instantiation of a random bit generator, see the SP 800-90 series of publications [6**?** , 7].

As part of the proper implementation of this recommendation, system users and/or agents trusted to act on their behalf should determine that the components selected for inclusion in any required nonces meet their security requirements.