INTERNATIONAL STANDARD

**ISO/IEC 18033-2**

First edition
2006-05-01

# Information technology — Security techniques — Encryption algorithms —

## Part 2: Asymmetric ciphers

*Technologies de l'information — Techniques de sécurité — Algorithmes de chiffrement —*

*Partie 2: Chiffres asymétriques*



Reference number
ISO/IEC 18033-2:2006(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

 

# Contents

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 18033-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 27, *IT Security techniques*.

ISO/IEC 18033 consists of the following parts, under the general title *Information technology — Security techniques — Encryption algorithms*:

— *Part 1: General*

— *Part 2: Asymmetric ciphers*

— *Part 3: Block ciphers*

— *Part 4: Stream ciphers*

## Introduction

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this International Standard may involve the use of patents.

The ISO and IEC take no position concerning the evidence, validity and scope of this patent right. The holder of this patent right has assured the ISO and IEC that he is willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statement of the holder of this patent right is registered with the ISO and IEC. Information may be obtained from:

***ISO/IEC JTC 1/SC 27 Standing Document 8 (SD8) "Patent Information"***

Standing Document 8 (SD8) is publicly available at: http://www.ni.din.de/sc27

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

# Information technology — Security techniques — Encryption algorithms —

# Part 2: Asymmetric ciphers



## 1 Scope

This part of ISO/IEC 18033 specifies several asymmetric ciphers. These specifications prescribe the functional interfaces and correct methods of use of such ciphers in general, as well as the precise functionality and cipher text format for several specific asymmetric ciphers (although conforming systems may choose to use alternative formats for storing and transmitting cipher-texts).

A normative annex (Annex A) gives ASN.1 syntax for object identifiers, public keys, and parameter structures to be associated with the algorithms specified in this part of ISO/IEC 18033.

However, these specifications do not prescribe protocols for reliably obtaining a public key, for proof of possession of a private key, or for validation of either public or private keys; see ISO/IEC 11770-3 for guidance on such key management issues.

The asymmetric ciphers that are specified in this part of ISO/IEC 18033 are indicated in Clause 7.6.

NOTE Briefly, the asymmetric ciphers are:

— ECIES-HC; PSEC-HC; ACE-HC: generic hybrid ciphers based on ElGamal encryption;
— RSA-HC: a generic hybrid cipher based on the RSA transform;
— RSAES: the OAEP padding scheme applied to the RSA transform;
— HIME(R): a scheme based on the hardness of factoring.

## 2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9797-1:1999, *Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher*

ISO/IEC 9797-2:2002, *Information technology — Security techniques— Message Authentication Codes (MACs) — Part 2: Mechanisms using a dedicated hash-function*

ISO/IEC 10118-2:2000, *Information technology — Security techniques — Hash-functions — Part 2: Hash-functions using an n-bit block cipher*

ISO/IEC 10118-3:2004, *Information technology — Security techniques — Hash-functions — Part 3: Dedicated hash-functions*

ISO/IEC 18033-3:2005, *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*

# 3 Definitions

For the purposes of this document, the following terms and definitions apply.

NOTE Where appropriate, forward references are given to clauses which contain more detailed definitions and/or further elaboration.

**3.1**
**asymmetric cipher**
system based on asymmetric cryptographic techniques whose public transformation is used for encryption and whose private transformation is used for decryption
[ISO/IEC 18033-1]
NOTE See Clause 7.

**3.2**
**asymmetric cryptographic technique**
cryptographic technique that uses two related transformations, a public transformation (defined by the public key) and a private transformation (defined by the private key). The two transformations have the property that, given the public transformation, it is computationally infeasible to derive the private transformation.
[ISO/IEC 11770-1:1996]

**3.3**
**asymmetric key pair**
pair of related keys, a public key and a private key, where the private key defines the private transformation and the public key defines the public transformation
[ISO/IEC 9798-1:1997]
NOTE See Clauses 7, 8.1.

**3.4**
**bit**
one of the two symbols `0' or `1'
NOTE See Clause 5.2.1.

**3.5**
**bit string**
sequence of bits
NOTE See Clause 5.2.1.

**3.6**
**block**
string of bits of a defined length
[ISO/IEC 18033-1]
NOTE In this part of ISO/IEC 18033, a block will be restricted to be an octet string (interpreted in a natural way as a bit string).

**3.7**
**block cipher**
symmetric cipher with the property that the encryption algorithm operates on a block of plain-text, i.e., a string of bits of a defined length, to yield a block of cipher text
[ISO/IEC 18033-1]
NOTE See Clause 6.4.
NOTE In this part of ISO/IEC 18033, plaintext/cipher text blocks will be restricted to be octet strings (interpreted in a natural way as bit strings).

**3.8**
**cipher**
cryptographic technique used to protect the confidentiality of data, and which consists of three component processes: an encryption algorithm, a decryption algorithm, and a method for generating keys
[ISO/IEC 18033-1]

**3.9**
**cipher text**
data which has been transformed to hide its information content
[ISO/IEC 10116:1997]

**3.10**
**concrete group**
explicit description of a finite abelian group, together with algorithms for performing the group operation and for encoding and decoding group elements as octet strings
NOTE See Clause 10.1.

**3.11**
**cryptographic hash function**
function that maps octet strings of any length to octet strings of fixed length, such that it is computationally infeasible to find correlations between inputs and outputs, and such that given one part of the output, but not the input, it is computationally infeasible to predict any bit of the remaining output. The precise security requirements depend on the application.
NOTE See Clause 6.1.

**3.12**
**data encapsulation mechanism**
cryptographic mechanism, based on symmetric cryptographic techniques, which protects both the confidentiality and the integrity of data
NOTE See Clause 8.2.

**3.13**
**decryption**
reversal of the corresponding encryption
[ISO/IEC 11770-1:1996]

**3.14**
**decryption algorithm**
process which transforms ciphertext into plaintext
[ISO/IEC 18033-1]

**3.15**
**encryption**
(reversible) transformation of data by a cryptographic algorithm to produce cipher text, i.e., to hide the information content of the data
[ISO/IEC 9797-1]

**3.16**
**explicitly given finite field**
finite field that is represented explicitly in terms of its characteristic and a multiplication table for a basis of the field over the underlying prime field
NOTE See Clause 5.3.

**3.17**
**encryption algorithm**
process which transforms plaintext into cipher text
[ISO/IEC 18033-1]

**3.18**
**encryption option**
option that may be passed to the encryption algorithm of an asymmetric cipher, or of a key encapsulation mechanism, to control the formatting of the output cipher text
NOTE See Clauses 7, 8.1.

**3.19**
**field**
mathematical notion of a field, i.e., a set of elements, together with binary operations for addition and multiplication on this set, such that the usual field axioms apply

**3.20**
**finite abelian group**
group such that the underlying set of elements is finite, and such that the underlying binary operation is commutative

**3.21**
**finite field**
field such that the underlying set of elements is finite

**3.22**
**group**
mathematical notion of a group, i.e., a set of elements, together with a binary operation on this set, such that the usual group axioms apply

**3.23**
**hybrid cipher**
asymmetric cipher that combines both asymmetric and symmetric cryptographic techniques

**3.24**
**key**
sequence of symbols that controls the operation of a cryptographic transformation (e.g., encryption, decryption)
[ISO/IEC 11770-1:1996]

**3.25**
**key derivation function**
a function that maps octet strings of any length to octet strings of an arbitrary, specified length, such that it is computationally infeasible to find correlations between inputs and outputs, and such that given one part of the output, but not the input, it is computationally infeasible to predict any bit of the remaining output. The precise security requirements depend on the application.
NOTE See Clause 6.2.

**3.26**
**key encapsulation mechanism**
similar to an asymmetric cipher, but the encryption algorithm takes as input a public key and generates a secret key and an encryption of this secret key
NOTE See Clause 8.1.

**3.27**
**key generation algorithm**
method for generating asymmetric key pairs
NOTE See Clauses 7, 8.1.

**3.28**
**label**
octet string that is input to both the encryption and decryption algorithms of an asymmetric cipher, and of a data encapsulation mechanism. A label is public information that is bound to the cipher text in a non-malleable way
NOTE See Clauses 7, 8.2.

**3.29**
**length**
length of a string of digits or the representation of an integer
Specifically:
(1) length of a bit string is the number of bits in the string
NOTE See Clause 5.2.1.
(2) length of an octet string is the number of octets in the string
NOTE See Clause 5.2.2.
(3) bit length of a non-negative integer n is the number of bits in its binary representation, i.e., $\mathrm{dlog}_2(n + 1)$
NOTE See Clause 5.2.4.
(4) octet length of a non-negative integer n is the number of digits in its representation base 256, i.e., $\mathrm{dlog}_{256}(n + 1)$
NOTE See Clause 5.2.4.

**3.30**
**message authentication code (MAC)**
string of bits which is the output of a MAC algorithm
[ISO/IEC 9797-1]
NOTE See Clause 6.3.
NOTE In this part of ISO/IEC 18033, a MAC will be restricted to be an octet string (interpreted in a natural way as a bit string).

**3.31**
**MAC algorithm**
algorithm for computing a function which maps strings of bits and a secret key to fixed-length strings of bits, satisfying the following two properties:
- for any key and any input string, the function can be computed efficiently;
- for any fixed key, and given no prior knowledge of the key, it is computationally infeasible to compute the function value on any new input string, even given knowledge of the set of input strings and corresponding function values, where the value of the ith input string may have been chosen after observing the value of the first *i* - 1 function values.
[ISO/IEC 9797-1]
NOTE See Clause 6.3.
NOTE In this part of ISO/IEC 18033, the input and output strings of a MAC algorithm will be restricted to be octet strings (interpreted in a natural way as bit strings).

**3.32**
**octet**
a bit string of length 8
NOTE See Clause 5.2.2.

**3.33**
**octet string**
a sequence of octets
NOTE See Clause 5.2.2.
NOTE When appropriate, an octet string may be interpreted as a bit string, simply by concatenating all of the component octets.

**3.34**
**plaintext**
unencrypted information
[ISO/IEC 10116:1997]

**3.35**
**prefix free set**
a set S of bit/octet strings such that there do not exist strings $x \neq y \in S$ such that x is a prefix of y

**3.36**
**primitive**
a function used to convert between data types

**3.37**
**private key**
the key of an entity's asymmetric key pair which should only be used by that entity
[ISO/IEC 11770-1:1996]
NOTE See Clauses 7, 8.1.

**3.38**
**public key**
the key of an entity's asymmetric key pair which can be made public
[ISO/IEC 11770-1:1996]
NOTE See Clauses 7, 8.1.

**3.39**
**secret key**
key used with symmetric cryptographic techniques by a specified set of entities
[ISO/IEC 11770-3:1999]

**3.40**
**symmetric cipher**
cipher based on symmetric cryptographic techniques that uses the same secret key for both the encryption and decryption algorithms
[ISO/IEC 18033-1]

**3.41**
**system parameters**
choice of parameters that selects a particular cryptographic scheme or function from a family of cryptographic schemes or functions

## 4 Symbols and notation

For the purposes of this document, the following symbols and notation apply.

NOTE Where appropriate, forward references are given to clauses which contain more detailed definitions and/or further elaboration.

| | |
|---|---|
| $\lfloor x \rfloor$ | the largest integer less than or equal to the real number x. For example, $\lfloor 5 \rfloor = 5, \lfloor 5.3 \rfloor = 5$, and $\lfloor -5.3 \rfloor = -6$ |
| $\lceil x \rceil$ | the smallest integer greater than or equal to the real number $x$. For example, $\lceil 5 \rceil = 5, \lceil 5.3 \rceil = 6$, and $\lceil -5.3 \rceil = -6$ |
| $[a..b]$ | the interval of integers from $a$ to $b$, including both $a$ and $b$ |
| $[a..b)$ | the interval of integers from $a$ to $b$, including $a$ but not $b$ |
| $\lvert X \rvert$ | if X is a finite set, then the cardinality of X; if X is a finite abelian group or a finite field, then the cardinality of the underlying set of elements; if X is a real number, then the absolute value of X; if X is a bit/octet string, then the length in bits/octets of the string<br>NOTE See Clauses 5.2.1, 5.2.2. |
| $x \oplus y$ | if $x$ and $y$ are bit/octet strings of the same length, the bit-wise exclusive-or (XOR) of the two strings.<br>NOTE See Clauses 5.2.1, 5.2.2. |
| $\langle x_1, \ldots, x_l \rangle$ | if $x_1, \ldots, x_l$ are bits/octets, the bit/octet string of length l consisting of the bits/octets x1; : : : ; xl, in the given order<br>NOTE See Clauses 5.2.1, 5.2.2. |
| $x \lvert\lvert y$ | if $x$ and $y$ are bit/octet strings, the concatenation of the two strings $x$ and $y$, resulting in the string consisting of x followed by $y$<br>NOTE See Clauses 5.2.1, 5.2.2. |
| gcd($a$, $b$) | for integers $a$ and $b$, the greatest common divisor of $a$ and $b$, i.e., the largest positive integer that divides both $a$ and $b$ (or 0 if $a = b = 0$) |
| $a \mid b$ | relation between integers $a$ and $b$ that holds if and only if $a$ divides $b$, i.e., there exists an integer $c$ such that $b = ac$ |
| $a \equiv b \pmod{n}$ | for a non-zero integer $n$, a relation between integers $a$ and $b$ that holds if and only if $a$ and $b$ are congruent modulo $n$, i.e., $n \mid (a - b)$ |
| $a \bmod n$ | for integer $a$ and positive integer $n$, the unique integer $r \in [0 \,..\, n)$ such that $r \equiv a \pmod{n}$ |
| $a^{-1} \bmod n$ | for integer a and positive integer n, such that gcd$(a; n) = 1$, the unique integer $b \in [0..n)$ such that $ab \equiv 1 \pmod{n}$ |
| $F^*$ | for a field $F$, the multiplicative group of units of $F$ |
| $0_F$ | for a field $F$, the additive identity (zero element) of $F$ |
| $1_F$ | for a field $F$, the multiplicative identity of $F$ |
| *BS2IP* | bit string to integer conversion primitive<br>NOTE See Clause 5.2.5. |

| | |
|---|---|
| *EC2OSP* | elliptic curve to octet string conversion primitive. *(See Clause 5.4.3.)* |
| *FE2OSP* | field element to octet string conversion primitive. *(See Clause 5.3.1.)* |
| *FE2IP* | field element to integer conversion primitive. *(See Clause 5.3.1.)* |
| *I2BSP* | integer to bit string conversion primitive. *(See Clause 5.2.5.)* |
| *I2OSP* | integer to octet string conversion primitive. *(See Clause 5.2.5.)* |
| *OS2ECP* | octet string to elliptic curve conversion primitive. *(See Clause 5.4.3.)* |
| *OS2FEP* | octet string to field element conversion primitive. *(See Clause 5.3.1.)* |
| *OS2IP* | octet string to integer conversion primitive. *(See Clause 5.2.5.)* |
| $Oct(m)$ | the octet whose integer value is $m$. *(See Clause 5.2.4.)* |
| $\mathcal{L}(n)$ | the length in octets of an integer $n$. *(See Clause 5.2.5.)* |

# 5 Mathematical conventions

This clause describes certain mathematical conventions used in this part of ISO/IEC 18033, including the representation of mathematical objects, and primitives for data type conversion.

## 5.1 Functions and algorithms

For ease of presentation, functions and probabilistic functions (i.e., functions whose value depends not only on the input value but also on a randomly chosen auxiliary value) are often specified in algorithmic form. Except where explicitly noted, an implementor may choose to employ any equivalent algorithm (i.e., one which yields the same function or probabilistic function). Moreover, in the case of probabilistic functions, when the algorithm describing the function indicates that a random value should be generated, an implementor shall use an appropriate random generator to generate this value (see ISO/IEC 18031 for more guidance on this issue).

In describing a function in algorithmic terms, the following convention is adopted. An algorithm either computes a value, or alternatively, it may **fail**. By convention, if an algorithm **fails**, then unless otherwise specified, another algorithm that invokes this algorithm as a sub-routine also **fails**.

NOTE Thus, **failing** is analogous to the notion of "throwing an exception" in many programming languages; however, it can also be viewed as returning a special value that is by definition distinct from all values returned by the algorithm when it does not **fail**. With this latter interpretation of **failing**, an algorithm still properly describes a function. The details of how an implementation achieves the effect of **failing** are not specified here. However, in a typical implementation, an algorithm may return an "error code" of some sort to its environment that indicates the reason for the failure. It should be noted that in some cases, for reasons of security, the implementation should take care *not* to reveal the precise cause of certain types of errors.



 

## 5.2 Bit strings and octet strings

### 5.2.1 Bits and bit strings

A *bit* is one of the two symbols '0' or '1'.

A *bit string* is a sequence of bits. For bits $x_1, \ldots, x_l$, $\langle x_1, \ldots, x_l \rangle$ denotes the bit string of length $l$ consisting of the bits $x_1, \ldots, x_l$, in the given order.

For a bit string $x = \langle x_1, \ldots, x_l \rangle$, the length $l$ of $x$ is denoted by $|x|$, and if $l > 0$, $x_1$ is called the *first* bit of $x$, and $x_l$ the *last* bit of $x$.

For bit strings $x$ and $y$, $x \,\|\, y$ denotes the concatenation of $x$ and $y$; that is, if $x = \langle x_1, \ldots, x_l \rangle$ and $y = \langle y_1, \ldots, y_m \rangle$, then $x \,\|\, y = \langle x_1, \ldots, x_l, y_1, \ldots, y_m \rangle$.

For bit strings $x$ and $y$ of equal length, $x \oplus y$ denotes the bit-wise exclusive-or (XOR) of $x$ and $y$.

The bit string of length zero is called the *null* bit string.

NOTE No special subscripting operator is defined for bit strings. Thus, if $x$ is a bit string, $x_i$ does not necessarily denote any particular bit of $x$.

### 5.2.2 Octets and octet strings

An *octet* is a bit string of length 8.

An *octet string* is a sequence of octets.

For octets $x_1, \ldots, x_l$, $\langle x_1, \ldots, x_l \rangle$ denotes the octet string of length $l$ consisting of the octets $x_1, \ldots, x_l$, in the given order.

For an octet string $x = \langle x_1, \ldots, x_l \rangle$, the length $l$ of $x$ is denoted by $|x|$, and if $l > 0$, $x_1$ is called the *first* octet of $x$, and $x_l$ the *last* octet of $x$.

For octet strings $x$ and $y$, $x \,\|\, y$ denotes the concatenation of $x$ and $y$; that is, if $x = \langle x_1, \ldots, x_l \rangle$ and $y = \langle y_1, \ldots, y_m \rangle$, then $x \,\|\, y = \langle x_1, \ldots, x_l, y_1, \ldots, y_m \rangle$.

For octet strings $x$ and $y$ of equal length, $x \oplus y$ denotes the bit-wise exclusive-or (XOR) of $x$ and $y$.

The octet string of length zero is called the *null* octet string.

NOTE 1 No special subscripting operator is defined for octet strings. Thus, if $x$ is an octet string, $x_i$ does not necessarily denote any particular octet of $x$.

NOTE 2 Note that since an octet is a bit string of length 8, if $x$ and $y$ are octets, then $x \,\|\, y$ is a *bit* string of length 16, $\langle x \rangle$ and $\langle y \rangle$ are each *octet* strings of length 1, and $\langle x \rangle \,\|\, \langle y \rangle = \langle x, y \rangle$ is an *octet* string of length 2.



 

### 5.2.3 Octet string/bit string conversion

Primitives *OS2BSP* and *BS2OSP* to convert between octet strings and bit strings are defined as follows.

The function $OS2BSP(x)$ takes as input an octet string $x = \langle x_1, \ldots, x_l \rangle$, and outputs the bit string $y = x_1 \| \cdots \| x_l$.

The function $BS2OSP(y)$ takes as input a bit string $y$, whose length is a multiple of 8, and outputs the unique octet string $x$ such that $y = OS2BSP(x)$.

### 5.2.4 Bit string/integer conversion

Primitives *BS2IP* and *I2BSP* to convert between bit strings and integers are defined as follows.

The function $BS2IP(x)$ maps a bit string $x$ to an integer value $x'$, as follows. If $x = \langle x_{l-1}, \ldots, x_0 \rangle$ where $x_0, \ldots, x_{l-1}$ are bits, then the value $x'$ is defined as

$$x' = \sum_{\substack{0 \le i < l \\ x_i = \text{`1'}}} 2^i.$$

The function $I2BSP(m, l)$ takes as input two non-negative integers $m$ and $l$, and outputs the unique bit string $x$ of length $l$ such that $BS2IP(x) = m$, if such an $x$ exists. Otherwise, the function **fails**.

The *length in bits of a non-negative integer* $n$ is the number of bits in its binary representation, i.e., $\lceil \log_2(n+1) \rceil$.

As a notational convenience, $Oct(m)$ is defined as $Oct(m) = I2BSP(m, 8)$.

NOTE Note that $I2BSP(m, l)$ **fails** if and only if the length of $m$ in bits is greater than $l$.

### 5.2.5 Octet string/integer conversion

Primitives *OS2IP* and *I2OSP* to convert between octet strings and integers are defined as follows.

The function $OS2IP(x)$ takes as input an octet string, and outputs the integer $BS2IP(OS2BSP(x))$.

The function $I2OSP(m, l)$ takes as input two non-negative integers $m$ and $l$, and outputs the unique octet string $x$ of length $l$ such that $OS2IP(x) = m$, if such an $x$ exists. Otherwise, the function **fails**.

The *length in octets of a non-negative integer* $n$ is the number of digits in its representation base 256, i.e., $\lceil \log_{256}(n+1) \rceil$; this quantity is denoted $\mathcal{L}(n)$.

NOTE Note that $I2OSP(m, l)$ **fails** if and only if the length of $m$ in octets is greater than $l$.

## 5.3 Finite fields

This clause describes a very general framework for describing specific finite fields. A finite field specified in this way is called an *explicitly given finite field*, and it is determined by *explicit data*.



 

For a finite field $F$ of cardinality $q = p^e$, where $p$ is prime and $e \geq 1$, explicit data for $F$ consists of $p$ and $e$, along with a "multiplication table," which is a matrix $T = (T_{ij})_{1 \leq i,j \leq e}$, where each $T_{ij}$ is an $e$-tuple over $[0\,..\,p)$.

The set of elements of $F$ is the set of all $e$-tuples over $[0\,..\,p)$. The entries of $T$ are themselves viewed as elements of $F$.

Addition in $F$ is defined element-wise: if

$$a = (a_1, \ldots, a_e) \in F \quad \text{and} \quad b = (b_1, \ldots, b_e) \in F,$$

then $a + b = c$, where

$$c = (c_1, \ldots, c_e) \quad \text{and} \quad c_i = (a_i + b_i) \bmod p \ (1 \leq i \leq e).$$

A scalar multiplication operation for $F$ is also defined element-wise: if

$$a = (a_1, \ldots, a_e) \in F \quad \text{and} \quad d \in [0\,..\,p),$$

then $d \cdot a = c$, where

$$c = (c_1, \ldots, c_e) \quad \text{and} \quad c_i = (d \cdot a_i) \bmod p \ (1 \leq i \leq e).$$

Multiplication in $F$ is defined via the multiplication table $T$, as follows: if

$$a = (a_1, \ldots, a_e) \in F \quad \text{and} \quad b = (b_1, \ldots, b_e) \in F,$$

$$a \cdot b = \sum_{i=1}^{e} \sum_{j=1}^{e} (a_i b_j \bmod p) T_{ij},$$

where the products $(a_i b_j \bmod p)T_{ij}$ are defined using the above rule for scalar multiplication, and where these products are summed using the above rule for addition in $F$. It is assumed that the multiplication table defines an algebraic structure that satisfies the usual axioms of a field; in particular, there exist additive and multiplicative identities, every element has an additive inverse, and every element besides the additive identity has a multiplicative inverse.

Observe that the additive identity of $F$, denoted $0_F$, is the all-zero $e$-tuple, and that the multiplicative identity of $F$, denoted $1_F$, is a non-zero $e$-tuple whose precise format depends on $T$.

NOTE 1 The field $F$ is a vector space of dimension $e$ over the prime field $F'$ of cardinality $p$, where scalar multiplication is defined as above. The prime $p$ is called the *characteristic* of $F$. For $1 \leq i \leq e$, let $\theta_i$ denote the $e$-tuple over $F'$ whose $i$th component is 1, and all of whose other components are 0. The elements $\theta_1, \ldots, \theta_e$ form an ordered basis of $F$ as a vector space over $F'$. Note that for $1 \leq i, j \leq e$, we have $\theta_i \cdot \theta_j = T_{ij}$.

NOTE 2 For $e > 1$, two types of *standard bases* are defined that are commonly used in implementations of finite field arithmetic:

— $\theta_1, \ldots, \theta_e$ is called a *polynomial basis* for $F$ over $F'$ if for some $\theta \in F$, $\theta_i = \theta^{e-i}$ for $1 \leq i \leq e$. Note that in this case, $1_F = \theta_e$.

— $\theta_1, \ldots, \theta_e$ is called a *normal basis* for $F$ over $F'$ if for some $\theta \in F$, $\theta_i = \theta^{p^{i-1}}$ for $1 \leq i \leq e$. Note that in this case, $1_F = c \sum_{i=1}^{e} \theta_i$ for some $c \in [1\,..\,p)$; if $p = 2$, then the only possible choice for $c$ is 1; moreover, one can always choose a normal basis for which $c = 1$.

NOTE 3 The definition given here of an explicitly given finite field comes from [23].

### 5.3.1 Octet string and integer/finite field conversion

Primitives $OS2FEP_F$ and $FE2OSP_F$ to convert between octet strings and elements of an explicitly given finite field $F$, as well as the primitive $FE2IP_F$ to convert elements of $F$ to integer values, are defined as follows.

The function $FE2IP_F$ maps an element $a \in F$ to an integer value $a'$, as follows. If the cardinality of $F$ is $q = p^e$, where $p$ is prime and $e \geq 1$, then an element $a$ of $F$ is an $e$-tuple $(a_1, \ldots, a_e)$, where $a_i \in [0\,..\,p)$ for $1 \leq i \leq e$, and the value $a'$ is defined as

$$a' = \sum_{i=1}^{e} a_i p^{i-1}.$$

The function $FE2OSP_F(a)$ takes as input an element $a$ of the field $F$ and outputs the octet string $I2OSP(a', l)$, where $a' = FE2IP_F(a)$, and $l$ is the length in octets of $|F|-1$, i.e., $l = \lceil \log_{256} |F| \rceil$. Thus, the output of $FE2OSP_F(a)$ is always an octet string of length exactly $\lceil \log_{256} |F| \rceil$.

The function $OS2FEP_F(x)$ takes as input an octet string $x$, and outputs the (unique) field element $a \in F$ such that $FE2OSP_F(a) = x$, if any such $a$ exists, and otherwise **fails**. Note that $OS2FEP_F(x)$ **fails** if and only if either $x$ does not have length exactly $\lceil \log_{256} |F| \rceil$, or $OS2IP(x) \geq |F|$.

## 5.4 Elliptic curves

An elliptic curve $E$ over an explicitly given finite field $F$ is a set of points $P = (x, y)$, where $x$ and $y$ are elements of $F$ that satisfy a certain equation, together with the "point at infinity," denoted by $\mathcal{O}$. For the purposes of this part of ISO/IEC 18033, the curve $E$ is specified by two field elements $a, b \in F$, called the *coefficients* of $E$.

Let $p$ be the characteristic of $F$.

If $p > 3$, then $a$ and $b$ shall satisfy $4a^3 + 27b^2 \neq 0_F$, and every point $P = (x, y)$ on $E$ (other than $\mathcal{O}$) shall satisfy the equation

$$y^2 = x^3 + ax + b.$$

If $p = 2$, then $b$ shall satisfy $b \neq 0_F$, and every point $P = (x, y)$ on $E$ (other than $\mathcal{O}$) shall satisfy the equation

$$y^2 + xy = x^3 + ax^2 + b.$$

If $p = 3$, then $a$ and $b$ shall satisfy $a \neq 0_F$ and $b \neq 0_F$, and every point $P = (x, y)$ on $E$ (other than $\mathcal{O}$) shall satisfy the equation

$$y^2 = x^3 + ax^2 + b.$$

The points on an elliptic curve form a finite abelian group, where $\mathcal{O}$ is the identity element. There exist efficient algorithms to perform the group operation of an elliptic curve, but the implementation of such algorithms is out of the scope of this part of ISO/IEC 18033.

NOTE See, for example, ISO/IEC 15946-1, as well as [9], for more information on how to efficiently implement elliptic curve group operations.



 

### 5.4.1 Compressed elliptic curve points

Let $E$ be an elliptic curve over an explicitly given finite field $F$, where $F$ has characteristic $p$.

A point $P \neq \mathcal{O}$ can be represented in either *compressed*, *uncompressed*, or *hybrid* form.

If $P = (x, y)$, then $(x, y)$ is the uncompressed form of $P$.

Let $P = (x, y)$ be a point on the curve $E$, as above. The *compressed form* of $P$ is the pair $(x, \tilde{y})$, where $\tilde{y} \in \{0, 1\}$ is determined as follows.

— If $p \neq 2$ and $y = 0_F$, then $\tilde{y} = 0$.

— If $p \neq 2$ and $y \neq 0_F$, then $\tilde{y} = ((y'/p^f) \bmod p) \bmod 2$, where $y' = FE2IP_F(y)$, and where $f$ is the largest non-negative integer such that $p^f \mid y'$.

— If $p = 2$ and $x = 0_F$, then $\tilde{y} = 0$.

— If $p = 2$ and $x \neq 0_F$, then $\tilde{y} = \lfloor z'/2^f \rfloor \bmod 2$, where $z = y/x$, where $z' = FE2IP_F(z)$, and where $f$ is the largest non-negative integer such that $2^f$ divides $FE2IP_F(1_F)$.

The *hybrid form* of $P = (x, y)$ is the triple $(x, \tilde{y}, y)$, where $\tilde{y}$ is as in the previous paragraph.

### 5.4.2 Point decompression algorithms

There exist efficient procedures for *point decompression*, i.e., computing $y$ from $(x, \tilde{y})$. These are briefly described here.

— Assume $p \neq 2$, and let $(x, \tilde{y})$ be the compressed form of $(x, y)$. The point $(x, y)$ satisfies an equation $y^2 = f(x)$ for a polynomial $f(x)$ over $F$ in $x$. If $f(x) = 0_F$, then there is only one possible choice for $y$, namely $y = 0_F$. Otherwise, if $f(x) \neq 0$, then there are two possible choices of $y$, which differ only in sign, and the correct choice is determined by $\tilde{y}$. There are well-known algorithms for computing square roots in finite fields, and so the two choices of $y$ are easily computed.

— Assume $p = 2$, and let $(x, \tilde{y})$ be the compressed form of $(x, y)$. The point $(x, y)$ satisfies an equation $y^2 + xy = x^3 + ax^2 + b$. If $x = 0_F$, then we have $y^2 = b$, from which $y$ is uniquely determined and easily computed. Otherwise, if $x \neq 0_F$, then setting $z = y/x$, we have $z^2 + z = g(x)$, where $g(x) = (x + a + bx^{-2})$. The value of $y$ is uniquely determined by, and easily computed from, the values $z$ and $x$, and so it suffices to compute $z$. To compute $z$, observe that for a fixed $x$, if $z$ is one solution to the equation $z^2 + z = g(x)$, then there is exactly one other solution, namely $z + 1_F$. It is easy to compute these two candidate values of $z$, and the correct choice of $z$ is easily seen to be determined by $\tilde{y}$.

### 5.4.3 Octet string/elliptic curve conversion

Primitives $EC2OSP_E$ and $OS2ECP_E$ for converting between points on an elliptic curve $E$ and octet strings are defined as follows.

Let $E$ be an elliptic curve over an explicitly given finite field $F$.



 

The function $EC2OSP_E(P, fmt)$ takes as input a point $P$ on $E$ and a format specifier $fmt$, which is one of the symbolic values compressed, uncompressed, or hybrid. The output is an octet string $EP$, computed as follows.

— If $P = \mathcal{O}$, then $EP = \langle Oct(0) \rangle$.

— If $P = (x, y) \neq \mathcal{O}$, with compressed form $(x, \tilde{y})$, then

$$EP = \langle H \rangle \,\|\, X \,\|\, Y,$$

where

— $H$ is a single octet of the form $Oct(4U + C \cdot (2 + \tilde{y}))$, where

— $U = 1$ if $fmt$ is either uncompressed or hybrid, and otherwise, $U = 0$;

— $C = 1$ if $fmt$ is either compressed or hybrid, and otherwise, $C = 0$;

— $X$ is the octet string $FE2OSP_F(x)$;

— $Y$ is the octet string $FE2OSP_F(y)$ if $fmt$ is either uncompressed or hybrid, and otherwise $Y$ is the null octet string.

NOTE If the format specifier $fmt$ is uncompressed, then the value $\tilde{y}$ need not be computed.

The function $OS2ECP_E(EP)$ takes as input an octet string $EP$. If there exists a point $P$ on the curve $E$ and a format specifier $fmt$ such that $EC2OSP_E(P, fmt) = EP$, then the function outputs $P$ (in uncompressed form), and otherwise the function **fails**. Note that the point $P$, if it exists, is uniquely defined, and so the function $OS2ECP_E(EP)$ is well defined.



# 6 Cryptographic transformations

This clause describes several cryptographic transformations that will be referred to in subsequent clauses. The types of transformations are *cryptographic hash functions*, *key derivation functions*, *message authentication codes*, *block ciphers*, and *symmetric ciphers*. For each type of transformation, the abstract input/output characteristics are given, and then specific implementations of these transformations that are allowed for use in this part of ISO/IEC 18033 are specified.

## 6.1 Cryptographic hash functions

A cryptographic hash function is essentially a function that maps an octet string of variable length to an octet string of fixed length. More precisely, a cryptographic hash function *Hash* specifies

— a positive integer *Hash.len* that denotes the length of the hash function output,

— a positive integer *Hash.MaxInputLen* that denotes the maximum length hash input,

— and a function *Hash.eval* that denotes the hash function itself, which maps octet strings of length at most *Hash.MaxInputLen* to octet strings of length *Hash.len*.

The invocation of *Hash.eval* **fails** if and only if the input length exceeds *Hash.MaxInputLen*.

 

### 6.1.1 Allowable cryptographic hash functions

For the purposes of this part of ISO/IEC 18033, the allowable cryptographic hash functions are those described in ISO/IEC 10118-2 and ISO/IEC 10118-3, with the following provisos:

— The hash functions described in ISO/IEC 10118 map bit strings to bit strings, whereas in this part of ISO/IEC 18033, they map octet strings to octet strings. Therefore, a hash function in ISO/IEC 10118-2 or ISO/IEC 10118-3 is allowed in this part of ISO/IEC 18033 only if the length in bits of the output is a multiple of 8, in which case the mapping between octet strings and bit strings is affected by the functions *OS2BSP* and *BS2OSP*.

— Whereas the hash functions in ISO/IEC 10118 are not defined for inputs exceeding a given length, a hash function in this part of ISO/IEC 18033 is defined to **fail** for such inputs.

## 6.2 Key derivation functions

A *key derivation function* is a function $KDF(x, l)$ that takes as input an octet string $x$ and an integer $l \geq 0$, and outputs an octet string of length $l$. The string $x$ is of arbitrary length, although an implementation may define a (very large) maximum length for $x$ and maximum size for $l$, and **fail** if these bounds are exceeded.

NOTE In some other documents and standards, the term "mask generation function" is used instead of "key derivation function."

### 6.2.1 Allowable key derivation functions

The key derivation functions that are allowed in this part of ISO/IEC 18033 are *KDF1*, described below in Clause 6.2.2, and *KDF2*, described below in Clause 6.2.3.

### 6.2.2 *KDF1*

#### 6.2.2.1 System parameters

*KDF1* is a family of key derivation functions, parameterized by the following system parameters:

— *Hash*: a cryptographic hash function, as described in Clause 6.1.

#### 6.2.2.2 Specification

For an octet string $x$ and a non-negative integer $l$, $KDF1(x, l)$ is defined to be the first $l$ octets of

$$Hash.eval(x \,\|\, I2OSP(0, 4)) \,\|\, \cdots \,\|\, Hash.eval(x \,\|\, I2OSP(k - 1, 4)),$$

where $k = \lceil l / Hash.len \rceil$.

NOTE This function will **fail** if and only if $k > 2^{32}$ or if $|x| + 4 > Hash.MaxInputLen$.

### 6.2.3 *KDF2*

#### 6.2.3.1 System parameters

*KDF2* is a family of key derivation functions, parameterized by the following system parameters:

— *Hash*: a cryptographic hash function, as described in Clause 6.1.



 

#### 6.2.3.2 Specification

For an octet string $x$ and a non-negative integer $l$, $KDF2(x, l)$ is defined to be the first $l$ octets of

$$Hash.eval(x \,\|\, I2OSP(1, 4)) \,\|\, \cdots \,\|\, Hash.eval(x \,\|\, I2OSP(k, 4)),$$

where $k = \lceil l/Hash.len \rceil$.

NOTE 1 This function will **fail** if and only if $k \geq 2^{32}$ or if $|x| + 4 > Hash.MaxInputLen$.

NOTE 2 *KDF2* is the same as *KDF1*, except that the counter runs from 1 to $k$, rather than from 0 to $k - 1$.

## 6.3 MAC algorithms

A MAC algorithm *MA* is a scheme that defines two positive integers *MA.KeyLen* and *MA.MACLen*, along with a function $MA.eval(k', T)$ that takes a secret key $k'$, which is an octet string of length *MA.KeyLen*, along with an arbitrary octet string $T$ as input, and computes as output an octet string *MAC* of length *MA.MACLen*.

An implementation may impose a maximum value for the length of $T$, and $MA.eval(k', T)$ will **fail** if this bound is exceeded.

NOTE See Annex B.1 for a discussion on the desired security properties of MAC algorithms.

### 6.3.1 Allowable MAC algorithms

For the purposes of this part of ISO/IEC 18033, the allowable MAC algorithms are those described in ISO/IEC 9797-1 and ISO/IEC 9797-2, with the following provisos:

— For MAC the algorithms described in ISO/IEC 9797-1 and ISO/IEC 9797-2, the inputs are bit strings, and the secret key and outputs are fixed-length bit strings. Therefore, an algorithm in ISO/IEC 9797-1 or ISO/IEC 9797-2 is allowed in this part of ISO/IEC 18033 only if the lengths in bits of the MAC and of the secret key are multiples of 8, in which case the mapping between octet strings and bit strings is affected by the functions *OS2BSP* and *BS2OSP*.

— Whereas the algorithms in ISO/IEC 9797-1 and ISO/IEC 9797-2 are not defined for inputs exceeding a given length, a MAC algorithm in this part of ISO/IEC 18033 is defined to **fail** for such inputs.

## 6.4 Block ciphers

A block cipher *BC* specifies the following:

— a positive integer *BC.KeyLen*, which is the length in octets of the secret key,

— a positive integer *BC.BlockLen*, which is the length in octets of a block of plaintext or ciphertext,

— a function $BC.Encrypt(k, b)$, which takes as input a secret key $k$, which is an octet string of length *BC.KeyLen*, and a plaintext block $b$, which is an octet string of length *BC.BlockLen*, and outputs a ciphertext block $b'$, which is an octet string of length *BC.BlockLen*, and



 

— a function $BC.Decrypt(k, b')$, which takes as input a secret key $k$, which is an octet string of length $BC.KeyLen$, and a ciphertext block $b'$, which is an octet string of length $BC.BlockLen$, and outputs a plaintext block $b$, which is an octet string of length $BC.BlockLen$.

For any fixed secret key $k$, the function $b \mapsto BC.Encrypt(k, b)$ acts as a permutation on the set of octet strings of length $BC.BlockLen$, and the function $b' \mapsto BC.Decrypt(k, b)$ acts as the inverse permutation.

NOTE See Annex B.2 for a discussion of the desired security properties of block ciphers.

### 6.4.1 Allowable block ciphers

For the purposes of this part of ISO/IEC 18033, the allowable block ciphers are those described in ISO/IEC 18033-3, with the following proviso:

— In ISO/IEC 18033-3, plaintext/ciphertext blocks and secret keys are fixed-length bit strings, whereas in this part of ISO/IEC 18033, they are fixed-length octet strings. Therefore, a block cipher in ISO/IEC 18033-3 is allowed in this part of ISO/IEC 18033 only if the lengths in bits of plaintext/ciphertext blocks and of the secret key are multiples of 8, in which case the mapping between octet strings and bit strings is affected by the functions *OS2BSP* and *BS2OSP*.

## 6.5 Symmetric ciphers

A symmetric cipher $SC$ specifies a key length $SC.KeyLen$, along with encryption and decryption algorithms:

— The encryption algorithm $SC.Encrypt(k, M)$ takes as input a secret key $k$, which is an octet string of length $SC.KeyLen$, and a plaintext $M$, which is an octet string of arbitrary length. It outputs a ciphertext $c$, which is an octet string.

The encryption algorithm may **fail** if the length of $M$ exceeds some large, implementation-defined limit.

— The decryption algorithm $SC.Decrypt(k, c)$ takes as input a secret key $k$, which is an octet string of length $SC.KeyLen$, and a ciphertext $c$, which is an octet string of arbitrary length. It outputs a plaintext $M$, which is an octet string.

The decryption algorithm may **fail** under some circumstances.

The encryption and decryption algorithms are deterministic. Also, for all secret keys $k$ and all plaintexts $M$, if $M$ does not exceed the length bound of the encryption algorithm, and if $c = SC.Encrypt(k, M)$, then $SC.Decrypt(k, c)$ does not **fail** and $SC.Decrypt(k, c) = M$.

NOTE See Annex B.3 for a discussion on the desired security properties for a symmetric cipher.

### 6.5.1 Allowable symmetric ciphers

The symmetric ciphers that are allowed in this part of ISO/IEC 18033 are

— *SC1*, described below in Clause 6.5.2, and

— *SC2*, described below in Clause 6.5.3.

### 6.5.2 *SC1*

This symmetric cipher is the cipher obtained by using a block cipher in a particular cipher block chaining (CBC) mode (see ISO/IEC 10116), together with a particular padding scheme to pad cleartexts so that their length is a multiple of the block size of the underlying block cipher.

#### 6.5.2.1 System parameters

*SC1* is a family of symmetric ciphers, parameterized by the following system parameters:

— *BC*: a block cipher, as described in Clause 6.4.

Strictly speaking, one must make the restriction that $BC.BlockLen < 256$; however, in practice this restriction is always met.

#### 6.5.2.2 Specification

$SC1.KeyLen = BC.KeyLen$.

The function $SC1.Encrypt(k, M)$ works as follows.

a) Set $padLen = BC.BlockLen - (|M| \bmod BC.BlockLen)$.

b) Let $P_1 = Oct(padLen)$.

c) Let $P_2$ be the octet string formed by repeating the octet $P_1$ a total of $padLen$ times (so $|P_2| = padLen$).

d) Let $M' = M \,\|\, P_2$.

e) Parse $M'$ as $M'_1 \,\|\, \cdots \,\|\, M'_l$, where for $1 \le i \le l$, $M'_i$ is an octet string of length $BC.BlockLen$.

f) Let $c_0$ be the octet string consisting of $BC.BlockLen$ copies of the octet $Oct(0)$, and for $1 \le i \le l$, let $c_i = BC.Encrypt(k, M'_i \oplus c_{i-1})$.

g) Let $c = c_1 \,\|\, \cdots \,\|\, c_l$

h) Output $c$.

The function $SC1.Decrypt(k, c)$ works as follows.

a) If $|c|$ is not a non-zero multiple of $BC.BlockLen$, then **fail**.

b) Parse $c$ as $c = c_1 \,\|\, \cdots \,\|\, c_l$, where for $1 \le i \le l$, $c_i$ is an octet string of length $BC.BlockLen$. Also, let $c_0$ be the octet string consisting of $BC.BlockLen$ copies of the octet $Oct(0)$.

c) For $1 \le i \le l$, let $M'_i = BC.Decrypt(k, c_i) \oplus c_{i-1}$.

d) Let $P_1$ be the last octet of $M'_l$, and let $padLen = BS2IP(P_1)$.



 

e) If $padLen \notin [1\,..\,BC.BlockLen]$, then **fail**.

f) Check that the last $padLen$ octets of $M'_l$ are equal to $P_1$; if not, then **fail**.

g) Let $M''_l$ be the octet string consisting of the first $BC.BlockLen - padLen$ octets of $M'_l$.

h) Set $M = M'_1 \,\|\, \cdots \,\|\, M'_{l-1} \,\|\, M''_l$.

i) Output $M$.

### 6.5.3 *SC2*

#### 6.5.3.1 System parameters

*SC2* is a family of symmetric ciphers, parameterized by the following system parameters:

— *KDF*: a key derivation function, as described in Clause 6.2;

— *KeyLen*: a positive integer.

#### 6.5.3.2 Specification

The value of *SC2.KeyLen* is equal to the value of the system parameter *KeyLen*.

The function $SC2.Encrypt(k, M)$ works as follows.

a) Set $mask = KDF(k, |M|)$.

b) Set $c = mask \oplus M$.

c) Output $c$.

The function $SC2.Decrypt(k, c)$ works as follows.

a) Set $mask = KDF(k, |c|)$.

b) Set $M = mask \oplus c$.

c) Output $M$.



## 7 Asymmetric ciphers

An asymmetric cipher $AC$ consists of three algorithms:

— A key generation algorithm $AC.KeyGen()$, that outputs a public-key/private-key pair $(PK, pk)$. The structure of $PK$ and $pk$ depends on the particular cipher.

— An encryption algorithm $AC.Encrypt(PK, L, M, opt)$ that takes as input a public key $PK$, a label $L$, a plaintext $M$, and an encryption option $opt$, and outputs a ciphertext $C$. Note that $L$, $M$, and $C$ are octet strings. See Clause 7.2 below for more on *labels*. See Clause 7.4 below for more on *encryption options*.

 

The encryption algorithm may **fail** if the lengths $L$ or $M$ exceed some implementation-defined limits.

— A decryption algorithm $AC.Decrypt(pk, L, C)$ that takes as input a private key $pk$, a label $L$, and a ciphertext $C$, and outputs a plaintext $M$.

The decryption algorithm may **fail** under some circumstances.

In general, the key generation and encryption algorithms will be probabilistic algorithms, while the decryption algorithm is deterministic.

NOTE 1 The intent is that all of the asymmetric ciphers described in this part of ISO/IEC 18033 provide reasonable security against adaptive chosen ciphertext attack (as defined in [30], and which is equivalent to a notion of "non-malleability" defined in [17]). This notion of security is generally regarded by the cryptographic research community as the appropriate form of security that a general-purpose asymmetric cipher should provide. The formal definition of this notion of security is presented in Annex B.4, appropriately adapted to take into account variable length plaintexts and the role of *labels*; also, a slightly weaker notion of security, called "benign malleability," is defined. This notion of "benign malleability" is also adequate for most, if not all, applications of asymmetric ciphers, and some of the asymmetric ciphers described in this part of ISO/IEC 18033 only achieve this level of security.

NOTE 2 A basic requirement of any asymmetric cipher is *correctness*: for any public-key/private-key pair $(PK, pk)$, for any label/plaintext pair $(L, M)$, such that the lengths of $L$ and $M$ do not exceed the implementation-defined limits, any encryption of $M$ with label $L$ under $PK$ decrypts with label $L$ under $pk$ to the original plaintext $M$. This requirement may be relaxed, so that it holds only for all but a negligible fraction of public-key/private-key pairs.

NOTE 3 As an example of an asymmetric cipher $AC$ for which the above correctness requirement may not always hold, consider any RSA-based cipher where the modulus $n = pq$, where $p$ and $q$ should be prime. The key generation algorithm may use a probabilistic algorithm for testing if $p$ and $q$ are prime, and this algorithm may produce incorrect results with a negligible probability; if this happens, the decryption algorithm may not be the inverse of the encryption algorithm.

## 7.1 Plaintext length

It is important to note that plaintexts may be of arbitrary and variable length, although an implementation may impose a (typically, very large) upper bound on this length.

However, two degenerate types of asymmetric ciphers are defined as follows:

— A *fixed-plaintext-length* asymmetric cipher $AC$ only encrypts plaintexts whose length (in octets) is equal to a fixed value $AC.MsgLen$.

— A *bounded-plaintext-length* asymmetric cipher $AC$ only encrypts plaintexts whose length (in octets) is less than or equal to a fixed value $AC.MaxMsgLen(PK)$. Here, the maximum plaintext length may depend on the public key $PK$ of the cipher.

NOTE Except for fixed-plaintext-length and bounded-plaintext-length asymmetric ciphers, the encryption of a plaintext will in general not hide the length of the plaintext. Therefore, it is up to the application using the asymmetric cipher to ensure, perhaps by an appropriate padding scheme, that no sensitive information is implicitly encoded in the length of a plaintext.



 

## 7.2 The use of labels

A *label* is an octet string whose value is used by the encryption and decryption algorithms. It may contain public data that is implicit from context and need not be encrypted, but that should nevertheless be bound to the ciphertext.

A label is an octet string that is meaningful to the application using the asymmetric cipher, and that is independent of the implementation of the asymmetric cipher.

Labels may be of arbitrary and variable length, although a particular cipher may choose to impose a (very large) upper bound on this length.

A degenerate type of asymmetric cipher is defined as follows:

— A *fixed-label-length* asymmetric cipher is one in which the encryption and decryption algorithms only accept labels whose length (in octets) is equal to a fixed value $AC.LabelLen$.

NOTE 1 The traditional notion of security against adaptive chosen ciphertext attack has been extended in Annex B.4, so that intuitively, for a secure asymmetric cipher, the encryption algorithm should bind the label to the ciphertext in an appropriate "non-malleable" fashion.

NOTE 2 For example, there are key exchange protocols in which one party, say $A$, encrypts a session key $K$ under the public key of the other party, say $B$. In order for the protocol to be secure, party $A$'s identity (or public key or certificate) must be non-malleably bound to the ciphertext. One way to do this is simply to append this identity to the plaintext. However, this creates an unnecessarily large ciphertext, since $A$'s identity is typically already known to $B$ in the context of such a protocol. A good implementation of the labeling mechanism achieves the same effect, without increasing the size of the ciphertext.

## 7.3 Ciphertext format

The asymmetric ciphers proposed in this part of ISO/IEC 18033 describe precisely how a ciphertext is to be formatted as an octet string. However, an implementation is free to store and/or transmit ciphertexts in alternative formats, if this is convenient. Moreover, the process of encrypting a plaintext and converting the resulting ciphertext into an alternative format may be collapsed into a single, functionally equivalent process; likewise, the process of converting from an alternative format and decrypting the ciphertext may be collapsed into a single, functionally equivalent process. Thus, in a given system, ciphertexts need never appear in the format prescribed here.

NOTE Besides promoting inter-operability, prescribing the format of a ciphertext is necessary in order to make meaningful claims and to reason about the security of an asymmetric cipher against adaptive chosen ciphertext attacks.

## 7.4 Encryption options

Some asymmetric ciphers allow certain types of scheme-specific options to be passed to the encryption algorithm, which is why an extra encryption option argument *opt* is allowed in the abstract interface for an asymmetric cipher.

Some asymmetric ciphers presented here may naturally be viewed as not having any encryption options, in which case, the cipher is said to take no encryption option.



 

A system may provide a "default" value of *opt*; however, such provisions are outside the scope of this part of ISO/IEC 18033.

NOTE Among the specific asymmetric ciphers described in this part of ISO/IEC 18033, only the elliptic-curve-based ciphers use an encryption option, which is used to indicate the desired format for encoding points on elliptic curves.

### 7.5 Method of operation of an asymmetric cipher

Typically, the key generation algorithm is run by some party, known as the *owner* of the key pair, or by some trusted party on the owner's behalf. The public key shall be made available to all parties who wish to send encrypted messages to the owner, while the private key shall not be divulged to any party other than the owner. Mechanisms and protocols for making a public key available to other parties are out of the scope of this part of ISO/IEC 18033. See ISO/IEC 11770 for guidance on this issue.

Each of the asymmetric ciphers presented in this part of ISO/IEC 18033 are actually members of *families* of asymmetric ciphers, where a particular cipher is selected from the family by choosing particular values for the *system parameters* defining the family of ciphers.

For a cipher selected from a family of ciphers, prior to key generation, specific values of the system parameters for the family shall be chosen. Depending on the conventions used for encoding public keys, some of the choices of the system parameters may be embedded in the encoding of the public key as well. These system parameters shall remain fixed throughout the lifetime of the public key.

NOTE For example, if an asymmetric cipher may be parameterized in terms of a cryptographic hash function, the choice of hash function should be fixed once and for all at some point prior to the generation of a public-key/private-key pair, and the encryption and decryption algorithms should use the chosen hash function throughout the lifetime of the public key. Failure to abide by this rule not only makes an implementation non-conforming, but also invalidates the security analysis for the cipher, and may in some cases expose the implementation to severe security risks.

### 7.6 Allowable asymmetric ciphers

Users who wish to employ an asymmetric cipher from this part of ISO/IEC 18033 shall select one of the following:

— a generic hybrid cipher chosen from the family *HC* of hybrid ciphers described in Clause 8.3;

— a bounded-plaintext-length asymmetric cipher from the family *RSAES* of ciphers described in Clause 11.4;

— a bounded-plaintext-length asymmetric cipher from the family *HIME(R)* of ciphers described in Clause 12.3.

NOTE As each of *HC*, *RSAES*, and *HIME(R)* are families of ciphers, parameterized by various system parameters, a user will have to choose specific values of these system parameters from the set of allowable system parameters specified in the corresponding clause in which each family is described.

## 8 Generic hybrid ciphers

In designing an efficient asymmetric cipher, a useful approach is to design a *hybrid cipher*, where one uses asymmetric cryptographic techniques to encrypt a secret key that can then be used to



 

encrypt the actual message using symmetric cryptographic techniques. This clause describes a specific type of hybrid cipher, called a *generic hybrid cipher*. A generic hybrid cipher is built from two lower-level "building blocks": a *key encapsulation mechanism* and a *data encapsulation mechanism*. Clause 8.3 specifies in detail the family *HC* of generic hybrid ciphers.

## 8.1 Key encapsulation mechanisms

A key encapsulation mechanism *KEM* consists of three algorithms:

— A key generation algorithm $KEM.KeyGen()$, that outputs a public-key/private-key pair $(PK, pk)$. The structure of $PK$ and $pk$ depends on the particular scheme.

— An encryption algorithm $KEM.Encrypt(PK, opt)$ that takes as input a public key $PK$ along with an encryption option $opt$, and outputs a secret-key/ciphertext pair $(K, C_0)$. Both $K$ and $C_0$ are octet strings. The role of $opt$ is analogous to its role in asymmetric ciphers (see Clause 7.4).

— A decryption algorithm $KEM.Decrypt(pk, C_0)$ that takes as input a private key $pk$ and a ciphertext $C_0$, and outputs a secret key $K$. Both $K$ and $C_0$ are octet strings.

  The decryption algorithm may **fail** under some circumstances.

A key encapsulation mechanism also specifies a positive integer *KEM.KeyLen* — the length of the secret key output by *KEM.Encrypt* and *KEM.Decrypt*.

NOTE Any key encapsulation mechanism should satisfy a correctness property analogous to the correctness property of an asymmetric cipher: for any public-key/private-key pair $(PK, pk)$, for any output $(K, C_0)$ of the encryption algorithm on input $(PK, opt)$, the ciphertext $C_0$ should decrypt under $pk$ to $K$. This requirement may be relaxed, so that it holds only for all but a negligible fraction of public-key/private-key pairs.

### 8.1.1 Prefix-freeness property

Additionally, a key encapsulation mechanism must satisfy the following property. The set of all possible ciphertext outputs of the encryption algorithm should be a subset of a *candidate* set of octet strings (that may depend on the public key), such that the candidate set is prefix free and elements of the candidate set are easy to recognize (given either the public key or the private key).

### 8.1.2 Allowable key encapsulation mechanisms

The key encapsulation mechanisms that are allowed in this part of ISO/IEC 18033 are

— *ECIES-KEM* (described in Clause 10.2),

— *PSEC-KEM* (described in Clause 10.3),

— *ACE-KEM* (described in Clause 10.4), and

— *RSA-KEM* (described in Clause 11.5).

NOTE 1 As a matter of convention, the corresponding generic hybrid ciphers built from these key encapsulation mechanisms via the generic hybrid construction in Clause 8.3 shall be called (respectively) *ECIES-HC*, *PSEC-HC*, *ACE-HC*, and *RSA-HC*.



 

NOTE 2 Roughly speaking, a key encapsulation mechanism works just like an asymmetric cipher, except that the encryption algorithm takes no input other than the recipient's public key: instead of taking a message as input and producing a ciphertext, the encryption algorithm generates a secret-key/ciphertext pair $(K, C_0)$, where $K$ is an octet string of some specified length, and $C_0$ is an encryption of $K$, that is, the decryption algorithm applied to $C_0$ yields $K$.

NOTE 3 One can always use a (possibly fixed-plaintext-length or bounded-plaintext-length) asymmetric cipher for this purpose, generating a random octet string $K$, and then encrypting it under the recipient's public key (and any encryption options) to obtain $C_0$. However, one can construct a key encapsulation mechanism in other, more efficient, ways as well.

NOTE 4 For the purposes of building a generic hybrid cipher that is secure against adaptive chosen ciphertext attack, there is a corresponding notion of security for a key encapsulation mechanism. This is discussed in detail in Annex B.5.

## 8.2 Data encapsulation mechanisms

A data encapsulation mechanism *DEM* specifies a key length *DEM.KeyLen*, along with encryption and decryption algorithms:

— The encryption algorithm $DEM.Encrypt(K, L, M)$ takes as input a secret key $K$, a label $L$, and a plaintext $M$. It outputs a ciphertext $C_1$. Here, $K$, $L$, $M$, and $C_1$ are octet strings, and $L$ and $M$ may have arbitrary length, and $K$ is of length *DEM.KeyLen*.

  The encryption algorithm may **fail** if the lengths $L$ or $M$ exceed some (very large) implementation-defined limits.

— The decryption algorithm $DEM.Decrypt(K, L, C_1)$ takes as input a secret key $K$, a label $L$, and a ciphertext $C_1$. It outputs a plaintext $M$.

  The decryption algorithm may **fail** under some circumstances.

NOTE The encryption and decryption algorithms should be deterministic, and should satisfy the following correctness requirement: for all secret keys $K$, all labels $L$, and all plaintexts $M$, such that the lengths of $L$ and $M$ do not exceed the implementation-defined limits,

$$DEM.Decrypt(K, L, DEM.Encrypt(K, L, M)) = M.$$



### 8.2.1 Degenerate types of data encapsulation mechanisms

Two different, degenerate types of data encapsulation mechanisms are defined as follows:

— A *fixed-label-length* data encapsulation mechanism is one for which the encryption and decryption algorithms only accept labels whose lengths are equal to a fixed value *DEM.LabelLen*.

— A *fixed-plaintext-length* data encapsulation mechanism is one for which the encryption algorithm only accepts plaintexts whose lengths are equal to a fixed value *DEM.MsgLen*.

### 8.2.2 Allowable data encapsulation mechanisms

The data encapsulation mechanisms that are allowed in this part of ISO/IEC 18033 are described in Clause 9.

 

NOTE 1 Roughly speaking, a data encapsulation mechanism provides a "digital envelope" that protects both the confidentiality and integrity of data using symmetric cryptographic techniques; it may also bind the data to a public label.

NOTE 2 For the purposes of building a generic hybrid cipher that is secure against adaptive chosen ciphertext attack, there is a corresponding notion of security for a data encapsulation mechanism. This is discussed in detail in Annex B.6.

## 8.3 *HC*

### 8.3.1 System parameters

*HC* is a family of asymmetric ciphers parameterized by the following system parameters:

— *KEM*: a key encapsulation mechanism, as described in Clause 8.1;

— *DEM*: a data encapsulation mechanism, as described in Clause 8.2.

Any combination of *KEM* and *DEM* may be used, provided $KEM.KeyLen = DEM.KeyLen$.

NOTE 1 If *DEM* is a fixed-label-length data encapsulation mechanism with labels restricted to length *DEM.LabelLen*, then *HC* is a fixed-label-length asymmetric cipher with $HC.LabelLen = DEM.LabelLen$.

NOTE 2 If *DEM* is a fixed-plaintext-length data encapsulation mechanism, with plaintexts restricted to length *DEM.MsgLen*, then *HC* is a fixed-plaintext-length asymmetric cipher with $HC.MsgLen = DEM.MsgLen$.

NOTE 3 For all the allowable choices of *KEM*, the value of *KEM.KeyLen* is a system parameter that may be chosen so as to equal *DEM.KeyLen*. Thus, all possible combinations of allowable *KEM* and *DEM* may be realized by appropriate choices of system parameters.

### 8.3.2 Key generation

The key generation algorithm, public key, and private key for *HC* are the same as that of *KEM*. The encryption options of *HC* are the same as that of *KEM*.

Let $(PK, pk)$ denote a public-key/private-key pair.

### 8.3.3 Encryption

The encryption algorithm *HC.Encrypt* takes as input a public key $PK$, a label $L$, a plaintext $M$, and an encryption option $opt$. It runs as follows.

a) Compute $(K, C_0) = KEM.Encrypt(PK, opt)$.

b) Compute $C_1 = DEM.Encrypt(K, L, M)$.

c) Set $C = C_0 \,||\, C_1$.

d) Output $C$.



 

### 8.3.4 Decryption

The decryption algorithm *HC.Decrypt* takes as input a private key $pk$, a label $L$, and a ciphertext $C$. It runs as follows.

a) Using the prefix-freeness property of the ciphertexts associated with *KEM* (see Clause 8.1.1), parse $C$ as $C = C_0 \,\|\, C_1$, where $C_0$ and $C_1$ are octet strings such that $C_0$ is an element of the candidate set of possible ciphertexts associated with *KEM*. This step **fails** if $C$ cannot be so parsed.

b) Compute $K = KEM.Decrypt(pk, C_0)$.

c) Compute $M = DEM.Decrypt(K, L, C_1)$

d) Output $M$.

NOTE The security of *HC* is discussed in Annex B.7. It is only remarked here that so long as *KEM* and *DEM* satisfy the appropriate security properties, then *HC* will be secure against adaptive chosen ciphertext attack.

## 9 Constructions of data encapsulation mechanisms

This clause specifies the data encapsulation mechanisms that are allowed in this part of ISO/IEC 18033. These mechanisms are

— *DEM1*, described below in Clause 9.1,

— *DEM2*, described below in Clause 9.2, and

— *DEM3*, described below in Clause 9.3.

### 9.1 *DEM1*

#### 9.1.1 System parameters

*DEM1* is a family of data encapsulation mechanisms, parameterized by the following system parameters:

— *SC*: a symmetric cipher, as described in Clause 6.5;

— *MA*: a MAC algorithm, as described in Clause 6.3.

The value of *DEM1.KeyLen* is defined as $DEM1.KeyLen = SC.KeyLen + MA.KeyLen$.

#### 9.1.2 Encryption

The algorithm *DEM1.Encrypt* takes as input a secret key $K$, a label $L$, and a plaintext $M$. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = SC.KeyLen$ and $|k'| = MA.KeyLen$.



 

b) Compute $c = SC.Encrypt(k, M)$.

c) Let $T = c \,\|\, L \,\|\, I2OSP(8 \cdot |L|, 8)$.

d) Compute $MAC = MA.eval(k', T)$.

e) Set $C_1 = c \,\|\, MAC$.

f) Output $C_1$.

### 9.1.3 Decryption

The algorithm *DEM1.Decrypt* takes as input a secret key $K$, a label $L$, and a ciphertext $C_1$. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = SC.KeyLen$ and $|k'| = MA.KeyLen$.

b) If $|C_1| < MA.MACLen$, then **fail**.

c) Parse $C_1$ as $C_1 = c \,\|\, MAC$, where $c$ and $MAC$ are octet strings such that $|MAC| = MA.MACLen$.

d) Let $T = c \,\|\, L \,\|\, I2OSP(8 \cdot |L|, 8)$.

e) Compute $MAC' = MA.eval(k', T)$.

f) If $MAC \neq MAC'$, then **fail**.

g) Compute $M = SC.Decrypt(k, c)$.

h) Output $M$.

NOTE A detailed discussion of the security of this construction is found in Annex B.6.1. It is only remarked here that provided the underlying *SC* and *MA* satisfy the appropriate security requirements, then so too will *DEM1*.

## 9.2 *DEM2*

### 9.2.1 System parameters

*DEM2* is a family of fixed-label-length data encapsulation mechanisms, parameterized by the following system parameters:

— *SC*: a symmetric cipher, as described in Clause 6.5;

— *MA*: a MAC algorithm, as described in Clause 6.3;

— *LabelLen*: a non-negative integer.

The value of *DEM2.LabelLen* is defined to be equal to the value of the system parameter *LabelLen*.

The value of *DEM2.KeyLen* is defined as $DEM2.KeyLen = SC.KeyLen + MA.KeyLen$.



 

### 9.2.2 Encryption

The algorithm *DEM2.Encrypt* takes as input a secret key $K$, a label $L$ of length *LabelLen*, and a plaintext $M$. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = SC.KeyLen$ and $|k'| = MA.KeyLen$.

b) Compute $c = SC.Encrypt(k, M)$.

c) Let $T = c \,\|\, L$.

d) Compute $MAC = MA.eval(k', T)$.

e) Set $C_1 = c \,\|\, MAC$.

f) Output $C_1$.

### 9.2.3 Decryption

The algorithm *DEM2.Decrypt* takes as input a secret key $K$, a label $L$ of length *LabelLen*, and a ciphertext $C_1$. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = SC.KeyLen$ and $|k'| = MA.KeyLen$.

b) If $|C_1| < MA.MACLen$, then **fail**.

c) Parse $C_1$ as $C_1 = c \,\|\, MAC$, where $c$ and $MAC$ are octet strings such that $|MAC| = MA.MACLen$.

d) Let $T = c \,\|\, L$.

e) Compute $MAC' = MA.eval(k', T)$.

f) If $MAC \neq MAC'$, then **fail**.

g) Compute $M = SC.Decrypt(k, c)$.

h) Output $M$.

NOTE 1 A detailed discussion of the security of this construction is found in Annex B.6.1. It is only remarked here that provided the underlying *SC* and *MA* satisfy the appropriate security requirements, then so too will *DEM2*.

NOTE 2 *DEM2* is provided mainly for compatibility with other standards.

## 9.3 *DEM3*

### 9.3.1 System parameters

*DEM3* is a family of fixed-plaintext-length data encapsulation mechanisms, parameterized by the following system parameters:



 

— *MA*: a MAC algorithm, as described in Clause 6.3;

— *MsgLen*: a positive integer.

The value of *DEM3.MsgLen* is defined to be equal to the value of the system parameter *MsgLen*.

The value of *DEM3.KeyLen* is defined as $DEM3.KeyLen = MsgLen + MA.KeyLen$.

### 9.3.2 Encryption

The algorithm *DEM3.Encrypt* takes as input a secret key $K$, a label $L$, and a plaintext $M$ of length *MsgLen*. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = MsgLen$ and $|k'| = MA.KeyLen$.

b) Compute $c = k \oplus M$.

c) Let $T = c \,\|\, L$.

d) Compute $MAC = MA.eval(k', T)$.

e) Set $C_1 = c \,\|\, MAC$.

f) Output $C_1$.

### 9.3.3 Decryption

The algorithm *DEM3.Decrypt* takes as input a secret key $K$, a label $L$, and a ciphertext $C_1$. It runs as follows.

a) Parse $K$ as $K = k \,\|\, k'$, where $k$ and $k'$ are octet strings such that $|k| = MsgLen$ and $|k'| = MA.KeyLen$.

b) If $|C_1| \neq MsgLen + MA.MACLen$, then **fail**.

c) Parse $C_1$ as $C_1 = c \,\|\, MAC$, where $c$ and $MAC$ are octet strings such that $|c| = MsgLen$ and $|MAC| = MA.MACLen$.

d) Let $T = c \,\|\, L$.

e) Compute $MAC' = MA.eval(k', T)$.

f) If $MAC \neq MAC'$, then **fail**.

g) Compute $M = k \oplus c$.

h) Output $M$.

NOTE 1 A detailed discussion of the security of this construction is found in Annex B.6.1. It is only remarked here that provided the underlying *MA* satisfies the appropriate security requirement, then so too will *DEM3*.

NOTE 2 *DEM3* is provided mainly for compatibility with other standards.



 

# 10 ElGamal-based key encapsulation mechanisms

This clause describes several key encapsulation mechanisms based on the discrete logarithm problem:

— *ECIES-KEM* is described in Clause 10.2;

— *PSEC-KEM* is described in Clause 10.3;

— *ACE-KEM* is described in Clause 10.4.

NOTE All of these schemes are variations on the original ElGamal encryption scheme [18].

## 10.1 Concrete groups

ElGamal encryption is based on arithmetic in a finite group. For the purposes of describing key encapsulation mechanisms based on ElGamal encryption, a group is described as an abstract data type. The description and analysis of these schemes relies on this abstract interface; however, this part of ISO/IEC 18033 only allows an implementation to use certain types of groups when instantiating this abstract data type.

As a matter of convention, additive notation will always be used for a group. Also, group elements will be typeset in boldface, and $\mathbf{0}$ denotes the identity element of the group.

A *concrete group* $\Gamma$ is a tuple $(\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}')$, where:

— $\mathcal{H}$ is a finite abelian group in which all group computations are actually performed. Note that this group need not be cyclic.

— $\mathcal{G}$ is a *cyclic* subgroup of $\mathcal{H}$.

— $\mathbf{g}$ is a generator for $\mathcal{G}$.

— $\mu$ is the order (i.e., size) of $\mathcal{G}$, and $\nu$ is the index of $\mathcal{G}$ in $\mathcal{H}$, i.e., $\nu = |\mathcal{H}|/\mu$.

It is required that $\mu$ is prime. For some cryptographic schemes, it is further required that $\gcd(\mu, \nu) = 1$.

— $\mathcal{E}(\mathbf{a}, fmt)$ is an "encoding" function that maps a group element $\mathbf{a} \in \mathcal{H}$ to an octet string.

The second argument *fmt* is a format specifier that is used to choose from one of a small number of several possible formats for the encoding of a group element. The allowable values of *fmt* depend on the group.

The following requirements shall be met:

— The set of all outputs of $\mathcal{E}$ is prefix free.

— The identity element has a unique encoding; that is, for all format specifiers $fmt, fmt'$, we have $\mathcal{E}(\mathbf{0}, fmt) = \mathcal{E}(\mathbf{0}, fmt')$.



 

— Except on the identity element, the encoding function is one to one; that is, for all $\mathbf{a}, \mathbf{a}' \in \mathcal{H}$ and for all format specifiers $fmt, fmt'$, if $(\mathbf{a}, fmt) \neq (\mathbf{a}', fmt')$, and if either $\mathbf{a} \neq \mathbf{0}$ or $\mathbf{a}' \neq \mathbf{0}$, then $\mathcal{E}(\mathbf{a}, fmt) \neq \mathcal{E}(\mathbf{a}', fmt')$.

An octet string $x$ is called a *valid encoding* of a group element $\mathbf{a} \in \mathcal{H}$ if $x = \mathcal{E}(\mathbf{a}, fmt)$ for some format specifier $fmt$.

— $\mathcal{D}(x)$ is the function that **fails** if $x$ is not a valid encoding of an element of $\mathcal{H}$; otherwise, it returns the unique group element $\mathbf{a} \in \mathcal{H}$ such that $\mathcal{E}(\mathbf{a}, fmt) = x$ for some format specifier $fmt$.

— $\mathcal{E}'(\mathbf{a})$ is a "partial encoding" function that maps a group element $\mathbf{a} \in \mathcal{H}$ to an octet string.

It is required that the set of all outputs of $\mathcal{E}'$ is prefix free.

An octet string $x$ is called a *valid partial encoding* of a group element $\mathbf{a}$ if $x = \mathcal{E}'(\mathbf{a})$.

— $\mathcal{D}'(x)$ is a function that either **fails** if $x$ is not a valid partial encoding of an element of $\mathcal{H}$; otherwise, it returns the set containing all group elements $\mathbf{a} \in \mathcal{H}$ such that $\mathcal{E}'(\mathbf{a}) = x$. It is assumed that the size of this set is bounded by a small constant.

It is assumed that arithmetic in $\mathcal{H}$ can be carried out efficiently. Also, all of the above algorithms should have efficient implementations. The function $\mathcal{D}'$ will never be used by any of the schemes, but the existence of this function is necessary to analyze their security.

It is also assumed that one can efficiently test if an element of $\mathcal{H}$ lies in the subgroup $\mathcal{G}$. Note that if all elements in $\mathcal{H}$ of order $\mu$ lie in $\mathcal{G}$, then one can test if $\mathbf{a} \in \mathcal{G}$ by testing if $\mu \cdot \mathbf{a} = \mathbf{0}$. This test is therefore applicable if $\mathcal{H}$ is itself cyclic, or if $\gcd(\mu, \nu) = 1$. For specific groups, there may be more efficient tests of subgroup membership.

A set $\{\mathcal{E}(\mathbf{a}_1, fmt_1), \ldots, \mathcal{E}(\mathbf{a}_m, fmt_m)\}$ of valid encodings of group elements is called *consistent* if the encodings of all non-identity group elements use the same format specifier; that is, for all $1 \leq i, j \leq m$, if $\mathbf{a}_i \neq \mathbf{0}$ and $\mathbf{a}_j \neq \mathbf{0}$, then $fmt_i = fmt_j$. Given the above assumptions, one can efficiently test if a given set of valid encodings is consistent.

NOTE Different cryptographic applications will make different intractability assumptions about a group. These assumptions are discussed in Annex B.8.

### 10.1.1 Allowable concrete groups

This part of ISO/IEC 18033 allows only the following two families of concrete groups, described below in Clauses 10.1.2 and 10.1.3.

### 10.1.2 Subgroups of explicitly given finite fields

Let $F$ be an explicitly given finite field, as defined in Clause 5.3, and consider the multiplicative group $F^*$ of units in $F$. Let $\mathcal{H}$ denote $F^*$. Let $\mathcal{G}$ denote any prime-order subgroup of $F^*$, and let $\mathbf{g}$ be a generator for $\mathcal{G}$. Set $\mu = |\mathcal{G}|$ and $\nu = (|F| - 1)/\mu$.

Because $\mathcal{H}$ is itself cyclic, it follows that $\mathcal{G}$ contains all elements of $\mathcal{H}$ whose order divides $\mu$, even if $\gcd(\mu, \nu) \neq 1$. Thus, one may always test if an element $\mathbf{a} \in \mathcal{H}$ lies in $\mathcal{G}$ by testing if $\mu \cdot \mathbf{a} = \mathbf{0}$; there may, however, be other, more efficient tests; for example, if $F$ is a prime finite field, and $\nu = 2$, this test may be implemented via a Jacobi symbol computation.



 

The encoding map $\mathcal{E}$ is implemented using the function $FE2OSP_F$, so that all group elements are encoded as octet strings of length $\lceil \log_{256} |F| \rceil$. Only one format is allowed. The map $\mathcal{D}$ is implemented using $OS2FEP_F$, and **fails** if $OS2FEP_F$ **fails** or yields $0_F$. The function $\mathcal{E}'$ is the same as $\mathcal{E}$, and $\mathcal{D}'$ is the same as $\mathcal{D}$.

### 10.1.3 Subgroups of Elliptic Curves

Let $E$ be an elliptic curve defined over an explicitly given finite field $F$, as in Clause 5.4. Let $\mathcal{H}$ denote this group $E$. Let $\mathcal{G}$ denote a prime-order subgroup of $\mathcal{H}$, and let $\mathbf{g}$ be a generator for $\mathcal{G}$. Let $\mu$ be the order of $\mathcal{G}$, and $\nu$ be its index in $\mathcal{H}$.

Observe that $\mathcal{H}$ is not in general cyclic. If $\gcd(\mu, \nu) = 1$, then one may test if an element $\mathbf{a} \in \mathcal{H}$ lies in $\mathcal{G}$ by testing if $\mu \cdot \mathbf{a} = \mathbf{0}$. If $\gcd(\mu, \nu) \neq 1$, then more information about the group structure of $E$ is required in order to construct an efficient test for membership in $\mathcal{G}$.

The encoding/decoding maps $\mathcal{E}$ and $\mathcal{D}$ are implemented using the functions $EC2OSP_E$ and $OS2ECP_E$. Thus, the encoding of a point is an octet string of length either 1, $1 + \lceil \log_{256} |F| \rceil$, or $1 + 2\lceil \log_{256} |F| \rceil$. The set of allowable format specifiers may be chosen to be any non-empty subset of $\{\mathsf{uncompressed}, \mathsf{compressed}, \mathsf{hybrid}\}$. Thus, a concrete group defined using an elliptic curve may, but need not, allow multiple encoding formats.

The partial encoding map $\mathcal{E}'$ is defined as follows. Given a point $P$ on $E$, if $P = \mathcal{O}$, then the output is $FE2OSP_F(0_F)$, and if $P = (x, y) \neq \mathcal{O}$, where $x, y \in F$, then the output is $FE2OSP_F(x)$. Thus, the output of $\mathcal{E}'$ is an octet string of length $\lceil \log_{256} |F| \rceil$.

## 10.2 *ECIES-KEM*

This clause describes the key encapsulation mechanism *ECIES-KEM*.

NOTE *ECIES-KEM* is based on the work of Abdalla, Bellare, and Rogaway [1, 2].

### 10.2.1 System parameters

*ECIES-KEM* is a family of key encapsulation mechanisms, parameterized by the following system parameters:

— $\Gamma$: a concrete group

$$\Gamma = (\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}'),$$

as described in Clause 10.1;

— *KDF*: a key derivation function, as described in Clause 6.2;

— *CofactorMode*: one of two values: 0 or 1.

— *OldCofactorMode*: one of two values: 0 or 1.

— *CheckMode*: one of two values: 0 or 1.

— *SingleHashMode*: one of two values: 0 or 1.

— *KeyLen*: a positive integer.



 

Any combination of system parameters is allowed, except for the following restrictions:

— At most one of *CofactorMode*, *OldCofactorMode*, and *CheckMode* may be 1.

— If $\nu > 1$ and *CheckMode* $= 0$, then we must have $\gcd(\mu, \nu) = 1$.

The value of *ECIES-KEM.KeyLen* is defined to be equal to the value of the system parameter *KeyLen*.

NOTE The values of *CofactorMode* and *CheckMode* are used only by the decryption algorithm.

### 10.2.2 Key generation

The key generation algorithm *ECIES-KEM.KeyGen* takes no input, and runs as follows.

a) Generate a random number $x \in [1\,..\,\mu)$.

b) Compute $\mathbf{h} = x \cdot \mathbf{g}$.

c) Output the public key:

— $\mathbf{h}$: a non-zero element of $\mathcal{G}$.

d) Output the private key:

— $x$: an integer in the set $[1\,..\,\mu)$

### 10.2.3 Encryption

The encryption algorithm *ECIES-KEM.Encrypt* takes as input a public key, consisting of $\mathbf{h} \in \mathcal{G} \setminus \{\mathbf{0}\}$, together with an encryption option *fmt* that specifies the format to be used for encoding group elements. It runs as follows.

a) Generate a random number $r \in [1\,..\,\mu)$.

b) If *OldCofactorMode* $= 1$, then set $r' = r \cdot \nu \bmod \mu$; otherwise, set $r' = r$.

c) Compute $\tilde{\mathbf{g}} = r \cdot \mathbf{g}$ and $\tilde{\mathbf{h}} = r' \cdot \mathbf{h}$.

d) Set $C_0 = \mathcal{E}(\tilde{\mathbf{g}}, fmt)$.

e) If *SingleHashMode* $= 1$, then let $Z$ be the null octet string; otherwise, let $Z = C_0$.

f) Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.

g) Set $K = KDF(Z \,\|\, PEH, KeyLen)$.

h) Output the ciphertext $C_0$ and the secret key $K$.

### 10.2.4 Decryption

The decryption algorithm *ECIES-KEM.Decrypt* takes as input a private key, consisting of $x \in [1\,..\,\mu)$, and a ciphertext $C_0$. It runs as follows.



 

a) Set $\tilde{\mathbf{g}} = \mathcal{D}(C_0)$; this step **fails** if $C_0$ is not a valid encoding of an element of $\mathcal{H}$.

b) If *CheckMode* $= 1$, test if $\tilde{\mathbf{g}} \in \mathcal{G}$; if not, then **fail**.

c) If *CofactorMode* $= 1$ or *OldCofactorMode* $= 1$, set $\hat{\mathbf{g}} = \nu \cdot \tilde{\mathbf{g}}$; otherwise, set $\hat{\mathbf{g}} = \tilde{\mathbf{g}}$.

d) If *CofactorMode* $= 1$, then set $\hat{x} = \nu^{-1} x \bmod \mu$; otherwise, set $\hat{x} = x$.

e) Compute $\tilde{\mathbf{h}} = \hat{x} \cdot \hat{\mathbf{g}}$.

f) If $\tilde{\mathbf{h}} = \mathbf{0}$, then **fail**.

g) If *SingleHashMode* $= 1$, then let $Z$ be the null octet string; otherwise, let $Z = C_0$.

h) Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.

i) Set $K = KDF(Z \,\|\, PEH, KeyLen)$.

j) Output the secret key $K$.

NOTE 1 Using *CofactorMode* $= 1$ or *OldCofactorMode* $= 1$ may yield a significant performance benefit if $\nu$ is fairly small. An advantage of using *CofactorMode* $= 1$ is that the behavior of the encryption algorithm is not affected by the value of *CofactorMode*.

NOTE 2 When using *CofactorMode* $= 1$, an implementation could simply pre-compute and store the value $\hat{x}$, instead of the value $x$.

NOTE 3 When using *SingleHashMode* $= 1$, even if $\Gamma$ supports multiple encoding formats, the value of *fmt* used during encryption does not affect any of the computations, except for the format of the resulting ciphertext. Thus, given a ciphertext $C_0$ that is an encoding of a group element $\tilde{\mathbf{g}}$, any ciphertext $C_0'$ that is also an encoding of $\tilde{\mathbf{g}}$ will decrypt in the same way as $C_0$.

NOTE 4 A discussion of the security of this scheme can be found in Annex B.9.

## 10.3 *PSEC-KEM*

This clause describes the key encapsulation mechanism *PSEC-KEM*.

NOTE *PSEC-KEM* is based on the work of Fujisaki and Okamoto [26].

### 10.3.1 System parameters

*PSEC-KEM* is a family of key encapsulation mechanisms, parameterized by the following system parameters:

— $\Gamma$: a concrete group

$$\Gamma = (\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}'),$$

as described in Clause 10.1;

— *KDF*: a key derivation function, as described in Clause 6.2;

— *SeedLen*: a positive integer;

— *KeyLen*: a positive integer.



 

### 10.3.2 Key Generation

The key generation algorithm *PSEC-KEM.KeyGen* takes no input, and runs as follows.

a) Generate a random number $x \in [0\,..\,\mu)$.

b) Compute $\mathbf{h} = x \cdot \mathbf{g}$.

c) Output the public key:

— $\mathbf{h}$: an element of $\mathcal{G}$.

d) Output the private key:

— $x$: an integer in the set $[0\,..\,\mu)$.

### 10.3.3 Encryption

Let $I0 = I2OSP(0, 4)$ and $I1 = I2OSP(1, 4)$.

The encryption algorithm *PSEC-KEM.Encrypt* takes as input a public key, consisting of $\mathbf{h} \in \mathcal{G}$, together with an encryption option *fmt* that specifies the format to be used for encoding group elements. It runs as follows.

a) Generate a random octet string *seed* of length *SeedLen*.

b) Compute

$$t = KDF(I0 \,\|\, seed, \lceil \log_{256} \mu \rceil + 16 + KeyLen),$$

an octet string of length $\lceil \log_{256} \mu \rceil + 16 + KeyLen$.

c) Parse $t$ as $t = u \,\|\, K$, where $u$ and $K$ are octet strings such that $|u| = \lceil \log_{256} \mu \rceil + 16$ and $|K| = KeyLen$.

d) Compute $r = OS2IP(u) \bmod \mu$.

e) Compute $\tilde{\mathbf{g}} = r \cdot \mathbf{g}$ and $\tilde{\mathbf{h}} = r \cdot \mathbf{h}$.

f) Set $EG = \mathcal{E}(\tilde{\mathbf{g}}, fmt)$ and $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.

g) Set $SeedMask = KDF(I1 \,\|\, EG \,\|\, PEH, SeedLen)$.

h) Set $MaskedSeed = seed \oplus SeedMask$.

i) Set $C_0 = EG \,\|\, MaskedSeed$.

j) Output the secret key $K$ and the ciphertext $C_0$.

### 10.3.4 Decryption

Let $I0 = I2OSP(0, 4)$ and $I1 = I2OSP(1, 4)$.

The decryption algorithm *PSEC-KEM.Decrypt* takes as input a private key, consisting of $x \in [0\,..\,\mu)$, and a ciphertext $C_0$. It runs as follows.



 

a) Parse $C_0$ as $C_0 = EG \,\|\, MaskedSeed$, $EG$ and $MaskedSeed$ are octet strings such that $|MaskedSeed| = SeedLen$; this step **fails** if $|C_0| < SeedLen$.

b) Set $\tilde{\mathbf{g}} = \mathcal{D}(EG)$; this step fails if $EG$ is not a valid encoding of a group element.

c) Compute $\tilde{\mathbf{h}} = x \cdot \tilde{\mathbf{g}}$.

d) Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.

e) Set $SeedMask = KDF(I1 \,\|\, EG \,\|\, PEH, SeedLen)$.

f) Set $seed = MaskedSeed \oplus SeedMask$.

g) Compute

$$t = KDF(I0 \,\|\, seed, \lceil \log_{256} \mu \rceil + 16 + KeyLen),$$

an octet string of length $\lceil \log_{256} \mu \rceil + 16 + KeyLen$.

h) Parse $t$ as $t = u \,\|\, K$, where $u$ and $K$ are octet strings such that $|u| = \lceil \log_{256} \mu \rceil + 16$ and $|K| = KeyLen$.

i) Compute $r = OS2IP(u) \bmod \mu$.

j) Compute $\bar{\mathbf{g}} = r \cdot \mathbf{g}$.

k) Test if $\bar{\mathbf{g}} = \tilde{\mathbf{g}}$; if not, then **fail**.

l) Output the secret key $K$.

NOTE A discussion of the security of this scheme can be found in Annex B.10.

## 10.4 *ACE-KEM*

This clause describes the key encapsulation mechanism *ACE-KEM*.

NOTE *ACE-KEM* is based on the work of Cramer and Shoup [13, 14].

### 10.4.1 System parameters

*ACE-KEM* is a family of key encapsulation mechanisms, parameterized by the following system parameters:

— $\Gamma$: a concrete group

$$\Gamma = (\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}'),$$

as described in Clause 10.1;

— *KDF*: a key derivation function, as described in Clause 6.2;

— *Hash*: a cryptographic hash function, as described in Clause 6.1;

— *CofactorMode*: one of two values: 0 or 1.

— *KeyLen*: a positive integer.



 

Any combination of allowable system parameters is allowed, except for the following restrictions:

— *Hash.len* must be less than $\log_{256} \mu$.

— If $\nu = 1$, then *CofactorMode* should be 0.

— If $\nu > 1$, *CofactorMode* may be 1 provided $\gcd(\mu, \nu) = 1$.

NOTE The value of *CofactorMode* is used only by the decryption algorithm.

### 10.4.2 Key generation

The key generation algorithm *ACE-KEM.KeyGen* takes no input, and runs as follows.

a) Generate random numbers $w, x, y, z \in [0\,..\,\mu)$.

b) Compute the group elements

$$\mathbf{g}' = w \cdot \mathbf{g},\ \mathbf{c} = x \cdot \mathbf{g},\ \mathbf{d} = y \cdot \mathbf{g},\ \mathbf{h} = z \cdot \mathbf{g}.$$

c) Output the public key:

— $\mathbf{g}', \mathbf{c}, \mathbf{d}, \mathbf{h}$: elements of $\mathcal{G}$.

d) Output the private key:

— $w, x, y, z$: integers in the set $[0\,..\,\mu)$.

### 10.4.3 Encryption

The encryption algorithm *ACE-KEM.Encrypt* takes as input a public key, consisting of

$$\mathbf{g}', \mathbf{c}, \mathbf{d}, \mathbf{h} \in \mathcal{G},$$

together with an encryption option *fmt* that specifies the format to be used for encoding group elements. It runs as follows.

a) Generate a random number $r \in [0\,..\,\mu)$.

b) Compute group elements

$$\mathbf{u} = r \cdot \mathbf{g},\ \mathbf{u}' = r \cdot \mathbf{g}',\ \tilde{\mathbf{h}} = r \cdot \mathbf{h}.$$

c) Compute the octet strings

$$EU = \mathcal{E}(\mathbf{u}, fmt),\ EU' = \mathcal{E}(\mathbf{u}', fmt).$$

d) Compute the integer

$$\alpha = OS2IP(Hash.eval(EU \,\|\, EU')).$$

e) Compute the integer

$$r' = \alpha \cdot r \bmod \mu.$$



 

f) Compute the group element

$$\mathbf{v} = r \cdot \mathbf{c} + r' \cdot \mathbf{d}.$$

g) Set $EV = \mathcal{E}(\mathbf{v}, fmt)$.

h) Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.

i) Set $C_0 = EU \,\|\, EU' \,\|\, EV$.

j) Set $K = KDF(EU \,\|\, PEH, KeyLen)$.

k) Output the ciphertext $C_0$ and the secret key $K$.

### 10.4.4 Decryption

The decryption algorithm *ACE-KEM.Decrypt* takes as input a private key, consisting of

$$w, x, y, z \in [0\,..\,\mu),$$

and a ciphertext $C_0$. It runs as follows.

a) Parse $C_0$ as $C_0 = EU \,\|\, EU' \,\|\, EV$, where $EU$, $EU'$, and $EV$ are octet strings such that for some (uniquely determined) group elements $\mathbf{u}, \mathbf{u}', \mathbf{v} \in \mathcal{H}$, we have $\mathbf{u} = \mathcal{D}(EU)$, $\mathbf{u}' = \mathcal{D}(EU')$, $\mathbf{v} = \mathcal{D}(EV)$. This step **fails** if $C_0$ cannot be so parsed.

b) Check that $\{EU, EU', EV\}$ is a consistent set of valid encodings; if not, then **fail**.

c) If $CofactorMode = 1$, set

$$\hat{\mathbf{u}} = \nu \cdot \mathbf{u},\ \hat{w} = \nu^{-1} w \bmod \mu,\ \hat{x} = \nu^{-1} x \bmod \mu,\ \hat{y} = \nu^{-1} y \bmod \mu,\ \hat{z} = \nu^{-1} z \bmod \mu;$$

otherwise, set

$$\hat{\mathbf{u}} = \mathbf{u},\ \hat{w} = w,\ \hat{x} = x,\ \hat{y} = y,\ \hat{z} = z.$$

d) If $CofactorMode \neq 1$ and $\nu > 1$: test if $\mathbf{u} \in \mathcal{G}$; if $\mathbf{u} \notin \mathcal{G}$, then **fail**.

e) Compute the integer

$$\alpha = OS2IP(Hash.eval(EU \,\|\, EU'))$$

f) Compute the integer

$$t = \hat{x} + \hat{y}\alpha \bmod \mu.$$

g) Test if

$$\hat{w} \cdot \hat{\mathbf{u}} = \mathbf{u}' \text{ and } t \cdot \hat{\mathbf{u}} = \mathbf{v}.$$

If not, then **fail**.

h) Compute the group element

$$\tilde{\mathbf{h}} = \hat{z} \cdot \hat{\mathbf{u}}.$$

i) Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.



 

j) Set $K = KDF(EU \,\|\, PEH, KeyLen)$.

k) Output the secret key $K$.

For security reasons, it is recommended that an implementation reveals no information about the cause of the error in Step g. In particular, an implementation should output the same error message at the same time, regardless of the cause of error.

NOTE 1 Using $CofactorMode = 1$ may yield a performance benefit if $\nu$ is fairly small. Note that in this mode, an implementation could simply pre-compute and store the values $\hat{w}, \hat{x}, \hat{y}, \hat{z}$, instead of the values $w, x, y, z$.

NOTE 2 An implementation is free to use the following, functionally equivalent, version of the decryption algorithm. The implementation need not necessarily compute $\mathbf{u}'$ and $\mathbf{v}$ in Step a of the decryption algorithm, but rather, simply syntactically parse $C_0$, obtaining $EU$, $EU'$, and $EV$, and convert only $EU$ to a group element $\mathbf{u}$. Step b may be omitted. Then the test in Step g of the decryption algorithm runs as follows: if $\mathbf{u} = \mathbf{0}$, then test if $EU'$ and $EV$ are (the unique) encodings of $\mathbf{0}$; otherwise, let $fmt$ be the format specifier of $EU$ (which is evident from $EU$ itself), and test if $\mathcal{E}(w \cdot \hat{\mathbf{u}}, fmt) = EU'$ and $\mathcal{E}(t \cdot \hat{\mathbf{u}}, fmt) = EV$.

NOTE 3 A detailed discussion of the security of this scheme can be found in Annex B.11.

# 11 RSA-based asymmetric ciphers and key encapsulation mechanisms

This clause describes asymmetric ciphers and key encapsulation mechanisms based on the RSA transform. The cipher *RSAES* is described in Clause 11.4; the key encapsulation mechanism *RSA-KEM* is described in Clause 11.5.

NOTE 1 These schemes are variations of the original RSA encryption scheme [31].

NOTE 2 In some other ISO standards, the term "integer factorization" is used in place of "RSA based"; however, as this standard defines several different schemes that are based on integer factorization, it adopts a new naming convention.

## 11.1 RSA key generation algorithms

An RSA key generation algorithm $RSAKeyGen()$ is a probabilistic algorithm that takes no input, and produces a triple $(n, e, d)$, where

— $n$ is an integer that is the product of two primes $p$ and $q$ of similar length, with $p \neq q$,

— $e$ is a positive integer such that $\gcd(e, (p-1)(q-1)) = 1$, and

— $d$ is a positive integer such that $e \cdot d \equiv 1 \pmod{\lambda(n)}$, where $\lambda(n)$ is the least common multiple of $(p-1)$ and $(q-1)$.

The output distribution of an RSA key generation algorithm depends on the particular algorithm. The algorithm is allowed to produce an output that fails to satisfy the above conditions, so long as this happens with negligible probability.

NOTE 1 In describing RSA-based ciphers, these ciphers are parameterized in terms of $RSAKeyGen$; i.e., $RSAKeyGen$ is treated as a system parameter of the cipher. In a typical implementation, a particular

RSA key generation algorithm may be selected from a family of such algorithms parameterized by a "security parameter" (e.g., the length of $n$).

NOTE 2 See ISO/IEC 18032 for guidance on designing algorithms for generating prime numbers $p$ and $q$ as above.

## 11.2 RSA transform

The algorithm $RSATransform(X, \alpha, n)$ takes as input

— an octet string $X$,

— a positive integer $\alpha$, and

— a positive integer $n$,

and outputs an octet string. It runs as follows:

a) Check if $|X| = \mathcal{L}(n)$; if not, then **fail**.

b) Set $x = OS2IP(X)$.

c) Check if $x < n$; if not, then **fail**.

d) Set $y = x^{\alpha} \bmod n$.

e) Set $Y = I2OSP(y, \mathcal{L}(n))$.

f) Output $Y$.

NOTE It is well known that if $(n, e, d)$ is the output of an RSA key generation algorithm and $X = I2OSP(x, \mathcal{L}(n))$ for some integer $x$ with $0 \leq x < n$, then

$$RSATransform(RSATransform(X, e, n), d, n) = X.$$

## 11.3 RSA encoding mechanisms

An RSA encoding mechanism *REM* specifies two algorithms:

— $REM.Encode(M, L, ELen)$ takes as input a plaintext $M$, a label $L$, and an output length $ELen$. Here, $M$ and $L$ are octet strings whose lengths are bounded, as described below. It outputs an octet string $E$ of length $ELen$.

— $REM.Decode(E, L)$ takes as input an octet string $E$ and a label $L$. It attempts to find a plaintext $M$ such that $REM.Encode(M, L, |E|) = E$. It returns $M$ if such an $M$ exists, and otherwise **fails**.

In addition to this, the mechanism should specify a bound $REM.Bound$ such that when $REM.Encode(M, L, ELen)$ is invoked, the condition $|M| \leq ELen - REM.Bound$ should hold; if not, the encoding algorithm **fails**. Additionally, the encoding algorithm may also **fail** if $|L|$ exceeds some (very large) implementation-defined bound.



 

The algorithm *REM.Encode* will in general be probabilistic, so that the same plaintext can be encoded in a number of ways. Also, for technical reasons, it is required that the first octet of the output of *REM.Encode* is always $Oct(0)$.

### 11.3.1 Allowable RSA encoding mechanisms

The only RSA encoding mechanism allowed in this part of ISO/IEC 18033 is *REM1*, described below in Clause 11.3.2.

### 11.3.2 *REM1*

This clause describes a particular RSA encoding mechanism, called *REM1*.

NOTE *REM1* is based on the OAEP construction of Bellare and Rogaway [8].

#### 11.3.2.1 System parameters

*REM1* is a family of RSA encoding mechanisms, parameterized by the following system parameters:

— *Hash*: a cryptographic hash function, as described in Clause 6.1;

— *KDF*: a key derivation function, as described in Clause 6.2.

The quantity *REM1.Bound* is defined as

$$REM1.Bound = 2 \cdot Hash.len + 2.$$

#### 11.3.2.2 Encoding function

The algorithm $REM1.Encode(M, L, ELen)$ runs as follows:

a) Check that $|M| \leq ELen - 2 \cdot Hash.len - 2$; if not, then **fail**.

b) Let *pad* be the octet string of length $ELen - |M| - 2 \cdot Hash.len - 2$ consisting of a sequence of $Oct(0)$ octets.

c) Generate a random octet string *seed* of length *Hash.len*.

d) Set $check = Hash.eval(L)$.

e) Set $DataBlock = check \,\|\, pad \,\|\, \langle\, Oct(1) \,\rangle \,\|\, M$.

f) Set $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$.

g) Set $MaskedDataBlock = DataBlockMask \oplus DataBlock$.

h) Set $SeedMask = KDF(MaskedDataBlock, Hash.len)$.

i) Set $MaskedSeed = SeedMask \oplus seed$.

j) Set $E = \langle\, Oct(0) \,\rangle \,\|\, MaskedSeed \,\|\, MaskedDataBlock$.

k) Output $E$.



 

#### 11.3.2.3 Decoding function

The algorithm $REM1.Decode(E, L)$ runs as follows.

a) Let $ELen = |E|$.

b) Check if $ELen \geq 2 \cdot Hash.len + 2$; if not, then **fail**.

c) Set $check = Hash.eval(L)$.

d) Parse $E$ as $E = \langle X \rangle \,\|\, MaskedSeed \,\|\, MaskedDataBlock$, where $X$ is an octet, and $MaskedSeed$ and $MaskedDataBlock$ are octet strings such that $|MaskedSeed| = Hash.len$, and $|MaskedDataBlock| = ELen - Hash.len - 1$.

e) Set $SeedMask = KDF(MaskedDataBlock, Hash.len)$.

f) Set $seed = MaskedSeed \oplus SeedMask$.

g) Set $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$.

h) Set $DataBlock = MaskedDataBlock \oplus DataBlockMask$.

i) Parse $DataBlock$ as $DataBlock = check' \,\|\, M'$, where $check'$ and $M'$ are octet strings such that $|check'| = Hash.len$ and $|M'| = ELen - 2 \cdot Hash.len - 1$.

j) Let $M' = \langle M_1, M_2, \ldots, M_l \rangle$, where $M_1, M_2, \ldots, M_l$ are octets, and $l = ELen - 2 \cdot Hash.len - 1$; also, let $m$ be the largest positive integer such that $m \leq l$ and $M_1 = M_2 = \cdots M_{m-1} = Oct(0)$, and let $T$ denote the octet $M_m$ and let $M$ denote the octet string $\langle M_{m+1}, \ldots, M_l \rangle$.

k) If $check' \neq check$, $X \neq Oct(0)$, or $T \neq Oct(1)$, then **fail**.

l) Output $M$.

For security reasons, it is essential that an implementation reveal no information about the cause of the error in Step k. In particular, an implementation should output the same error message at the same time, regardless of the cause of error.

## 11.4 *RSAES*

### 11.4.1 System parameters

*RSAES* is a family of bounded-plaintext-length asymmetric ciphers, parameterized by the following system parameters:

— *RSAKeyGen*: an RSA key generation algorithm, as described in Clause 11.1;

— *REM*: an RSA encoding mechanism, as described in Clause 11.3.

Any combination of system parameters is allowed, subject to the following restrictions:

— The length in octets of the output $n$ of $RSAKeyGen()$ must always be greater than $REM.Bound$.



 

### 11.4.2 Key generation

The algorithm *RSAES.KeyGen* takes no input, and runs as follows:

a) Compute $(n, e, d) = RSAKeyGen()$.

b) Output the public key $PK$:

— $n$: a positive integer.

— $e$: a positive integer.

c) Output the private key $pk$:

— $n$: a positive integer.

— $d$: a positive integer.

*RSAES* is a bounded-plaintext-length asymmetric cipher. For a given public key $PK = (n, e)$, the value of $RSAES.MaxMsgLen(PK)$ is $\mathcal{L}(n) - REM.Bound$.

The encryption and decryption algorithms make use of the *RSATransform* algorithm, defined in Clause 11.2.

### 11.4.3 Encryption

The algorithm *RSAES.Encrypt* takes as input

— a public key, consisting of a positive integer $n$, and a positive integer $e$,

— a label $L$,

— a plaintext $M$, whose length is at most $\mathcal{L}(n) - REM.Bound$, and

— no encryption option.

It runs as follows:

a) Set $E = REM.Encode(M, L, \mathcal{L}(n))$.

b) Set $C = RSATransform(E, e, n)$.

c) Output $C$.

### 11.4.4 Decryption

The algorithm *RSAES.Decrypt* takes as input

— a private key, consisting of a positive integer $n$, and a positive integer $d$,

— a label $L$, and



 

— a ciphertext $C$.

It runs as follows:

a) Set $E = RSATransform(C, d, n)$; note that this step may **fail**.

b) Set $M = REM.Decode(E, L)$; note that this step may **fail**.

c) Output $M$.

NOTE The security of *RSAES* is discussed in Annex B.13.

## 11.5 *RSA-KEM*

### 11.5.1 System parameters

*RSA-KEM* is a family of key encapsulation mechanisms, parameterized by the following system parameters:

— *RSAKeyGen*: an RSA key generation algorithm, as described in Clause 11.1;

— *KDF*: a key derivation function, as described in Clause 6.2;

— *KeyLen*: a positive integer.

The value of *RSA-KEM.KeyLen* is defined to be equal to the value of the system parameter *KeyLen*.

### 11.5.2 Key generation

The algorithm *RSA-KEM.KeyGen* takes no input, and runs as follows:

a) Compute $(n, e, d) = RSAKeyGen()$.

b) Output the public key $PK$:

— $n$: a positive integer.

— $e$: a positive integer.

c) Output the private key $pk$:

— $n$: a positive integer.

— $d$: a positive integer.

The encryption and decryption algorithms make use of the *RSATransform* algorithm, defined in Clause 11.2.

### 11.5.3 Encryption

The algorithm *RSA-KEM.Encrypt* takes as input



 

— a public key, consisting of a positive integer $n$, and a positive integer $e$, and

— no encryption option.

It runs as follows:

a) Generate a random number $r \in [0\,..\,n)$.

b) Set $R = I2OSP(r, \mathcal{L}(n))$.

c) Set $C_0 = RSATransform(R, e, n)$.

d) Compute $K = KDF(R, KeyLen)$.

e) Output the ciphertext $C_0$ and the secret key $K$.

### 11.5.4 Decryption

The algorithm *RSA-KEM.Decrypt* takes as input

— a private key, consisting of a positive integer $n$, and a positive integer $d$, and

— a ciphertext $C_0$.

It runs as follows:

a) Set $R = RSATransform(C_0, d, n)$; note that this step may **fail**.

b) Compute $K = KDF(R, KeyLen)$.

c) Output the secret key $K$.

NOTE The security of *RSA-KEM* is discussed in Annex B.14.



# 12 Ciphers based on modular squaring

This clause describes a family of asymmetric ciphers based on modular squaring. The cipher *HIME(R)* is described in Clause 12.3.

## 12.1 HIME key generation algorithms

For positive integers $l$ and $d > 1$, an $l$-bit HIME key generation algorithm *HIMEKeyGen* is a probabilistic algorithm that takes no input, and outputs positive integers $(p, q, d, n)$, where

— $p$ is a prime, with $2^{l-1} \leq p < 2^l$ and $p \equiv 3 \pmod 4$,

— $q$ is a prime, with $2^{l-1} \leq q < 2^l$, $q \equiv 3 \pmod 4$ and $p \neq q$,

— $n = p^d q$.

 

The output distribution of an $l$-bit HIME key generation algorithm depends on the particular algorithm. The algorithm is allowed to produce an output that fails to satisfy the above conditions, so long as this happens with negligible probability.

NOTE 1 In describing HIME-based ciphers, these schemes are parameterized in terms of *HIMEKeyGen*; i.e., *HIMEKeyGen* is treated as a system parameter of the cipher.

NOTE 2 See ISO/IEC 18032 for guidance on designing algorithms for generating prime numbers $p$ and $q$ as above.

## 12.2 HIME encoding mechanisms

A HIME encoding mechanism *HEM* specifies two algorithms:

— $HEM.Encode(M, L, ELen, KLen)$ takes as input a plaintext $M$, a label $L$, an output length $ELen$, and a positive integer $KLen$. $M$ and $L$ are octet strings whose lengths are bounded, as described below. $KLen$ satisfies $1 \leq KLen \leq 8$. It outputs an octet string $E$ of length $ELen$.

— $HEM.Decode(E, L, KLen)$ takes as input an octet string $E$ , a label $L$, and a positive integer $KLen$. It attempts to find a plaintext $M$ such that $HEM.Encode(M, L, |E|, KLen) = E$. It returns $M$ if such an $M$ exists, and otherwise **fails**.

### 12.2.1 Allowable HIME encoding mechanisms

The only HIME encoding mechanism allowed in this part of ISO/IEC 18033 is *HEM1*, described below in Clause 12.2.2.

### 12.2.2 *HEM1*

This clause describes a particular HIME encoding mechanism, called *HEM1*.

NOTE *HEM1* is based on the OAEP construction of Bellare and Rogaway [8].

#### 12.2.2.1 System parameters

*HEM1* is a family of HIME encoding mechanisms, parameterized by the following system parameters:

— *Hash*: a cryptographic hash function, as described in Clause 6.1;

— *KDF*: a key derivation function, as described in Clause 6.2.

The quantity *HEM1.Bound* is defined as

$$HEM1.Bound = 2 \cdot Hash.len + 2.$$

#### 12.2.2.2 Encoding function

The algorithm $HEM1.Encode(M, L, ELen, KLen)$ runs as follows:

a) Check that $|M| \leq ELen - 2 \cdot Hash.len - 2$; if not, then **fail**.



 

b) Let *pad* be the octet string of length $ELen - |M| - 2 \cdot Hash.len - 2$ consisting of a sequence of $Oct(0)$ octets.

c) Generate a random octet string *seed* of length $Hash.len + 1$.

d) Clear most significant *KLen*-bit of *seed*.

e) Set $check = Hash.eval(L)$.

f) Set $DataBlock = check \,\|\, pad \,\|\, \langle\, Oct(1) \,\rangle \,\|\, M$.

g) Set $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$.

h) Set $MaskedDataBlock = DataBlockMask \oplus DataBlock$.

i) Set $SeedMask = KDF(MaskedDataBlock, Hash.len + 1)$.

j) Clear most significant *KLen*-bit of *SeedMask*.

k) Set $MaskedSeed = SeedMask \oplus seed$.

l) Set $E = MaskedSeed \,\|\, MaskedDataBlock$.

m) Output $E$.

#### 12.2.2.3 Decoding function

The algorithm $HEM1.Decode(E, L, KLen)$ runs as follows.

a) Let $ELen = |E|$.

b) Set $check = Hash.eval(L)$.

c) Parse $E$ as $E = MaskedSeed \,\|\, MaskedDataBlock$, where *MaskedSeed* and *MaskedDataBlock* are octet strings such that $|MaskedSeed| = Hash.len + 1$, and $|MaskedDataBlock| = ELen - Hash.len - 1$.

d) Set $SeedMask = KDF(MaskedDataBlock, Hash.len + 1)$.

e) Clear most significant *KLen*-bit of *SeedMask*.

f) Set $seed = MaskedSeed \oplus SeedMask$.

g) Set $DataBlockMask = KDF(seed, ELen - Hash.len - 1)$.

h) Set $DataBlock = MaskedDataBlock \oplus DataBlockMask$.

i) Parse *DataBlock* as $DataBlock = check' \,\|\, M'$, where $check'$ and $M'$ are octet strings such that $|check'| = Hash.len$ and $|M'| = ELen - 2 \cdot Hash.len - 1$.

j) Let $M' = \langle\, M_1, M_2, \ldots, M_l \,\rangle$, where $M_1, M_2, \cdots, M_l$ are octets, and $l = ELen - 2 \cdot Hash.len - 1$; also, let $m$ be the largest positive integer such that $m \leq l$ and $M_1 = M_2 = \cdots M_{m-1} = Oct(0)$, and let $T$ denote the octet $M_m$ and let $M$ denote the octet string $\langle\, M_{m+1}, \ldots, M_l \,\rangle$.



 

k) If $check' \neq check$, most significant $KLen$-bit of $seed \neq$ bit string of 0, or $T \neq Oct(1)$, then **fail**.

l) Output $M$.

For security reasons, it is essential that an implementation reveals no information about the cause of the error in Step k. In particular, an implementation should output the same error message at the same time, regardless of the cause of error.

## 12.3 *HIME(R)*

### 12.3.1 System parameters

*HIME(R)* is a family of bounded-plaintext-length asymmetric ciphers, parameterized by the following system parameters:

— $d$: an integer with $d > 1$,

— *HIMEKeyGen*: an $l$-bit HIME key generation algorithm, as described in Clause 12.1;

— *HEM*: a HIME encoding mechanism, as described in Clause 12.2.

### 12.3.2 Key generation

The algorithm *HIME(R).KeyGen* takes no input, and runs as follows:

a) Compute $(p, q, n) = HIMEKeyGen()$.

b) Output the public key $PK$:

— $n$: a positive integer.

c) Output the private key $pk$:

— $n, p, q$: positive integers.

### 12.3.3 Encryption

The algorithm *HIME(R).Encrypt* takes as input

— a public key, consisting of a positive integer $n$,

— a label $L$,

— a plaintext $M$, whose length is at most $\mathcal{L}(n) - HEM.Bound$, and

— no encryption option.

It runs as follows:

a) Set $k = 8 \cdot \mathcal{L}(n) -$(bit length of $n$)$+1$.



 

b) Set $E = HEM.Encode(M, L, \mathcal{L}(n), k)$.

c) Set $e = OS2IP(E)$.

d) Set $c = e^2 \bmod n$.

e) Set $C = I2OSP(c, \mathcal{L}(n))$.

f) Output $C$.

### 12.3.4 Decryption

The algorithm *HIME(R).Decrypt* takes as input

— a private key, consisting of positive integers $n, p, q$,

— a label $L$, and

— a ciphertext $C$.

It runs as follows:

a) Set $c = OS2IP(C)$.

b) Set $k = 8 \cdot \mathcal{L}(n) - (\text{bit length of } n) + 1$.

c) Set $z = p^{-1} \bmod q$.

d) Set $c_p = c \bmod p$, and $c_q = c \bmod q$.

e) Set $\alpha_1 = c_p^{\frac{p+1}{4}} \bmod p$, and $\alpha_2 = p - \alpha_1$.

f) Set $\beta_1 = c_q^{\frac{q+1}{4}} \bmod q$ and $\beta_2 = q - \beta_1$.

g) Set

1) $u_0^{(1)} = \alpha_1$, and $u_1^{(1)} = (\beta_1 - u_0^{(1)})z \bmod q$.

2) $u_0^{(2)} = \alpha_1$, and $u_1^{(2)} = (\beta_2 - u_0^{(2)})z \bmod q$.

3) $u_0^{(3)} = \alpha_2$, and $u_1^{(3)} = (\beta_1 - u_0^{(3)})z \bmod q$.

4) $u_0^{(4)} = \alpha_2$, and $u_1^{(4)} = (\beta_2 - u_0^{(4)})z \bmod q$.

h) For $i$ from 1 to 4 do:

1) Set $v_1^{(i)} = u_0^{(i)} + u_1^{(i)} p$.

2) For $t$ from 2 to $d$ do:



 

i) Set $u_t^{(i)} = \left( (c - {v_{t-1}^{(i)}}^2 \bmod p^t q)/(p^{t-1}q) \right) (2u_0^{(i)})^{-1} \bmod p$.

ii) Set $v_t^{(i)} = v_{t-1}^{(i)} + u_t^{(i)} p^{t-1} q$.

3) Set $x_i = u_0^{(i)} + u_1^{(i)} p + \sum_{t=2}^{d} u_t^{(i)} p^{t-1} q$.

i) For $i$ from 1 to 4, set $X_i = I2OSP(x_i, \mathcal{L}(n))$.

j) If there exists a *unique* $i$ such that $HEM.Decode(X_i, L, k)$ does not **fail**, and $x_i^2 \bmod n = c$, then, for such $i$, set $M = HEM.Decode(X_i, L, k)$, otherwise **fail**.

k) Output $M$.

NOTE A discussion of the security of this scheme can be found in Annex B.15.



 

# Annex A
(normative)
# ASN.1 syntax for object identifiers

This annex gives ASN.1 syntax for object identifiers, public keys, and parameter structures to be associated with the algorithms specified in this part of ISO/IEC 18033.

```
--#########################################################################

EncryptionAlgorithms-2 {
   iso(1) standard(0) encryption-algorithms(18033) part(2)
      asn1-module(0) algorithm-object-identifiers(0) }
   DEFINITIONS EXPLICIT TAGS ::= BEGIN

-- EXPORTS All; --

IMPORTS
BlockAlgorithms
FROM EncryptionAlgorithms-3 { iso(1) standard(0)
encryption-algorithms(18033) part(3)
asn1-module(0) algorithm-object-identifiers(0) }
HashFunctionAlgs, id-sha1, NullParms
FROM DedicatedHashFunctions { iso(1) standard(0)
hash-functions(10118) part(3) asn1-module(1)
dedicated-hash-functions(0) };



--#########################################################################

-- oid definitions

OID ::= OBJECT IDENTIFIER  -- alias

-- Synonyms --

is18033-2 OID ::= { iso(1) standard(0) is18033(18033) part2(2)}

id-ac OID ::= { is18033-2 asymmetric-cipher(1) }
id-kem OID ::= { is18033-2 key-encapsulation-mechanism(2) }
id-dem OID ::= { is18033-2 data-encapsulation-mechanism(3) }
id-sc OID ::= { is18033-2 symmetric-cipher(4) }
id-kdf OID ::= { is18033-2 key-derivation-function(5) }
id-rem OID ::= { is18033-2 rsa-encoding-method(6) }
id-hem OID ::= { is18033-2 himer-encoding-method(7) }
id-ft OID ::= { is18033-2 field-type(8) }

-- Asymmetric ciphers --

id-ac-rsaes OID ::= { id-ac rsaes(1) }
id-ac-generic-hybrid OID ::= { id-ac generic-hybrid(2) }
id-ac-himer OID ::= { id-ac himer(3) }
```



 

```
-- Key encapsulation mechanisms --

id-kem-ecies OID ::= { id-kem ecies(1) }
id-kem-psec OID ::= { id-kem psec(2) }
id-kem-ace OID ::= { id-kem ace(3) }
id-kem-rsa OID ::= { id-kem rsa(4) }

-- Data encapsulation mechanisms --

id-dem-dem1 OID ::= { id-dem dem1(1) }
id-dem-dem2 OID ::= { id-dem dem2(2) }
id-dem-dem3 OID ::= { id-dem dem3(3) }

-- Symmetric ciphers --

id-sc-sc1 OID ::= { id-sc sc1(1) }
id-sc-sc2 OID ::= { id-sc sc2(2) }

-- Key derivation functions --

id-kdf-kdf1 OID ::= { id-kdf kdf1(1) }
id-kdf-kdf2 OID ::= { id-kdf kdf2(2) }

-- rsa encoding methods --

id-rem-rem1 OID ::= { id-rem rem1(1) }

-- hime(r) encoding methods --

id-hem-hem1 OID ::= { id-hem hem1(1) }


-- new field types oids
-- id-ft-prime-field OID ::= { id-ft prime-field(1) }

-- used only to define new basis type
id-ft-characteristic-two OID ::= { id-ft characteristic-two(2) }
id-ft-odd-characteristic OID ::= { id-ft odd-characteristic(3) }


id-ft-characteristic-two-basis OID ::=
{ id-ft-characteristic-two basisType(1) }
charTwoPolynomialBasis OID ::=
{ id-ft-characteristic-two-basis
charTwoPolynomialBasis(1) }

id-ft-odd-characteristic-basis OID ::= { id-ft-odd-characteristic
basisType(1)}
oddCharPolynomialBasis OID ::= {id-ft-odd-characteristic-basis
oddCharPolynomialBasis(1)}

--#####################################################################
-- normative comment:
-- whenever values of public key structures defined in this module
-- are to be carried in the SubjectPublicKeyInfo structure defined
-- in X.509
-- the value of the subjectPublicKey shall be the bit string
-- corresponding to the DER encoding of the public key structure and
-- the value of the algorithm field shall be the algorithm identifier
```



 

```
-- (defined in this module) of the algorithm for which the public key
-- is intended
--#####################################################################

-- RSAES asymmetric cipher


rsaes ALGORITHM ::= {
OID id-RSAES-OAEP PARMS RsaesParameters
}

RsaesPublicKey ::= RSAPublicKey

-- taken from PKCS#1
RSAPublicKey ::= SEQUENCE {
modulus INTEGER,-- n
publicExponent INTEGER -- e
}

-- the pSource field from PKCS #1 is omitted as it has
-- the default (empty) value
-- it plays no role in the encryption algorithm
RsaesParameters ::= SEQUENCE {
hashFunction [0] HashFunction DEFAULT alg-sha1,
keyDerivationFunction [1] RsaesKeyDerivationFunction
DEFAULT alg-mgf1-sha1
}
RsaesKeyDerivationFunction ::=
AlgorithmIdentifier {{ RKDFAlgorithms }}

RKDFAlgorithms ALGORITHM ::= {
KDFAlgorithms                          |
{ OID id-mgf1 PARMS HashFunction }
}
-- MGF1 in PKCS #1 is equivalent to KDF1 here
-- id-mgf1 should be used instead of id-kdf-kdf1 for compatibility
-- with existing implementations

alg-mgf1-sha1 RsaesKeyDerivationFunction ::= {
    algorithm   id-mgf1,
    parameters  HashFunction : alg-sha1
}

alg-sha1 HashFunction ::= {
algorithm id-sha1,
parameters NullParms : NULL
}



--#####################################################################

-- HIME(R) asymmetric cipher

himer ALGORITHM ::= {
OID id-ac-himer PARMS HimerParameters
}
```



 

```
HimerPublicKey ::= INTEGER

HimerParameters::=SEQUENCE{
d INTEGER(2..MAX),
encodingMethod HimerEncodingMethod
}

HimerEncodingMethod ::= AlgorithmIdentifier {{ HemAlgorithms }}

HemAlgorithms ALGORITHM ::= {
{ OID id-hem-hem1 PARMS Hem1Parameters },

...  -- Expect additional algorithms --
}

Hem1Parameters ::= SEQUENCE {
hashFunction HashFunction,
keyDerivationFunction KeyDerivationFunction
}

--#####################################################################

-- HC asymmetric cipher

genericHybrid ALGORITHM ::= {
OID id-ac-generic-hybrid PARMS GenericHybridParameters
}

GenericHybridParameters ::= SEQUENCE {
kem KeyEncapsulationMechanism,
dem DataEncapsulationMechanism
}

--#####################################################################
-- normative comment:
-- in SubjectPublicKeyInfo structure defined in X.509, the algorithm
-- field shall follow the genericHybrid syntax, and the
-- subjectPublicKey field shall be a bit string corresponding to a
-- value of type EciesKemPublicKey, PsecKemPublicKey,
-- AceKemPublicKey, or RsaKemPublicKey, according to the kem field of
-- GenericHybridParameters
--#####################################################################

-- KEM information objects

KeyEncapsulationMechanism ::= AlgorithmIdentifier {{ KEMAlgorithms }}

KEMAlgorithms ALGORITHM ::= {
{ OID id-kem-ecies PARMS EciesKemParameters }  |
{ OID id-kem-psec  PARMS PsecKemParameters  }  |
{ OID id-kem-ace   PARMS AceKemParameters   }  |
{ OID id-kem-rsa   PARMS RsaKemParameters   },

...  -- Expect additional algorithms --
}
```



 

```
--#####################################################################

-- ECIES-KEM

-- this must be a non-zero element of the group given in
-- EciesKemParameters
EciesKemPublicKey ::= FieldElement


EciesKemParameters ::= SEQUENCE {
group Group  OPTIONAL,
keyDerivationFunction KeyDerivationFunction,
oldCofactorMode BOOLEAN,
singleHashMode BOOLEAN,
keyLength KeyLength
}

--#####################################################################

-- PSEC-KEM

-- an element of the group given in PsecKemParameters (may be 0)
PsecKemPublicKey ::= FieldElement

PsecKemParameters ::= SEQUENCE {
group Group  OPTIONAL,
keyDerivationFunction KeyDerivationFunction,
seedLength INTEGER (1..MAX),
keyLength KeyLength
}

--#####################################################################

-- ACE-KEM

-- all components of public key are elements of the group given in
-- AceKemParameters
AceKemPublicKey ::= SEQUENCE {
gPrime FieldElement,
c FieldElement,
d FieldElement,
h FieldElement
}

AceKemParameters ::= SEQUENCE {
group Group  OPTIONAL,
keyDerivationFunction KeyDerivationFunction,
hashFunction HashFunction,
keyLength KeyLength
}

--#####################################################################

-- RSA-KEM
RsaKemPublicKey ::= RSAPublicKey

RsaKemParameters ::= SEQUENCE {
keyDerivationFunction KeyDerivationFunction,
keyLength KeyLength
}
```



 

```
--#####################################################################
-- DEM specifications

DataEncapsulationMechanism ::=  AlgorithmIdentifier {{DEMAlgorithms}}

DEMAlgorithms ALGORITHM ::= {
{ OID id-dem-dem1 PARMS Dem1Parameters }  |
{ OID id-dem-dem2 PARMS Dem2Parameters }  |
{ OID id-dem-dem3 PARMS Dem3Parameters },

...  -- Expect additional algorithms --
}

Dem1Parameters ::= SEQUENCE{
symmetricCipher SymmetricCipher,
mac MacAlgorithm
}

Dem2Parameters ::= SEQUENCE{
symmetricCipher SymmetricCipher,
mac MacAlgorithm,
labelLength INTEGER (0..MAX)
}

Dem3Parameters ::= SEQUENCE{
mac MacAlgorithm,
msgLength INTEGER (0..MAX)
}
--#####################################################################

-- finite field, group, and elliptic curve representations

Group ::= CHOICE {
groupOid OBJECT IDENTIFIER,
groupHashId OCTET STRING, -- defined in RFC2528
groupParameters GroupParameters
}

GroupParameters ::= CHOICE {
explicitFiniteFieldSubgroup
[0] ExplicitFiniteFieldSubgroupParameters,
ellipticCurveSubgroup
[1] EllipticCurveSubgroupParameters
}

ExplicitFiniteFieldSubgroupParameters ::= SEQUENCE {
fieldID FieldID {{FieldTypes}},
generator FieldElement,
subgroupOrder INTEGER,
subgroupIndex INTEGER
}

FIELD-ID ::= TYPE-IDENTIFIER

FieldID { FIELD-ID:IOSet } ::= SEQUENCE {
fieldType FIELD-ID.&id({IOSet}),
parameters FIELD-ID.&Type({IOSet}{@fieldType}) OPTIONAL
}
```



 

```
FieldTypes FIELD-ID ::= {
{ Prime-p IDENTIFIED BY prime-field } |
{ Characteristic-two IDENTIFIED BY characteristic-two-field }|
{ Odd-characteristic IDENTIFIED BY id-ft-odd-characteristic },

...  -- expect additional field types
}


-- prime fieds
Prime-p ::= INTEGER


-- characteristic two fields
CHARACTERISTIC-TWO ::= TYPE-IDENTIFIER

-- when basis is gnBasis then the basis shall be an optimal
-- normal basis of Type T where T is determined as follows:
-- if an ONB of Type 2 exists for the given value of m then
-- T shall be 2, otherwise if an ONB of Type 1 exists for the
-- given value of m then T shall be 1, otherwise T shall be
-- the least value for which an ONB of Type T exists for the
-- given value of m
-- when basis is gnBasis then m shall not be divisible by 8
-- note: the above rule is from ANSI X9.62
-- note: for the given m and T the ONB is unique
Characteristic-two ::= SEQUENCE {
m INTEGER,-- extension degree
basis CHARACTERISTIC-TWO.&id({BasisTypes}),
parameters CHARACTERISTIC-TWO.&Type({BasisTypes}{@basis})
}

BasisTypes CHARACTERISTIC-TWO ::= {
{ NULL IDENTIFIED BY gnBasis } |
{ Trinomial IDENTIFIED BY tpBasis } |
{ Pentanomial IDENTIFIED BY ppBasis } |
{ CharTwoPolynomial IDENTIFIED BY charTwoPolynomialBasis },

... -- expect additional basis types
}

Trinomial ::= INTEGER

Pentanomial ::= SEQUENCE {
k1 INTEGER,
k2 INTEGER,
k3 INTEGER
}

-- characteric two general irreducible polynomial representation

-- the irreducible polymial
-- a(n)*x^n + a(n-1)*x^(n-1) + ... + a(1)*x + a(0)
-- is encoded in the bit string with a(n) in the first bit, the
-- following coefficients in the following bit positions and a(0)
-- in the last bit of the bit string (one could omit a(n) and a(0)
-- but it may be simpler and less error-prone to leave them in
-- the encoding)
```



 

```
-- the degree of the polynomial is to be inferred from the length
-- of the bit string
CharTwoPolynomial ::= BIT STRING


-- odd characteristic extension fields

ODD-CHARACTERISTIC ::= TYPE-IDENTIFIER

Odd-characteristic ::= SEQUENCE {
characteristic INTEGER(3..MAX),
degree INTEGER(2..MAX),
basis ODD-CHARACTERISTIC.&id({OddCharBasisTypes}),
parameters ODD-CHARACTERISTIC.&Type({OddCharBasisTypes}{@basis})
}

OddCharBasisTypes ODD-CHARACTERISTIC ::= {
{ OddCharPolynomial IDENTIFIED BY oddCharPolynomialBasis },

... -- expect additional basis types
}

-- the monic irreducible polynomial is encoded as follows
-- the leading coefficient is ignored
-- the remaining coefficients define an element of the finite field
-- which is encoded in an octet string using FE2OSP
OddCharPolynomial ::= FieldElement

EllipticCurveSubgroupParameters ::= SEQUENCE {
version INTEGER { ecpVer1(1) } (ecpVer1),
fieldID FieldID {{ FieldTypes }},
curve Curve,
generator ECPoint,
subgroupOrder INTEGER,
subgroupIndex INTEGER,
...
}

Curve ::= SEQUENCE {
aCoeff FieldElement,
bCoeff FieldElement,
seed BIT STRING OPTIONAL
}

--#####################################################################

-- auxiliary definitions

FieldElement ::= OCTET STRING -- obtained through FE2OSP
ECPoint ::= OCTET STRING -- obtained through EC2OSP

KeyLength  ::= INTEGER (1..MAX)

MacAlgorithm ::= AlgorithmIdentifier {{ MACAlgorithms }}

MACAlgorithms ALGORITHM ::= {
{ OID hMAC-SHA1 PARMS NULL }  ,
...  -- Expect additional algorithms --
}
```



 

```
HashFunction ::= AlgorithmIdentifier {{ HashFunctionAlgorithms }}

HashFunctionAlgorithms ALGORITHM ::= {
HashFunctionAlgs,-- from 10118-3
...   -- expect additional algorithms
}

KeyDerivationFunction ::= AlgorithmIdentifier {{ KDFAlgorithms }}

KDFAlgorithms ALGORITHM ::= {
{ OID id-kdf-kdf1 PARMS HashFunction }  |
{ OID id-kdf-kdf2 PARMS HashFunction }  ,

...  -- Expect additional algorithms --
}

SymmetricCipher ::= AlgorithmIdentifier {{ SymmetricAlgorithms }}

SymmetricAlgorithms ALGORITHM ::= {
{ OID id-sc-sc1 PARMS BlockCipher }  |
{ OID id-sc-sc2 PARMS BlockCipher },

...  -- Expect additional algorithms --
}

BlockCipher ::= AlgorithmIdentifier {{ BlockAlgorithms }}


--##################################################################

-- external OIDs

-- RSA-OAEP
pkcs-1  OID ::=  { iso(1) member-body(2)
                         us(840) rsadsi(113549) pkcs(1) 1 }

id-RSAES-OAEP  OID ::=  { pkcs-1 7 }
id-mgf1  OID ::= { pkcs-1 8 }

-- HMAC-SHA1

hMAC-SHA1 OID ::= {
   iso(1) identified-organization(3) dod(6) internet(1) security(5)
     mechanisms(5) 8 1 2 }


-- X9.62 finite field and basis types
ansi-X9-62 OID ::= { iso(1) member-body(2) us(840) 10045 }

id-fieldType OID ::= { ansi-X9-62 fieldType(1) }

prime-field OID ::= { id-fieldType 1 }
characteristic-two-field OID ::= { id-fieldType 2 }

-- characteristic two basis
id-characteristic-two-basis OID ::= { characteristic-two-field
basisType(3) }
```



 

```
gnBasis OID ::= { id-characteristic-two-basis 1 }
tpBasis OID ::= { id-characteristic-two-basis 2 }
ppBasis OID ::= { id-characteristic-two-basis 3 }

--#####################################################################

-- Cryptographic algorithm identification --

ALGORITHM ::= CLASS {
&id OBJECT IDENTIFIER  UNIQUE,
&Type OPTIONAL
}
   WITH SYNTAX { OID &id [PARMS &Type] }

AlgorithmIdentifier { ALGORITHM:IOSet } ::= SEQUENCE {
algorithm ALGORITHM.&id( {IOSet} ),
parameters ALGORITHM.&Type( {IOSet}{@algorithm} )  OPTIONAL
}


END  -- EncryptionAlgorithms-2 --
```



 

# Annex B
(informative)
# Security considerations

This annex discusses the security properties of the various cryptographic schemes described in this part of ISO/IEC 18033. For each type of scheme (e.g., asymmetric cipher, MAC algorithm, etc.), an appropriate formal definition of security is given, and for each particular scheme, the extent to which this definition is satisfied is discussed.

The security of several schemes can be proven formally, based on certain intractability assumptions, or based on the assumption that other, lower-level mechanisms are secure. These proofs are "reductions," which show how to turn an adversary $A$ that breaks the scheme into an adversary $A'$ that solves the presumed-to-be-hard problem or breaks the presumed-to-be-secure mechanism. In most cases, the "quality" of the reduction is indicated by quantitatively describing the relationship between the resource requirements (e.g., running time) and advantage (i.e., success probability) of $A$ and those of $A'$. A reduction is called "tight" if the resource requirements of $A'$ are not significantly greater than those of $A$, and if the advantage of $A'$ is not significantly less than that of $A$.

The approach to security taken here is "concrete," as in [6], rather than "asymptotic": security reductions are stated in terms of specific schemes, rather than in terms of families of schemes indexed by a "security parameter" that tends to infinity. However, some quantitative estimates will be stated using "big-O" notation, which imply hidden, but quite small, constants.

Some of the proofs of security are in the so-called "random oracle" model, which was first formalized in [7], and has since been used in the analysis of numerous cryptographic schemes in the literature. In the random oracle model, one models a hash function or key derivation function as a random function to which all algorithms as well as the adversary have only "black box," i.e., oracle, access. Such random oracle proofs of security are perhaps best viewed as heuristic proofs — it is conceivable that a scheme that is secure in the random oracle model can be broken without either breaking the underlying intractability or security assumptions, and without finding any particular weakness in the hash function or key derivation function [12]. Nevertheless, a random oracle proof does rule out a broad class of attacks.



## B.1 MAC algorithms

This clause describes the basic security property that shall be required of a MAC algorithm in this part of ISO/IEC 18033.

Consider a MAC algorithm $MA$, as defined in Clause 6.3.

Consider the following attack scenario. An octet string $T^*$ is chosen by the adversary, and a secret key $k'$ is chosen at random. The value $MAC^* = MA.eval(k', T^*)$ is given to the adversary. The adversary outputs a list of pairs $(T, MAC)$, where $T$ is an octet string with $T \neq T^*$ (and not necessarily of the same length as $T^*$), and $MAC$ is an octet string of length $MA.MACLen$. The adversary's *advantage* is defined to be the probability that for one such pair $(T, MAC)$, we have $MA.eval(k', T) = MAC$.

For a given adversary $A$ and a given MAC algorithm $MA$, the above advantage is denoted by $Adv_{MA}(A)$. If the adversary $A$ runs in time at most $t$, generates a list of at most $l$ pairs, and $T^*$ and all the $T$ are bounded in length by $l'$, then $A$ is called a $MA[t, l, l']$-*adversary*.

Security means that this advantage is negligible for any efficient adversary.

Although the "single message" attack model considered here is sufficient for constructing secure data encapsulation mechanisms, for many other applications, it is not sufficient, and a "multiple message" attack model must be considered. In the "multiple message" attack model, instead of just obtaining the value of $MA.eval(k', \cdot)$ at a single input $T^*$, the adversary is allowed to obtain the value of $MA.eval(k', \cdot)$ at many (adaptively chosen) inputs $T_1^*, \ldots, T_s^*$. As before, the adversary outputs a list of pairs $(T, MAC)$, but now with the restriction that $T \neq T_i^*$, for $1 \leq i \leq s$.

Clause 6.3.1 allows for the use of the MAC algorithms described in ISO/IEC 9797-1 and ISO/IEC 9797-2, all of which are designed to be secure in the "multiple message" attack model, and some of which can be proven secure in this attack model based on certain assumptions about the underlying cryptographic hash function.

## B.2 Block ciphers

This clause describes the basic security property that shall be required of a block cipher in this part of ISO/IEC 18033.

Consider a block cipher $BC$, as defined in Clause 6.4.

$BC$ is called a *pseudo-random permutation* if it is difficult for an adversary to distinguish between a random permutation on octet strings of length $BC.BlockLen$ and the permutation $b \mapsto BC.Encrypt(k, b)$ for a randomly chosen secret key $k$. In such an attack, the adversary is given *oracle access* to the permutation — either to the random permutation or to the block cipher — and must guess which is the case. To be a pseudo-random permutation means that for any efficient adversary, its success at guessing which is the case should be negligibly close to 1/2.

Clause 6.4.1 allows for the use of the block ciphers described in ISO/IEC 18033-3. Although there is little formal justification, experience suggests that these block ciphers do indeed behave as pseudo-random permutations.

## B.3 Symmetric ciphers

This clause describes the basic security property that shall be required of a symmetric cipher in this part of ISO/IEC 18033.

Consider a symmetric cipher $SC$, as defined in Clause 6.5.

Consider the following attack scenario. The adversary generates two plaintexts (octet strings) $M_0, M_1$ of equal length, a random secret key $k$ is generated, a random bit $b$ is chosen, and $M_b$ is encrypted under the secret key $k$. The resulting ciphertext $c$ is given to the adversary. The adversary makes a guess $\hat{b}$ at $b$. The adversary's *advantage* is defined to be $|Pr[\hat{b} = b] - 1/2|$.

For a given adversary $A$ and a given symmetric cipher $SC$, this advantage is denoted by $Adv_{SC}(A)$. If the adversary runs in time at most $t$, and the *output* of the encryption algorithm is at most $l$ octets in length, then $A$ is called a $SC[t, l]$-*adversary*.

Security means that this advantage is negligible for any efficient adversary.



 

Although the "single plaintext" attack model considered here is sufficient for constructing secure data encapsulation mechanisms, for many other applications, it is not sufficient. For some applications, one must consider a "multiple plaintext" attack model, where an adversary is allowed to adaptively obtain many encryptions of its choice, and not just a single encryption. This type of attack is also called a "chosen plaintext" attack. Still another type of attack is a "chosen ciphertext" attack, where an adversary is allowed to adaptively obtain decryptions of its choice.

### B.3.1 Security of *SC1*

This clause discusses the security of *SC1*, defined in Clause 6.5.2.

This is a symmetric cipher parameterized in terms of block cipher $BC$.

The basic cipher-block-chaining (CBC) mode with a random initial value (IV) is analyzed in [4], where it is shown to be secure against a "multiple plaintext" attack, as discussed above, assuming $BC$ is a pseudo-random permutation (see Annex B.2). The cipher *SC1* uses a fixed initial value; nevertheless, it is easy to adapt the proof of security in [4] to show that *SC1* is secure against "single plaintext" attacks, which is adequate for the constructions in this document.

Note that the paper [35] presents some attacks on *SC1*. However, the attacks in [35] are "chosen ciphertext" attacks, and are therefore not relevant here. Indeed, the padding scheme plays a role in the security of CBC encryption only when considering "chosen ciphertext" attacks.

### B.3.2 Security of *SC2*

This clause discusses the security of *SC2*, defined in Clause 6.5.3.

There is no known formal reduction which reduces the security of *SC2* to the security of some other mechanisms or the intractability of some problem. However, if one is willing to model a key derivation function as a random oracle, then of course, one should be willing to believe that *SC2* is a secure symmetric cipher.

## B.4 Asymmetric ciphers

This clause describes the basic security property that shall be required of an asymmetric cipher.

Consider an asymptotic cipher $AC$, as defined in Clause 7.

Consider the following "adaptive chosen ciphertext" attack scenario.

**Stage 1:** The key generation algorithm is run, generating a public key and private key. The adversary, of course, obtains the public key, but not the private key.

**Stage 2:** The adversary makes a series of arbitrary queries to a *decryption oracle*. Each query is a label/ciphertext pair $(L, C)$ that is decrypted by the decryption oracle, making use of the private key. The resulting decryption is given to the adversary; moreover, if the decryption algorithm **fails**, then this information is given to the adversary, and the attack continues. The adversary is free to construct these label/ciphertext pairs in an arbitrary way — it is certainly *not* required to compute them using the encryption algorithm.

**Stage 3:** The adversary prepares a label $L^*$ and two "target" plaintexts $M_0, M_1$ of equal length, and gives these to an *encryption oracle*. If the scheme supports any encryption



 

options, the adversary also chooses these. The encryption oracle chooses $b \in \{0,1\}$ at random, encrypts $M_b$ with label $L^*$, and gives the resulting "target" ciphertext $C^*$ to the adversary.

**Stage 4:** The adversary continues to submit label/ciphertext pairs $(L, C)$ to the decryption oracle, subject only to the restriction that $(L, C) \neq (L^*, C^*)$.

**Stage 5:** The adversary outputs $\hat{b} \in \{0,1\}$, and halts.

The *advantage* of $A$ in this game is defined to be $|\Pr[\hat{b} = b] - 1/2|$. For a given adversary $A$, and a given asymptotic cipher $AC$, this advantage is denoted by $Adv_{AC}(A)$. If the adversary runs in time $t$, makes at most $q$ decryption oracle queries, all ciphertexts output from the encryption oracle and input to the decryption oracle are at most $l$ octets in length, and the labels input to the encryption and decryption oracle are at most $l'$ octets in length, then $A$ is called a $AC[t, q, l, l']$-*adversary*.

Security means that this advantage is negligible for all efficient adversaries.

This definition, in slightly different form, was first proposed by Rackoff and Simon [30]. Here, the definition in [30] has been generalized to take into account the fact the plaintexts may be of variable length, and to take into account the role of labels. It is generally agreed in the cryptographic research community that this is the "right" security property for a general-purpose asymmetric cipher. This notion of security implies other useful properties, like *non-malleability* (see [15, 16]). Intuitively, non-malleability means that it should be hard to transform a given label/ciphertext pair $(L, C)$ encrypting a plaintext $M$ into a different pair $(L', C')$, such that the decryption of $C'$ with label $L'$ is related in some "interesting" way to $M$. See [11, 14, 5, 15, 16] for more on notions of security for asymmetric ciphers.

See [27] for a definition of a weaker notion of security, sometimes called security against "lunchtime" attacks. In that setting, security is defined as it has been defined here, except that the adversary is not allowed to make any decryption oracle queries in Stage 4. Although this may seem like a natural definition of security, it is actually inadequate for many applications, and is not a suitable notion of security for a general-purpose asymmetric cipher.

An even weaker notion of security is called "semantic" security, and is defined in [21]. In that setting, security is defined as it has been defined here, except that the adversary is not allowed to make any decryption oracle queries at all.

### B.4.1 Hiding the plaintext length

Note that in the attack game, the adversary is required to submit two target plaintexts of *equal* length to the encryption oracle. This restriction on the adversary reflects the fact that one cannot expect to hide the length of an encrypted plaintext from the adversary — for many ciphers, this will be evident from the length of the ciphertext. It is in general up to the *application* using the cipher to ensure that the length of a plaintext does not reveal sensitive information.

For bounded-plaintext-length asymmetric ciphers, the notion of security is the same as for the ordinary case, except that the adversary *is not* required to submit target plaintexts of equal length to the encryption oracle. This reflects the fact that such schemes should hide the length of an encrypted plaintext from the adversary.

For fixed-plaintext-length asymmetric ciphers, this issue simply does not arise.



 

### B.4.2 Benign malleability: a slightly weaker notion of security

The definition of security given above may be viewed as being unnecessarily strong. For example, suppose one takes an asymmetric cipher $AC$ that satisfies the definition above, and modifies it as follows, obtaining a new cipher $AC'$: the cipher $AC'$ is the same as $AC$, except that it appends a random octet to the ciphertext upon encryption, and ignores this extra octet upon decryption. Technically speaking, $AC'$ does not satisfy the definition given above for adaptive chosen ciphertext security, yet this seems counter-intuitive. Indeed, although $AC'$ is technically "malleable," it is only malleable in a "benign" sort of way: one can create alternative encryptions of the same plaintext, and these alternative encryptions are all clearly recognizable as such.

This clause describes a formal notion of security that precisely captures the intuitive notion of "benign malleability."

For a particular asymmetric cipher $AC$, a polynomial-time, 0/1-valued function *Equiv* is called an *equivalence predicate* for $AC$ if with overwhelming probability, the output of $AC.KeyGen$ is a pair $(PK, pk)$, such that for any label $L$ and any two ciphertexts $C$ and $C'$, we have

$$Equiv(PK, L, C, C') = 1 \quad \text{implies} \quad AC.Decrypt(pk, L, C) = AC.Decrypt(pk, L, C').$$

An asymmetric cipher $AC$ is called *benignly malleable* if there exists an equivalence predicate *Equiv* as above, and if it satisfies the definition of security given above for adaptive chosen ciphertext security, but with the following modification in the attack game: when the adversary submits a label/ciphertext pair $(L, C)$ to the decryption oracle in Stage 4, then instead of requiring that $(L, C) \neq (L^*, C^*)$, it is required that $L \neq L^*$ or $Equiv(PK, L, C, C^*) = 0$. For an adversary $A$, its advantage in this setting is denoted by $Adv'_{AC}(A)$.

## B.5 Key encapsulation mechanisms

This clause describes the basic security property that shall be required of a key encapsulation mechanism.

Consider a key encapsulation mechanism *KEM*, as defined in Clause 8.1.

Consider the following "adaptive chosen ciphertext" attack scenario.

**Stage 1:** The key generation algorithm is run, generating a public key and private key. The adversary, of course, obtains the public key, but not the private key.

**Stage 2:** The adversary makes a series of arbitrary queries to a *decryption oracle*. Each query is a ciphertext $C_0$ that is decrypted by the decryption oracle, making use of the private key. The resulting decryption is given to the adversary; moreover, if the decryption algorithm **fails**, then this information is given to the adversary, and the attack continues. The adversary is free to construct these ciphertexts in an arbitrary way — it is certainly *not* required to compute them using the encryption algorithm.

**Stage 3:** The adversary invokes an *encryption oracle*, supplying any encryption options, if the scheme supports them. The encryption oracle does the following:

a) Run the encryption algorithm, generating a pair $(K^*, C_0^*)$.

b) Generate a random octet string $\tilde{K}$ of length $KEM.KeyLen$.



 

c) Choose $b \in \{0,1\}$ at random.

d) If $b = 0$, output $(K^*, C_0^*)$; otherwise output $(\tilde{K}, C_0^*)$.

**Stage 4:** The adversary continues to submit ciphertexts $C_0$ to the decryption oracle, subject only to the restriction that $C_0 \neq C_0^*$.

**Stage 5:** The adversary outputs $\hat{b} \in \{0,1\}$, and halts.

The *advantage* of $A$ in this game is defined to be $|\Pr[\hat{b} = b] - 1/2|$. For a given adversary $A$, and a given key encapsulation mechanism $KEM$, this advantage is denoted by $Adv_{KEM}(A)$. If the adversary runs in time $t$, and makes at most $q$ decryption oracle queries, then $A$ is called a $KEM[t,q]$-*adversary*.

Security means that this advantage is negligible for all efficient adversaries.

### B.5.1 Benign malleability

This clause defines the notion of benign malleability for a key encapsulation mechanism, corresponding to the notion of benign malleability for an asymmetric cipher, as in Annex B.4.2.

For a particular key encapsulation mechanism $KEM$, a polynomial-time, 0/1-valued function $Equiv$ is called an *equivalence predicate* for $KEM$ if with overwhelming probability, the output of $KEM.KeyGen$ is a pair $(PK, pk)$, such that for any two ciphertexts $C_0$ and $C_0'$, we have

$$Equiv(PK, C_0, C_0') = 1 \quad \text{implies} \quad KEM.Decrypt(pk, C_0) = KEM.Decrypt(pk, C_0').$$

A key encapsulation mechanism $KEM$ is called *benignly malleable* if there exists an equivalence predicate $Equiv$ as above, and if it satisfies the definition of security given above for adaptive chosen ciphertext security, but with the following modification in the attack game: when the adversary submits a ciphertext pair $C_0$ to the decryption oracle in Stage 4, then instead of requiring that $C_0 \neq C_0^*$, it is required that $Equiv(PK, C_0, C_0^*) = 0$. For an adversary $A$, its advantage in this setting is denoted by $Adv'_{KEM}(A)$.

## B.6 Data encapsulation mechanisms

This clause describes the basic security property that shall be required of a data encapsulation mechanism.

Consider a key encapsulation mechanism $DEM$, as defined in Clause 8.2.

Consider the following attack scenario. The adversary generates two plaintexts (octet strings) $M_0, M_1$ of equal length, and a label $L^*$. A random secret key $K$ is generated. A random bit $b$ is chosen, and $M_b$ is encrypted under secret key $K$. The resulting ciphertext $C_1^*$ is given to the adversary. The adversary then submits a series of requests to a *decryption oracle*: each such request is a label/ciphertext pair $(L, C_1) \neq (L^*, C_1^*)$, and the decryption oracle responds with the decryption of $C_1$ with label $L$ under secret key $K$. The adversary makes a guess $\hat{b}$ at $b$. The adversary's *advantage* is defined as $|Pr[\hat{b} = b] - 1/2|$.

For a specific adversary $A$ and data encapsulation mechanism $DEM$, this advantage is denoted by $Adv_{DEM}(A)$. If the adversary runs in time $t$, makes at most $q$ decryption oracle queries, the ciphertexts output from the encryption oracle and input to the decryption oracle are at most



 

$l$ octets in length, and the labels input to the encryption and decryption oracle are at most $l'$ octets in length, then $A$ is called a $DEM[t, q, l, l']$-*adversary*.

Security means that this advantage is negligible for any efficient adversary.

### B.6.1 Security of *DEM1*, *DEM2*, and *DEM3*

This clause discusses the security of the data encapsulation mechanisms *DEM1* (see Clause 9.1), *DEM2* (see Clause 9.2), and *DEM3* (see Clause 9.3).

Consider the data encapsulation mechanism *DEM1*. This scheme is parameterized by a symmetric cipher *SC* and a MAC algorithm *MA*. It can be shown that if *SC* satisfies the definition of security in Annex B.3 and *MA* satisfies the definition of security in Annex B.1, then *DEM1* satisfies the definition of security in Annex B.6.

More specifically, for any $DEM1[t, q, l, l']$-adversary $A$, we have

$$Adv_{DEM1}(A) \leq Adv_{SC}(A_1) + Adv_{MA}(A_2),$$

where

— $A_1$ is a $SC[t_1, l'']$-adversary, with $t_1 \approx t$,

— $A_2$ is a $MA[t_2, q, l'']$-adversary, with $t_2 \approx t$, and

— $l'' = l - MA.MACLen$.

Similarly, for any $DEM2[t, q, l, l']$-adversary $A$, where necessarily $l' = DEM2.LabelLen$, we have

$$Adv_{DEM2}(A) \leq Adv_{SC}(A_1) + Adv_{MA}(A_2),$$

where

— $A_1$ is a $SC[t_1, l'']$-adversary, with $t_1 \approx t$,

— $A_2$ is a $MA[t_2, q, l'' + l']$-adversary, with $t_2 \approx t$, and

— $l'' = l - MA.MACLen$.

Similarly, for any $DEM3[t, q, l, l']$-adversary $A$, where necessarily $l = DEM3.MsgLen + MA.MACLen$, we have

$$Adv_{DEM3}(A) \leq Adv_{MA}(A_2),$$

where

— $A_2$ is a $MA[t_2, q, DEM3.MsgLen + l']$-adversary, with $t_2 \approx t$.

These bounds are easily established from the definitions. See, for example, [14] for a proof for *DEM2* with $LabelLen = 0$. The proofs for the other cases can be established along similar lines of reasoning to that in [14].



 

## B.7 Security of *HC*

This clause discusses the security of the generic hybrid cipher *HC*, defined in Clause 8.3. This scheme is parameterized in terms of a key encapsulation mechanism *KEM* and a data encapsulation mechanism *DEM*.

It can be shown that if *KEM* satisfies the definition of security in Annex B.5 and *DEM* satisfies the definition of security in Annex B.6, then *HC* satisfies the definition of security in Annex B.4.

More specifically, for any $HC[t, q, l, l']$-adversary $A$, we have

$$Adv_{HC}(A) \leq 2 \cdot Adv_{KEM}(A_1) + Adv_{DEM}(A_2).$$

where

— $A_1$ is a $KEM[t_1, q]$-adversary, with $t_1 \approx t$, and

— $A_2$ is a $DEM[t_2, q, l, l']$-adversary, with $t_2 \approx t$.

The above inequality does not take into account the possibility that *KEM.KeyGen* outputs a "bad" key pair (i.e., one for which the decryption algorithm does not act as the inverse of the encryption algorithm) with non-zero probability. In this case, one must simply add this probability (which is assumed to be negligible) to the right hand side of the above inequality.

This bound is easily established from the definitions. See, for example, [14] for a detailed proof in the case where there are no labels. The proof in the case of labels can be established along similar lines of reasoning to that in [14]. If *KEM* is benignly malleable (see Annex B.5.1), then one can easily show that *HC* is also benignly malleable (see Annex B.4.2) with the same security bound as above.



## B.8 Intractability assumptions related to concrete groups

This clause defines several intractability assumptions related to concrete groups.

Let $\Gamma = (\mathcal{H}, \mathcal{G}, \mathbf{g}, \mu, \nu, \mathcal{E}, \mathcal{D}, \mathcal{E}', \mathcal{D}')$ be a concrete group, as defined in Clause 10.1.

### B.8.1 The Computational Diffie-Hellman problem

The Computational Diffie-Hellman (CDH) problem for $\Gamma$ is as follows. On input $(x\mathbf{g}, y\mathbf{g})$, where $x, y \in [0\,..\,\mu)$, compute $xy \cdot \mathbf{g}$. It is assumed that the inputs are random, i.e., that $x$ and $y$ are randomly chosen from the set $[0\,..\,\mu)$.

The CDH assumption is the assumption that this problem is intractable.

Note that in general, it is not feasible to even identify a correct solution to the CDH problem (this is the Decisional Diffie-Hellman problem — see below). In analyzing cryptographic systems, the types of algorithms for solving the CDH that most naturally arise are algorithms that produce a list of candidate solutions to a given instance of the CDH problem. For any algorithm $A$ for the CDH problem that produces a list of group elements as output, $AdvCDH_{\Gamma}(A)$ denotes the probability that this list contains a correct solution to the input problem instance. If $A$ runs in time $t$ and produces a list of at most $l$ group elements, then $A$ is called a $CDH_{\Gamma}[t, l]$-*adversary*.

Note that in [32], it is shown how to take a $CDH_{\Gamma}[t, l]$-adversary $A$ with $\epsilon = AdvCDH_{\Gamma}(A)$, and a given value of $\delta$, and transform this into a $CDH_{\Gamma}[t', 1]$-adversary $A'$, such that $AdvCDH_{\Gamma}(A') =$

 

$1-\delta$, and such that $t'$ is roughly equal to $O(t \cdot \epsilon^{-1} \log(1/\delta))$, plus the time to perform

$$O(\epsilon^{-1} l \log(1/\delta) \log \mu + (\log \mu)^2)$$

additional group operations.

### B.8.2 The Decisional Diffie-Hellman problem

The Decisional Diffie-Hellman (DDH) problem for $\Gamma$ is as follows.

One defines two distributions.

Distribution $\mathbf{R}$ consists of triples $(x\mathbf{g}, y\mathbf{g}, z\mathbf{g})$, where $x, y, z$ are chosen at random from $[0\,..\,\mu)$. Let $X_{\mathbf{R}}$ denote a random variable sampled from this distribution.

Distribution $\mathbf{D}$ consists of triples $(x\mathbf{g}, y\mathbf{g}, z\mathbf{g})$, where $x, y$ are chosen at random from $[0\,..\,\mu)$, and $z = xy \bmod \mu$. Let $X_{\mathbf{D}}$ denote a random variable sampled from this distribution.

The problem is to distinguish these two distributions.

For an algorithm $A$ that outputs either 0 or 1, its "DDH advantage" is defined as

$$AdvDDH_{\Gamma}(A) = |\Pr[A(X_{\mathbf{R}}) = 1] - \Pr[A(X_{\mathbf{D}}) = 1]|\,.$$

If $A$ runs in time $t$, then it is called a $DDH_{\Gamma}[t]$-*adversary*.

The DDH assumption is that this advantage is negligible for all efficient algorithms.

See [10, 25, 26] for further discussion of the DDH and related problems.

### B.8.3 The Gap-CDH Problem

The Gap-CDH problem is the problem of solving the CDH problem with the aid of an oracle for the DDH problem. In this case, since an algorithm for this problem has access to a DDH oracle, one may assume that the output of the algorithm is a single group element, rather than a list of group elements.

The Gap-CDH assumption is the assumption that this problem is intractable.

For any "oracle" algorithm $A$, $AdvGapCDH_{\Gamma}(A)$ is defined to be the probability that it outputs a correct solution to a random instance of the CDH problem, given access to a DDH oracle for $\Gamma$. If $A$ runs in time at most $t$, and makes at most $q$ queries to the DDH oracle, then $A$ is called a $GapCDH_{\Gamma}[t, q]$-*adversary*.

See [29] for more details about this assumption.

## B.9 Security of *ECIES-KEM*

This clause discusses the security of the key encapsulation mechanism *ECIES-KEM*, defined in Clause 10.2.

This scheme is parameterized in terms of a concrete group $\Gamma$ (see Clause 10.1) and a key derivation function *KDF* (see Clause 6.2).



 

This scheme can be shown secure in the random oracle model, where *KDF* is modeled as a random oracle, assuming the Gap-CDH problem is hard.

More specifically, suppose that the system parameters of *ECIES-KEM* are selected so that $SingleHashMode = 0$ and

$$CheckMode + CofactorMode + OldCofactorMode > 0.$$

Then if $A$ is a *ECIES-KEM*$[t, q]$-adversary that makes at most $q'$ random oracle queries, then we have

$$Adv_{ECIES\text{-}KEM}(A) = O(AdvGapCDH_{\Gamma}(A')),$$

where

— $A'$ is a $GapCDH_{\Gamma}[t', O(q')]$-adversary, where $t' \approx t$.

This bound is essentially proved in [14], at least for the case where $CheckMode = 1$ and group elements have unique encodings. The other cases can be proved by similar reasoning.

Alternatively, suppose that the system parameters of *ECIES-KEM* are selected so that $SingleHashMode = 1$ and

$$CheckMode + CofactorMode + OldCofactorMode > 0.$$

In this case, *ECIES-KEM* is no longer secure against adaptive chosen ciphertext attacks, but it is benignly malleable (see Annex B.5.1). If $A$ is a *ECIES-KEM*$[t, q]$-adversary that makes at most $q'$ random oracle queries, then we have

$$Adv'_{ECIES\text{-}KEM}(A) = O(AdvGapCDH_{\Gamma}(A')),$$

where

— $A'$ is a $GapCDH_{\Gamma}[t', O(q \cdot q')]$-adversary, where $t' \approx t$.

Besides satisfying only a weaker definition of security, this reduction is not as tight as in the case where $SingleHashMode = 0$. Also, the quality of the reduction degrades even further with $SingleHashMode = 1$ when one considers the multi-plaintext model formally defined in [3], whereas the reduction does not degrade significantly when $SingleHashMode = 0$.

If

$$CheckMode + CofactorMode + OldCofactorMode = 0,$$

then in both of the above estimates, the term

$$AdvGapCDH_{\Gamma}(A'),$$

must be replaced by

$$\nu \cdot AdvGapCDH_{\Gamma}(A').$$

Therefore, this selection of system parameters should only be used when $\nu$ is very small.

Instead of analyzing *ECIES-KEM* in terms of the Gap-CDH assumption in the random oracle model, one can analyze it without the use of random oracles, but under a very specific and non-standard assumption. See [1, 2] for details.



 

## B.10 Security of *PSEC-KEM*

This clause discusses the security of the key encapsulation mechanism *PSEC-KEM*, defined in Clause 10.3.

This scheme is parameterized in terms of a concrete group $\Gamma$ (see Clause 10.1) and a key derivation function *KDF* (see Clause 6.2).

This scheme can be proven secure in the random oracle model, viewing *KDF* as a random oracle, assuming the CDH problem is hard.

More specifically, for a given value of the system parameter *SeedLen*, and for any *PSEC-KEM*$[t, q]$-adversary $A$ that makes at most $q'$ random oracle queries, we have

$$Adv_{PSEC\text{-}KEM}(A) = O(AdvCDH_{\Gamma}(A') + (q + q')(\mu^{-1} + 2^{-SeedLen})),$$

where $A'$ is a $AdvCDH_{\Gamma}[t', O(q + q')]$-adversary, with $t' \approx t$.

This bound is proven in [41].

Also, the security does not significantly degrade in the multi-plaintext model formally defined in [10].

## B.11 Security of *ACE-KEM*

This clause discusses the security of the key encapsulation mechanism *ACE-KEM*, defined in Clause 10.4.

This scheme is parameterized in terms of a concrete group $\Gamma$ (see Clause 10.1), a key derivation function *KDF* (see Clause 6.2), and a hash function *Hash* (see Clause 6.1).

This scheme can be proven secure assuming the DDH problem is hard — it is to be emphasized that this proof of security is *not* in the random oracle model. Instead, some specific, and fairly standard, assumptions are made about *KDF* and *Hash*.

More specifically, for any *ACE-KEM*$[t, q]$-adversary $A$, we have

$$Adv_{ACE\text{-}KEM}(A) = O(\ AdvDDH_{\Gamma}(A_1) + Adv_{Hash}(A_2) + Adv_{KDF}(A_3) + q \cdot \mu^{-1}\ ),$$

where:

— $A_1, A_2, A_3$ denote adversaries that run in time essentially the same as $A$.

— $Adv_{Hash}(A_2)$ denotes the probability that an adversary $A_2$, given encodings $EU1^*$ and $EU2^*$ of two independent, random elements in $\mathcal{G}$, can find encodings $EU1$ and $EU2$ of elements in $\mathcal{G}$, such that $(EU1, EU2) \neq (EU1^*, EU2^*)$, but

$$Hash.eval(EU1 \parallel EU2) = Hash.eval(EU1^* \parallel EU2^*).$$

If the group supports multiple encodings, the adversary can choose the format it wants when $EU1^*$ and $EU2^*$ are generated; furthermore, the adversary may choose to use the same or different formats in its choice of $EU1$ and $EU2$; however, $EU1^*$ and $EU2^*$ must be consistent encodings, and the same holds for $EU1$ and $EU2$.



 

If $CofactorMode = 1$, then the adversary may choose $EU1$ to be an encoding of an element of $\mathcal{H}$ that does not necessarily lie in $\mathcal{G}$.

Note that this problem is a second-preimage collision problem, which is generally believed to be a much harder problem to solve than the problem of finding an arbitrary pair of colliding inputs.

— $Adv_{KDF}(A_3)$ denotes the advantage that an adversary $A_3$ has in distinguishing between the following two distributions. Let $\mathbf{u}_1$ and $\tilde{\mathbf{h}}$ be independent, random elements of $\mathcal{G}$, and let $EU1$ be an encoding of $\mathbf{u}_1$. Let $R$ be a random octet string of length $KeyLen$. The first distribution is $(R, EU1)$, and the second is $(KDF(EU1 \,\|\, \mathcal{E}'(\tilde{\mathbf{h}}), KeyLen), EU1)$.

The reader is referred to [14] for a detailed proof for the case where $CofactorMode = 0$ and group elements have unique encodings. The proof is easily adapted to handle the other cases as well, making use of the fact that the decryption algorithm checks for consistent encodings.

It is also shown in [14] that *ACE-KEM* is no less secure than *ECIES-KEM*, in the sense that for any *ACE-KEM*$[t, q]$-adversary $A$, there exists a *ECIES-KEM*$[t', q]$-adversary $A'$ such that $t' \approx t$ and $Adv_{ECIES\text{-}KEM}(A') \approx Adv_{ACE\text{-}KEM}(A)$. The proof in [14] is only for the case where $CofactorMode = 0$ and group elements have unique encodings. The proof is easily adapted to handle the other cases as well, again making use of the fact that the decryption algorithm checks for consistent encodings.

It is also shown in [14] that if *KDF* is viewed as a random oracle, then the security of *ACE-KEM* can be proven based on the CDH assumption. However, this security reduction is not very tight. The proof in [14] is only for the case where $CofactorMode = 0$ and group elements have unique encodings. The proof is easily adapted to handle the other cases as well.

As pointed out in Clause 10.4.4, care should be taken in the implementation of *ACE-KEM.Decrypt*. Specifically, the implementation of *ACE-KEM.Decrypt* should not reveal the cause of the error in Step g. If an attacker can obtain such information from a decryption oracle, the proof of security under the DDH assumption will no longer be valid; however, even if such information is available, no attack on the scheme is known, and moreover, the scheme is still no less secure than *ECIES-KEM*.



## B.12 The RSA inversion problem

This clause discusses the RSA inversion problem.

Let *RSAKeyGen* be an RSA key generation algorithm (see Clause 11.1).

The RSA inversion problem is this: given outputs $n$ and $e$ of $RSAKeyGen()$, along with random $x \in [0\,..\,n)$, compute $y$ such that $y^e \equiv x \pmod{n}$. For any given algorithm $A$ and any given RSA key generation algorithm *RSAKeyGen*, $Adv_{RSAKeyGen}(A)$ denotes the probability that $A$ solves the RSA inversion problem, as above. If $A$ runs in time at most $t$, then it is called a *RSAKeyGen*$[t]$*-adversary.*

The RSA assumption for *RSAKeyGen* is the assumption $Adv_{RSAKeyGen}(A)$ is negligible for any efficient algorithm $A$.

 

## B.13 Security of *RSAES*

This clause discusses the security of the bounded-plaintext-length asymmetric cipher *RSAES*, defined in Clause 11.4.

The paper [8] analyzes a more general setting in which (a minor variant of) the RSA encoding mechanism *REM1* (defined in Clause 11.3.2) is applied to a general "one-way trapdoor permutation," rather than to a specific function such as the RSA function. The analysis is done in the random oracle model, where the key derivation and hash functions are modeled as random oracles.

It is proven in [8] that the resulting scheme satisfies a technical property called "plaintext awareness," assuming the underlying permutation is indeed one way. However, as pointed out in [33], plaintext awareness *does not* imply security against adaptive chosen ciphertext attack — it only implies a weaker notion of security, namely, security against "lunchtime" attacks (see Annex B.4). Moreover, it is proven in [33] that *REM1* will in general not yield a cipher that is secure against adaptive chosen ciphertext attack, if the underlying permutation is *arbitrary*. This negative result does not imply that *RSAES* is insecure against adaptive chosen ciphertext attack — it only implies that the analysis in [8] does not establish this.

In [33], it is shown that *RSAES* is secure if the encryption exponent $e$ is very small (e.g., $e = 3$). This result was generalized in [20] to general encryption exponents. It should be pointed out, however, that the security reduction in [20] is not very tight — indeed, it is so bad that it actually says nothing at all about the security of *RSAES* for RSA moduli of up to several thousand bits. The security reduction in [33] for small encryption exponents is significantly better, but still is not quite as tight as one would like.

As pointed out in Clause 11.3.2.3, care must be taken in the implementation of *RSAES*. Specifically, it is essential that the implementation of *REM1.Decode* should not reveal the cause of the error in Step k; if an attacker can obtain such information from a decryption oracle, then the scheme can be easily broken, as described in [24].

## B.14 Security of *RSA-KEM*

This clause discusses the security of the key encapsulation mechanism *RSA-KEM*, defined in Clause 11.5.

This scheme can be easily shown to be secure in the random oracle model, where the system parameter *KDF* is modeled as a random oracle, assuming the RSA inversion problem is hard.

More specifically, for any RSA key generation algorithm *RSAKeyGen*, such that the output $(n, e, d)$ always satisfies $n \geq nBound$, and for any *RSA-KEM*$[t, q]$-adversary $A$, we have

$$Adv_{RSA\text{-}KEM}(A) \leq Adv_{RSAKeyGen}(A') + q/nBound,$$

where

— $A'$ is a *RSAKeyGen*$[t']$-adversary, with $t' \approx t$.

This inequality does not take into account the possibility that *RSAKeyGen* outputs a "bad" RSA key with non-zero probability. In this case, one must simply add this probability (which is assumed to be negligible) to the right hand side of the above inequality.



 

For a proof, see [34].

This security reduction is quite tight, unlike those for *RSAES* discussed above in Annex B.13. Moreover, in the multi-plaintext model formally defined in [3], the security of *RSA-KEM* does not degrade at all, due to the random self-reducibility of the RSA inversion problem. In contrast, the security of *RSAES* degrades linearly in the number of plaintexts, as the random self-reducibility property unfortunately cannot be exploited in this context.

Also, unlike *RSAES*, *RSA-KEM* does not seem to be susceptible to "implementation" attacks, such as the attack in [24].

## B.15 Security of *HIME(R)*

It can be shown that in the random oracle model, where the functions *Hash* and *KDF* in *HEM1* are modeled as random oracles, that *HIME(R)* is secure against adaptive chosen ciphertext attack, assuming that it is computationally infeasible to factor integers of the form output by algorithm *HIMEKeyGen*. For details, see [29, 35] — note that [35] corrects several mistakes in [29].



 

# Annex C
(informative)
# Test vectors

This annex gives test vectors for the encryption schemes specified in this part of ISO/IEC 18033.

For the ElGamal-based key encapsulation mechanisms, the "Modp" group is a subgroup of $\mathbf{Z}_p^*$ for the given prime $p$; the "ECModp" group is the elliptic curve over $\mathbf{Z}_p$ that is sometimes called "P192" in other standards; the "ECGF2" group is the elliptic curve over the finite field of $2^{163}$ elements that is sometimes called "B163" in other standards (elements of the field are represented with respect to the polynomial basis for the given irreducible polynomial $p$).

## C.1 Test vectors for *DEM1*

### C.1.1 Test vector

```
-------------
DEM1
-------------

SC=SC1(BC=AES(keylen=32))
MAC=HMAC(Hash=Sha1(), keylen=20, outlen=20)

--------------------------------
Trace for DEM1 encrypt
--------------------------------

Message in ASCII = "the rain in spain falls mainly on the plain"

Message as octet string = 0x746865207261696e20696e20737061696e2066616c6c
                          73206d61696e6c79206f6e2074686520706c61696e

Label in ASCII = "test"

Label as octet string = 0x74657374

k = 0x6434363064303334306635613764353333643739636535636535396235633737

k' = 0x3863323837346633333330653033653032303536

c = 0x0745c5f99ad56fe3ae4ebbeddc5385493cf67a8fa3e3fcdda5d8c82308a8e2b04c
    a4ac32241b1036f20fbe1f3aed19a3

T = 0x0745c5f99ad56fe3ae4ebbeddc5385493cf67a8fa3e3fcdda5d8c82308a8e2b04c
    a4ac32241b1036f20fbe1f3aed19a3746573740000000000000020

MAC = 0x016072f3d5cd979bb49a7c350b233b724f64bba9

C1 = 0x0745c5f99ad56fe3ae4ebbeddc5385493cf67a8fa3e3fcdda5d8c82308a8e2b04
     ca4ac32241b1036f20fbe1f3aed19a3016072f3d5cd979bb49a7c350b233b724f64
     bba9
```

### C.1.2 Test vector

```
-------------
DEM1
-------------

SC=SC2(Kdf=Kdf1(Hash=Sha1()), keylen=32)
MAC=HMAC(Hash=Sha1(), keylen=20, outlen=20)

--------------------------------
Trace for DEM1 encrypt
--------------------------------

Message in ASCII = "the rain in spain falls mainly on the plain"

Message as octet string = 0x746865207261696e20696e20737061696e2066616c6c
                          73206d61696e6c79206f6e2074686520706c61696e

Label in ASCII = "test"

Label as octet string = 0x74657374

k = 0x6434363064303334306635613764353333643739636535636535396235633737

k' = 0x38633238373466333333330653033653032303536

c = 0xae747466b1f160cf196d2ebe16ac9a70b6ff57c614436cf3de67ea38324f275791
    164cfcaea866b0024db7

T = 0xae747466b1f160cf196d2ebe16ac9a70b6ff57c614436cf3de67ea38324f275791
    164cfcaea866b0024db7746573740000000000000020

MAC = 0xa3462a9d5997aeaac247b33b6c13d748511e0f20

C1 = 0xae747466b1f160cf196d2ebe16ac9a70b6ff57c614436cf3de67ea38324f27579
     1164cfcaea866b0024db7a3462a9d5997aeaac247b33b6c13d748511e0f20
```

## C.2 Test vectors for *ECIES-KEM*

### C.2.1 Test vector

```
-------------
ECIES-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
CofactorMode=0
OldCofactorMode=0
CheckMode=1
SingleHashMode=0

-----

Group=Modp-Group:

p = 0x8a1b8d83ef967f4e8dc0a423a178b33f31a3aeb743fb332dc020970b44ba95bd29
    38eb60365ee9c1b1bda579d8276553758e84eb2a8f89c21f8c08ae12f2aacf
```



 

```
g = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a9207045605138 98d9
    4e40f3f6fc6a773249d63fc74bba14ceadc203b49f2344a6a22a0a8904c60b

mu = 0xdf0235fe94e74d2d70dbbc887389e5af9ec9ccd7

nu = 0x9e89f7f68e9a2e44b68affab0e53d03763d829685af48fa6405ce08865be6c7ee
     7221781300459df024b33e2

-----

Public Key


h = 0x61ddb01fad54cffe21a3a68c1cf388c23493699e74519931e42b8576a9652e47dc
    c65f7cd297039268d4a7d6b0337466415647a6f6204b6604d3659127f5c69f

-----

Private Key

x = 0x4a401de389f502aa4e1fb066b940a6784626a429


--------------------------------
Trace for ECIES-KEM encrypt
--------------------------------

r = 0x83bd99b480f6e3ab8b9dc4f410470949f9c9361d

r' = 0x83bd99b480f6e3ab8b9dc4f410470949f9c9361d

g~ = 0x5110f7e54f656e70c71ea2067c901570088a1eb1b230000abba1b2df4b774bed5
     43c0325b7083f2b477d5c02ddcafdfec0725672da2cbed39baf75f02dc078d0

h~ = 0x4e9752632f973db43ed3d06ffd5bd9e741af0f855cbc556b73ab530affd7850ca
     4c93d4b91d73b47db8718c05e296151e036cf9ba980cef6563af244438cac1b

PEH = 0x4e9752632f973db43ed3d06ffd5bd9e741af0f855cbc556b73ab530affd7850c
      a4c93d4b91d73b47db8718c05e296151e036cf9ba980cef6563af244438cac1b

z = 0x5110f7e54f656e70c71ea2067c901570088a1eb1b230000abba1b2df4b774bed54
    3c0325b7083f2b477d5c02ddcafdfec0725672da2cbed39baf75f02dc078d0

C0 = 0x5110f7e54f656e70c71ea2067c901570088a1eb1b230000abba1b2df4b774bed5
     43c0325b7083f2b477d5c02ddcafdfec0725672da2cbed39baf75f02dc078d0

K = 0x23e41472d780bfbb2daafd85a8fcdf8641fdca4d9f539a4ad175c473ca0f498728
    931bc311baa2c957ab528935aa22954075a2899ab1ce8ff5ba90a049aeba8cbb9019
    bccfc5c24c815ac8a1106e163936597b5d06ba4b52377ca48d82621b2768373a2103
    88998b964c11b0a2780c12c49cdea2cb454543fb3b725b026443d9
```

### C.2.2 Test vector

```
-------------
ECIES-KEM
-------------
```



 

```
Kdf=Kdf1(Hash=Sha1())
Keylen=128
CofactorMode=0
OldCofactorMode=0
CheckMode=0
SingleHashMode=0

-----

Group=ECModp-Group:

p = 0xffffffffffffffffffffffffffffffffeffffffffffffff

a = 0xffffffffffffffffffffffffffffffffeffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xfffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key


h(x) = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130

h(y) = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2

-----

Private Key

x = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3


-------------------------------
Trace for ECIES-KEM encrypt
-------------------------------

Encoding format = uncompressed_fmt

r = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8f21

r' = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8f21

g~(x) = 0xccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df

g~(y) = 0x047b2e07dd2ffb89359945f3d22ca8757874be2536e0f924

h~(x) = 0xcdec12c4cf1cb733a2a691ad945e124535e5fc10c70203b5

h~(y) = 0x0cae66e42ae0dd8857ab670c6397c93c1769f9a5f5b9d36d
```



 

```
PEH = 0xcdec12c4cf1cb733a2a691ad945e124535e5fc10c70203b5

z = 0x04ccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df047b2e07dd2ffb89
    359945f3d22ca8757874be2536e0f924

C0 = 0x04ccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df047b2e07dd2ffb8
     9359945f3d22ca8757874be2536e0f924

K = 0x9a709adeb6c7590ccfc7d594670dd2d74fcdda3f8622f2dbcf0f0c02966d5d9002
    db578c989bf4a5cc896d2a11d74e0c51efc1f8ee784897ab9b865a7232b5661b7cac
    87cf4150bdf23b015d7b525b797cf6d533e9f6ad49a4c6de5e7089724c9cadf0adf1
    3ee51b41be6713653fc1cb2c95a1d1b771cc7429189861d7a829f3
```

### C.2.3 Test vector

```
-------------
ECIES-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
CofactorMode=0
OldCofactorMode=0
CheckMode=0
SingleHashMode=0

-----

Group=ECModp-Group:

p = 0xffffffffffffffffffffffffffffffffefffffffffffffff

a = 0xffffffffffffffffffffffffffffffffefffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key


h(x) = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130

h(y) = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2

-----

Private Key

x = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3
```



 

```
------------------------------
Trace for ECIES-KEM encrypt
------------------------------

Encoding format = compressed_fmt

r = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8f21

r' = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8f21

g~(x) = 0xccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df

g~(y) = 0x047b2e07dd2ffb89359945f3d22ca8757874be2536e0f924

h~(x) = 0xcdec12c4cf1cb733a2a691ad945e124535e5fc10c70203b5

h~(y) = 0x0cae66e42ae0dd8857ab670c6397c93c1769f9a5f5b9d36d

PEH = 0xcdec12c4cf1cb733a2a691ad945e124535e5fc10c70203b5

z = 0x02ccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df

C0 = 0x02ccc9ea07b8b71d25646b22b0e251362a3fa9e993042315df

K = 0x8fbe0903fac2fa05df02278fe162708fb432f3cbf9bb14138d22be1d279f74bfb9
    4f0843a153b708fcc8d9446c76f00e4ccabef85228195f732f4aedc5e48efcf2968c
    3a46f2df6f2afcbdf5ef79c958f233c6d208f3a7496e08f505d1c792b314b45ff647
    237b0aa186d0cdbab47a00fb4065d62cfc18f8a8d12c78ecbee3fd
```

### C.2.4 Test vector

```
-------------
ECIES-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
CofactorMode=0
OldCofactorMode=0
CheckMode=0
SingleHashMode=0

-----

Group=ECGF2-Group:

p = 0x0800000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd

mu = 0x040000000000000000000292fe77e70c12a4234c33

nu = 0x01
```



 

```
g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1

-----

Public Key


h(x) = 0x03d401df33470c1eb3611ed1b9fd4dd12ffb48cbc1

h(y) = 0x057b470f90c82a900cc4daa27567d15b05d8bdbcb0

-----

Private Key

x = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c933


--------------------------------
Trace for ECIES-KEM encrypt
--------------------------------

Encoding format = uncompressed_fmt

r = 0xa9836a84a1583f601a2f9b2b2432a0aff42c8541

r' = 0xa9836a84a1583f601a2f9b2b2432a0aff42c8541

g~(x) = 0x0619b155dea55122f456a0b4741093a244893c91df

g~(y) = 0x03c75545c65707dd31d9a1a583aba4f107c0c2af51

h~(x) = 0x93c4a6f28021e71e1af8c9da440ab0317e12febd

h~(y) = 0x048d83cad5c3da366af4b7da10f5e13ec45eb1d65d

PEH = 0x0093c4a6f28021e71e1af8c9da440ab0317e12febd

z = 0x040619b155dea55122f456a0b4741093a244893c91df03c75545c65707dd31d9a1
    a583aba4f107c0c2af51

C0 = 0x040619b155dea55122f456a0b4741093a244893c91df03c75545c65707dd31d9a
     1a583aba4f107c0c2af51

K = 0x970d1027a42bb88402797cadc8b0822849218339f25189a624c1c7881a09814ede
    d59a9baafafd2ceb516d43b7c6594d1db583ac478bec07bfe37cc3d216a9a2929658
    fae29a7023e266abbdecff6ccecd19bd1f8e51d4db6329af82cae0c07ee093eb3188
    3c57511800057e60407d7d67210ba7366ae3b8b6877a9e81ecb774
```

### C.2.5 Test vector

```
-------------
ECIES-KEM
-------------
```

```
Kdf=Kdf1(Hash=Sha1())
Keylen=128
CofactorMode=0
OldCofactorMode=0
CheckMode=0
SingleHashMode=0

-----

Group=ECGF2-Group:

p = 0x080000000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd

mu = 0x04000000000000000000292fe77e70c12a4234c33

nu = 0x01

g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1

-----

Public Key


h(x) = 0x03d401df33470c1eb3611ed1b9fd4dd12ffb48cbc1

h(y) = 0x057b470f90c82a900cc4daa27567d15b05d8bdbcb0

-----

Private Key

x = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c933


--------------------------------
Trace for ECIES-KEM encrypt
--------------------------------

Encoding format = compressed_fmt

r = 0xa9836a84a1583f601a2f9b2b2432a0aff42c8541

r' = 0xa9836a84a1583f601a2f9b2b2432a0aff42c8541

g~(x) = 0x0619b155dea55122f456a0b4741093a244893c91df

g~(y) = 0x03c75545c65707dd31d9a1a583aba4f107c0c2af51

h~(x) = 0x93c4a6f28021e71e1af8c9da440ab0317e12febd

h~(y) = 0x048d83cad5c3da366af4b7da10f5e13ec45eb1d65d
```



 

```
PEH = 0x0093c4a6f28021e71e1af8c9da440ab0317e12febd

z = 0x030619b155dea55122f456a0b4741093a244893c91df

C0 = 0x030619b155dea55122f456a0b4741093a244893c91df

K = 0xdc66d10d56868d338b147186fdac210c351150862f94ff3ffcf4fc34b96c2117f1
    2e8cf39527419a96066ce00fd856b1742f3ec1865614d901b87ea7b89102417f9b62
    775e5806870e73db128fe00a0edd3efe21d93e84a4ae9609ade5838c96da784104db
    20170f74b430acde310785d4b66edd09d37f9f32c54ae44442c41f
```

## C.3 Test vectors for *PSEC-KEM*

### C.3.1 Test vector

```
-------------
PSEC-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
Seedlen=64

-----

Group=Modp-Group:

p = 0x8a1b8d83ef967f4e8dc0a423a178b33f31a3aeb743fb332dc020970b44ba95bd29
    38eb60365ee9c1b1bda579d8276553758e84eb2a8f89c21f8c08ae12f2aacf

g = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a9207045605138 98d9
    4e40f3f6fc6a773249d63fc74bba14ceadc203b49f2344a6a22a0a8904c60b

mu = 0xdf0235fe94e74d2d70dbbc887389e5af9ec9ccd7

nu = 0x9e89f7f68e9a2e44b68affab0e53d03763d829685af48fa6405ce08865be6c7ee
     7221781300459df024b33e2

-----

Public Key

h = 0x61ddb01fad54cffe21a3a68c1cf388c23493699e74519931e42b8576a9652e47dc
    c65f7cd297039268d4a7d6b0337466415647a6f6204b6604d3659127f5c69f

-----

Private Key

x = 0x4a401de389f502aa4e1fb066b940a6784626a429


-------------------------------
Trace for PSEC-KEM encrypt
-------------------------------

seed = 0x79878e0f7ef84d47753bf4b9a4fa5c33ec1bfa66fa140a3d998770496c613ad
       f8b9b6fdc083d4ac64f1960a9836a84a1583f601b1222a45b9ec718604eb67048
```

```
t = 0x583e88b2d550ec4b00419221470e635a63eb0ec74cb9fb6295b57c360e8b68eba9
    631b4e58bd6f118861b03d4dc8b12a3f2cb2e74a5a47e733f34e875891e980963615
    bad107bd2430e8e0d00c4f2d8f9306195b079ba4276900541f0fc7816815366b5190
    34810f6b0d6a6632e251a5ab70d176077701a9c048658a87178a4b94430190607b3a
    52cf66002e4d0251d2cf09f9b19cfbf4793251f7caf9d852a13ad7e37f

u = 0x583e88b2d550ec4b00419221470e635a63eb0ec74cb9fb6295b57c360e8b68eba9
    631b4e

r = 0x0a3b085c410f14847aa9c17ecae644cff418369e


g~ = 0x6e60226637400270f589f53577f00641538d241462441652cb18ffb244414789f
     6cfe71770e5248e74d80524927acd9b0242d273844f8415c4199d1b7037613f

h~ = 0x4ebe32dd0b9aa56cfb712581e7dcf9d8b5a4413544cbf6d09b074fa0d332ff335
     682de79a9a27cfae7a362f84c3e8ab15fca0ce2d1aae6aafc659438225c5559

EG = 0x6e60226637400270f589f53577f00641538d241462441652cb18ffb244414789f
     6cfe71770e5248e74d80524927acd9b0242d273844f8415c4199d1b7037613f

PEH = 0x4ebe32dd0b9aa56cfb712581e7dcf9d8b5a4413544cbf6d09b074fa0d332ff33
      5682de79a9a27cfae7a362f84c3e8ab15fca0ce2d1aae6aafc659438225c5559

SeedMask = 0xeab31c0d24a50c663d7e14d767cc2c4b5e2470deb00b09eab870d28ad0e
           a7c3a3cd05e998ce08c5a6f77a04e2d2b3b84c22d1747f36d5aff7794fbb0
           e27b7a80

MaskedSeed = 0x933492025a5d41214845e06ec3367078b23f8ab84a1f03d721f7a2c3b
             c8b46e5b74b314584ddc69c206ec0e7ae41bf259a12775ce14ffea4e953
             e3d0accd0ac8

C0 = 0x6e60226637400270f589f53577f00641538d241462441652cb18ffb244414789f
     6cfe71770e5248e74d80524927acd9b0242d273844f8415c4199d1b7037613f9334
     92025a5d41214845e06ec3367078b23f8ab84a1f03d721f7a2c3bc8b46e5b74b314
     584ddc69c206ec0e7ae41bf259a12775ce14ffea4e953e3d0accd0ac8

K = 0x58bd6f118861b03d4dc8b12a3f2cb2e74a5a47e733f34e875891e980963615bad1
    07bd2430e8e0d00c4f2d8f9306195b079ba4276900541f0fc7816815366b51903481
    0f6b0d6a6632e251a5ab70d176077701a9c048658a87178a4b94430190607b3a52cf
    66002e4d0251d2cf09f9b19cfbf4793251f7caf9d852a13ad7e37f
```

### C.3.2 Test vector

```
-------------
PSEC-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
Seedlen=64

-----

Group=ECModp-Group:

p = 0xffffffffffffffffffffffffffffffffeffffffffffffff
```



 

```
a = 0xfffffffffffffffffffffffffffffffefffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key

h(x) = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130

h(y) = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2

-----

Private Key

x = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3


--------------------------------
Trace for PSEC-KEM encrypt
--------------------------------

Encoding format = uncompressed_fmt

seed = 0xae8aeaf179878e0f7ef84d47753bf4b9a4fa5c33ec1bfa66fa140a3d9987704
       96c613adf8b9b6fdc083d4ac64f1960a9836a84a1583f601b1222a45b9ec71860

t = 0x336bbe43a45e8bb835c7fe866cf3501e9eff51d26d6d1dc10ae0775897f2f7a63f
    9d18df8a6880f99ed846a35852323b31b3b24eb1778db73a1195641b815990cf51ed
    62dd220189d600927c0fd9b19f8ddf5bde2305332cdbb202f915c76dca22bce645ea
    70b039ebbc12ac76d93590c4884062fca8a33ad29580fea2ddbf72e3746a334b8f5e
    f1f772aa09a6b7242df1fc806e605fcd45f50128f6d03db4c0581132f917f4e59d

u = 0x336bbe43a45e8bb835c7fe866cf3501e9eff51d26d6d1dc10ae0775897f2f7a63f
    9d18df8a6880f9

r = 0x9a53172304b54d475de3654019156aa4214a478cec066668


g~(x) = 0x87256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc0

g~(y) = 0x0c8e9ddf435a593e775339ed77b9f5f5bcc5097d0819c4b1

h~(x) = 0xb444acd74621f37573fcd0e79eb3a300fefd174b88cee971

h~(y) = 0x393eb322bac28badc949896dbff834da61954c1ebec59885

EG = 0x0487256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc00c8e9ddf435a593
     e775339ed77b9f5f5bcc5097d0819c4b1
```



 

```
PEH = 0xb444acd74621f37573fcd0e79eb3a300fefd174b88cee971

SeedMask = 0xda2ab7c99faf1b81e0ad09604c08b0978ebdef27be5bdce29c950fc061a
           3bb527eeb1aaae03e4082ba67effefa35fb8fb63c6457b049a1f5dcc0c321
           59530f7d

MaskedSeed = 0x74a05d38e628958e9e5544273933442e2a47b31452402684668105fdf
             824cb1b128a20756ba52f5eb25aa538b52c9b263556e0f6e876c1eecee2
             677ac794171d

C0 = 0x0487256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc00c8e9ddf435a593
     e775339ed77b9f5f5bcc5097d0819c4b174a05d38e628958e9e5544273933442e2a
     47b31452402684668105fdf824cb1b128a20756ba52f5eb25aa538b52c9b263556e
     0f6e876c1eecee2677ac794171d

K = 0x9ed846a35852323b31b3b24eb1778db73a1195641b815990cf51ed62dd220189d6
    00927c0fd9b19f8ddf5bde2305332cdbb202f915c76dca22bce645ea70b039ebbc12
    ac76d93590c4884062fca8a33ad29580fea2ddbf72e3746a334b8f5ef1f772aa09a6
    b7242df1fc806e605fcd45f50128f6d03db4c0581132f917f4e59d
```

### C.3.3 Test vector

```
-------------
PSEC-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
Seedlen=64

-----

Group=ECModp-Group:

p = 0xffffffffffffffffffffffffffffffffffffffeffffffffffffffff

a = 0xffffffffffffffffffffffffffffffffffffffeffffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key

h(x) = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130

h(y) = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2

-----
```



 

```
Private Key

x = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3


--------------------------------
Trace for PSEC-KEM encrypt
--------------------------------

Encoding format = compressed_fmt

seed = 0xae8aeaf179878e0f7ef84d47753bf4b9a4fa5c33ec1bfa66fa140a3d9987704
       96c613adf8b9b6fdc083d4ac64f1960a9836a84a1583f601b1222a45b9ec71860

t = 0x336bbe43a45e8bb835c7fe866cf3501e9eff51d26d6d1dc10ae0775897f2f7a63f
    9d18df8a6880f99ed846a35852323b31b3b24eb1778db73a1195641b815990cf51ed
    62dd220189d600927c0fd9b19f8ddf5bde2305332cdbb202f915c76dca22bce645ea
    70b039ebbc12ac76d93590c4884062fca8a33ad29580fea2ddbf72e3746a334b8f5e
    f1f772aa09a6b7242df1fc806e605fcd45f50128f6d03db4c0581132f917f4e59d

u = 0x336bbe43a45e8bb835c7fe866cf3501e9eff51d26d6d1dc10ae0775897f2f7a63f
    9d18df8a6880f9

r = 0x9a53172304b54d475de3654019156aa4214a478cec066668


g~(x) = 0x87256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc0

g~(y) = 0x0c8e9ddf435a593e775339ed77b9f5f5bcc5097d0819c4b1

h~(x) = 0xb444acd74621f37573fcd0e79eb3a300fefd174b88cee971

h~(y) = 0x393eb322bac28badc949896dbff834da61954c1ebec59885

EG = 0x0387256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc0

PEH = 0xb444acd74621f37573fcd0e79eb3a300fefd174b88cee971

SeedMask = 0xe63cf131069307ca1a2296e0ac3fa1afa25a6476a01254e56903c7301a5
           5dde0bd2cd68a28f2c94c867a0b8e4d6f825c041e63e463f6cabb1a9d290b
           f4c20673

MaskedSeed = 0x48b61bc07f1489c564dadba7d904551606a038454c09ae839317cd0d8
             3d2ada9d14dec55a369a6908e4741480276e2f58774e7453bc9aaa008bf
             8d506a051e13

C0 = 0x0387256b492f43b0cf7cf192faeb26ea354a0e19d1d9bdbbc048b61bc07f1489c
     564dadba7d904551606a038454c09ae839317cd0d83d2ada9d14dec55a369a6908e
     4741480276e2f58774e7453bc9aaa008bf8d506a051e13

K = 0x9ed846a35852323b31b3b24eb1778db73a1195641b815990cf51ed62dd220189d6
    00927c0fd9b19f8ddf5bde2305332cdbb202f915c76dca22bce645ea70b039ebbc12
    ac76d93590c4884062fca8a33ad29580fea2ddbf72e3746a334b8f5ef1f772aa09a6
    b7242df1fc806e605fcd45f50128f6d03db4c0581132f917f4e59d
```



 

### C.3.4 Test vector

```
-------------
PSEC-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
Seedlen=64

-----

Group=ECGF2-Group:

p = 0x0800000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd

mu = 0x040000000000000000000292fe77e70c12a4234c33

nu = 0x01

g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1

-----

Public Key

h(x) = 0x03d401df33470c1eb3611ed1b9fd4dd12ffb48cbc1

h(y) = 0x057b470f90c82a900cc4daa27567d15b05d8bdbcb0

-----

Private Key

x = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c933


-------------------------------
Trace for PSEC-KEM encrypt
-------------------------------

Encoding format = uncompressed_fmt

seed = 0xf179878e0f7ef84d47753bf4b9a4fa5c33ec1bfa66fa140a3d998770496c613
       adf8b9b6fdc083d4ac64f1960a9836a84a1583f601b1222a45b9ec718604eb670

t = 0xc6836e810a973cb54f73dc4b573505e2f1fe2b80c67633494fd53af386c73e42c5
    c4508d75b270dd95d81fff0518e500e42925ae1f699f498e8273e4884f31407b8a3a
    26aa6ee547d4f6b8448b72e9b05f51803bce733cf773bac707fb6127476ba914f74a
    5ad10ac0a7b87b59b9699a707a326924528af10911386c65388aebe88ebefa8ee2a1
    c9cca32a6d00d9833ca055f0437ee06379416cc139a7fb1900b8d3cadde2
```



 

```
u = 0xc6836e810a973cb54f73dc4b573505e2f1fe2b80c67633494fd53af386c73e42c5
    c4508d75

r = 0x02f40b3321460743cc5722182f8529f93ed53cc58c


g~(x) = 0x067ba0d66f34b80ade98971eaec46ae7df42e41864

g~(y) = 0x051879a0b595dacd15353f307a61f741467f1be232

h~(x) = 0x031878816c68b18a57a4528f1ae4247a33a319d4f5

h~(y) = 0x037b354c91ad6607a52fc1222972610dd4d0df1361

EG = 0x04067ba0d66f34b80ade98971eaec46ae7df42e41864051879a0b595dacd15353
     f307a61f741467f1be232

PEH = 0x031878816c68b18a57a4528f1ae4247a33a319d4f5

SeedMask = 0x4de1b17b54d897920299ffc57d414cc2f533521f737633dcc953ca8fd86
           e087722b7dae4df95d940c29d56fa08c6ead9f418786f092c993e5d6a314f
           fc6c6994

MaskedSeed = 0xbc9836f55ba66fdf45ecc431c4e5b69ec6df49e5158c27d6f4ca4dff9
             102694dfd3c418b039de40a04d24f9aa145805d5540470f123ebb9a06f4
             f6579c22dfe4

C0 = 0x04067ba0d66f34b80ade98971eaec46ae7df42e41864051879a0b595dacd15353
     f307a61f741467f1be232bc9836f55ba66fdf45ecc431c4e5b69ec6df49e5158c27
     d6f4ca4dff9102694dfd3c418b039de40a04d24f9aa145805d5540470f123ebb9a0
     6f4f6579c22dfe4

K = 0xb270dd95d81fff0518e500e42925ae1f699f498e8273e4884f31407b8a3a26aa6e
    e547d4f6b8448b72e9b05f51803bce733cf773bac707fb6127476ba914f74a5ad10a
    c0a7b87b59b9699a707a326924528af10911386c65388aebe88ebefa8ee2a1c9cca3
    2a6d00d9833ca055f0437ee06379416cc139a7fb1900b8d3cadde2
```

### C.3.5 Test vector

```
-------------
PSEC-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Keylen=128
Seedlen=64

-----

Group=ECGF2-Group:

p = 0x0800000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd

mu = 0x040000000000000000000292fe77e70c12a4234c33
```

```
nu = 0x01

g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1

-----

Public Key

h(x) = 0x03d401df33470c1eb3611ed1b9fd4dd12ffb48cbc1

h(y) = 0x057b470f90c82a900cc4daa27567d15b05d8bdbcb0

-----

Private Key

x = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c933


--------------------------------
Trace for PSEC-KEM encrypt
--------------------------------

Encoding format = compressed_fmt

seed = 0xf179878e0f7ef84d47753bf4b9a4fa5c33ec1bfa66fa140a3d998770496c613
       adf8b9b6fdc083d4ac64f1960a9836a84a1583f601b1222a45b9ec718604eb670

t = 0xc6836e810a973cb54f73dc4b573505e2f1fe2b80c67633494fd53af386c73e42c5
    c4508d75b270dd95d81fff0518e500e42925ae1f699f498e8273e4884f31407b8a3a
    26aa6ee547d4f6b8448b72e9b05f51803bce733cf773bac707fb6127476ba914f74a
    5ad10ac0a7b87b59b9699a707a326924528af10911386c65388aebe88ebefa8ee2a1
    c9cca32a6d00d9833ca055f0437ee06379416cc139a7fb1900b8d3cadde2

u = 0xc6836e810a973cb54f73dc4b573505e2f1fe2b80c67633494fd53af386c73e42c5
    c4508d75

r = 0x02f40b3321460743cc5722182f8529f93ed53cc58c


g~(x) = 0x067ba0d66f34b80ade98971eaec46ae7df42e41864

g~(y) = 0x051879a0b595dacd15353f307a61f741467f1be232

h~(x) = 0x031878816c68b18a57a4528f1ae4247a33a319d4f5

h~(y) = 0x037b354c91ad6607a52fc1222972610dd4d0df1361

EG = 0x03067ba0d66f34b80ade98971eaec46ae7df42e41864

PEH = 0x031878816c68b18a57a4528f1ae4247a33a319d4f5

SeedMask = 0xefce9dd9b8e3ebd1f563ead211fc08e3a21dca27d0a56ef447c201e85f3
           f33e144f6281fa60d1f94f8d31ee0bb791b276ede83dcda51d37ee35b1bb6
           1f349211
```



 

```
MaskedSeed = 0x1eb71a57b79d139cb216d126a858f2bf91f1d1ddb65f7afe7a5b86981
             65352db9b7db3707a0522de3e9c078012fa71a3cf86bcbcc143f1dab8c5
             dcae7f7a2461

C0 = 0x03067ba0d66f34b80ade98971eaec46ae7df42e418641eb71a57b79d139cb216d
     126a858f2bf91f1d1ddb65f7afe7a5b8698165352db9b7db3707a0522de3e9c0780
     12fa71a3cf86bcbcc143f1dab8c5dcae7f7a2461

K = 0xb270dd95d81fff0518e500e42925ae1f699f498e8273e4884f31407b8a3a26aa6e
    e547d4f6b8448b72e9b05f51803bce733cf773bac707fb6127476ba914f74a5ad10a
    c0a7b87b59b9699a707a326924528af10911386c65388aebe88ebefa8ee2a1c9cca3
    2a6d00d9833ca055f0437ee06379416cc139a7fb1900b8d3cadde2
```

## C.4 Test vectors for *ACE-KEM*

### C.4.1 Test vector

```
-------------
ACE-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=128
CofactorMode=0

-----

Group=Modp-Group:

p = 0x8a1b8d83ef967f4e8dc0a423a178b33f31a3aeb743fb332dc020970b44ba95bd29
    38eb60365ee9c1b1bda579d8276553758e84eb2a8f89c21f8c08ae12f2aacf

g = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a9207045605138 98d9
    4e40f3f6fc6a773249d63fc74bba14ceadc203b49f2344a6a22a0a8904c60b

mu = 0xdf0235fe94e74d2d70dbbc887389e5af9ec9ccd7

nu = 0x9e89f7f68e9a2e44b68affab0e53d03763d829685af48fa6405ce08865be6c7ee
     722178130045 9df024b33e2

-----

Public Key

g' = 0x32785f2307a7cb33cdf124e4349e8e6037040950e51171a4e3d47e0b7280b4798
     ec799752e8761d48de565a13962ad951a6322441074a3a7e001dd5bee6448e9

c = 0x84e3b74b067c33ea7ab19ac8e61863e704d56c43e96b14acfb2f2a056f4e72a413
    889732006a11bbd34e487e36084fab09c9ec7828308b76412d6a4753e55d31

d = 0x39967584286a71b1dc7fa5a486b26b9cfad2731a5902a8dcc611a5f37eae8d6e9c
    c8ad0948344e8edbe80fa607d1c35b2395487ff1aa94b66af9693e20a28027

h = 0x46d73cf934f674c1c9549c7b3e9460c826e2a52c31fd4c5d4cb8da9caddce1b493
    eec79ca9a9d6ec5377cf42d8d2968a28c4b183acc9a3bf0590d5bd147e1c14

-----
```



 

```
Private Key

w = 0x4a401de389f502aa4e1fb066b940a6784626a349

x = 0x83bd99b480f6e3ab8b9dc4f410470949f9c9355a

y = 0xa881357fe37c1047061a8192e51b5ebef3a34c23

z = 0x87b8cdd4253bbab89fae7e5c67b5dac6d637f3e7


--------------------------------
Trace for ACE-KEM encrypt
--------------------------------

r = 0x346dbd3e7b9fe6b6aebdfcb4077b9b0c6351e94e

u = 0x8a17046e6e2417994139c5b57fb1f8700062fb67d435b5ddfcf4a9d44f6c52fceb
    6eb10372486c1c9d01587ad776d285e6b02cdda1d5a80993b6f6d2fc356ac8

u' = 0x7e150711098af13547d25ab9f85615a892faa3842778d8442729dd00cf72687a2
     b86af2de61622ebae0823a03656501a01370da1cef809c9809ef2b749c09e0e

h~ = 0x31c724131f8fc689de7a23e51320d265321b1f33db2e161b75f35b66e63064115
     648a39c8b28345a3be4290bde2a9d93d6c87ca01f455e1912de76fd5672c755

EU = 0x8a17046e6e2417994139c5b57fb1f8700062fb67d435b5ddfcf4a9d44f6c52fce
     b6eb10372486c1c9d01587ad776d285e6b02cdda1d5a80993b6f6d2fc356ac8

EU' = 0x7e150711098af13547d25ab9f85615a892faa3842778d8442729dd00cf72687a
      2b86af2de61622ebae0823a03656501a01370da1cef809c9809ef2b749c09e0e

alpha = 0x7265603f0ff462e1940a060c68dd864b16b9ce22

r' = 0xc2114e9865736183434568cd3526c4e00dcc2b52

v = 0x378c692bb3450c9a506348f345019053ef00afd2d436b0e2f435722ecadbf728a3
    adda54806d9d759618d5be331907276d87a051c8260e0357c9a0130a8d43e5

EV = 0x378c692bb3450c9a506348f345019053ef00afd2d436b0e2f435722ecadbf728a
     3adda54806d9d759618d5be331907276d87a051c8260e0357c9a0130a8d43e5

PEH = 0x31c724131f8fc689de7a23e51320d265321b1f33db2e161b75f35b66e6306411
      5648a39c8b28345a3be4290bde2a9d93d6c87ca01f455e1912de76fd5672c755

C0 = 0x8a17046e6e2417994139c5b57fb1f8700062fb67d435b5ddfcf4a9d44f6c52fce
     b6eb10372486c1c9d01587ad776d285e6b02cdda1d5a80993b6f6d2fc356ac87e15
     0711098af13547d25ab9f85615a892faa3842778d8442729dd00cf72687a2b86af2
     de61622ebae0823a03656501a01370da1cef809c9809ef2b749c09e0e378c692bb3
     450c9a506348f345019053ef00afd2d436b0e2f435722ecadbf728a3adda54806d9
     d759618d5be331907276d87a051c8260e0357c9a0130a8d43e5

K = 0x72c0f34359abf9cbeebb3e52cf1273d14066479a43ef9c93f9fd6f4080a5f27916
    98ab80c57d163192b51dc2efa27740d7625db9eb5cfeb6af370e85af5832a035facf
    2e2a150cb847338eb173438cdf7126162230917e258cc8a5eee6cb006ec5493ce69d
    c91fe3aa2c3c5792e19fea7eeec3bef3db66c4e0b4b36b08507f4e
```



 

### C.4.2 Test vector

```
-------------
ACE-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=128
CofactorMode=0

-----

Group=ECModp-Group:

p = 0xffffffffffffffffffffffffffffffffefffffffffffffff

a = 0xffffffffffffffffffffffffffffffffefffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key

g'(x) = 0x5a9d4f57936977adcade30ca2350d00096bab728d97499a8

g'(y) = 0xb521a9a56bac905bdf8673a9e83a25ded725bf7a53631b90

c(x) = 0x48dd5e86ac11435b355f9e42ddf6c4509d4d00ed4dc7eb83

c(y) = 0xc4f840332c46a887c58f7e0731ec0f4b11433ea220ee078f

d(x) = 0x603a3be96761734ec5a11096686ec2d252ce79ebc4b9dd5d

d(y) = 0x7aa5a1a995563856c3eb8b03e7c40157009f86e03793dd35

h(x) = 0x28437b3ff9b4371d4eeabf4ca150a5366eb8b950ab779072

h(y) = 0x6569c7ce2e2020768c9ee52e7100e46a06c81365821d2b13

-----

Private Key

w = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3a4

x = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8e44

y = 0xb9a4fa5c33ec1bfa66fa146b9514f3e4d2b023da873d4cbb

z = 0xd8b41a0eb3f5f88ce888aed452af12a8e096873e563a9203
```



 

```
------------------------------
Trace for ACE-KEM encrypt
------------------------------

Encoding format = uncompressed_fmt

r = 0x9658ad41da2d788ddec09a0265990ccbe903be34126c26a9

u(x) = 0xfd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb02944

u(y) = 0x07eb4a06d8c64b8032a60394736c4d645003bcf412516fdf

u'(x) = 0x83123745fa28135677da40c250bb4254bd0cba6a1c2e2585

u'(y) = 0x6bdf0ade4befa54a9ed1aa7cd9831383a8d17ed3498a19df

h~(x) = 0x456af30e1cbacbb6d069244aa8d1f191ff3ebacdcfaf539b

h~(y) = 0x3c9a22e32c801a9ec37d9e8d6b8a90e5a41ba007204cb4ff

EU = 0x04fd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb0294407eb4a06d8c64b8
     032a60394736c4d645003bcf412516fdf

EU' = 0x0483123745fa28135677da40c250bb4254bd0cba6a1c2e25856bdf0ade4befa5
      4a9ed1aa7cd9831383a8d17ed3498a19df

alpha = 0xa1fd1f8238f51ea06ad52d55df7da4772f730e94

r' = 0x716a5800d4de6612fcf75653538c5eb5571a83040f2d47a4

v(x) = 0x1544105c84f3765f8f1fd490b271a18b0ed1c45e6ecc5071

v(y) = 0xf44c386f466f43eaa29e0434395bb20a218d21715d15316c

EV = 0x041544105c84f3765f8f1fd490b271a18b0ed1c45e6ecc5071f44c386f466f43e
     aa29e0434395bb20a218d21715d15316c

PEH = 0x456af30e1cbacbb6d069244aa8d1f191ff3ebacdcfaf539b

C0 = 0x04fd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb0294407eb4a06d8c64b8
     032a60394736c4d645003bcf412516fdf0483123745fa28135677da40c250bb4254
     bd0cba6a1c2e25856bdf0ade4befa54a9ed1aa7cd9831383a8d17ed3498a19df041
     544105c84f3765f8f1fd490b271a18b0ed1c45e6ecc5071f44c386f466f43eaa29e
     0434395bb20a218d21715d15316c

K = 0x94a6b23344a026db8e3f2669562ad8fc06a529befb032d89a192a460d0340f5a7d
    533d79ce5ce59b5c778c2874f3330e03e02056b92d6ae1ad5d9749babe116620b168
    d77de156ab53b52b328b0b42c12ef7c74887805ee3fa82c0fb88e6e27ef65e669fa9
    43844124c9d5de423d08766dbfa44686fbb5d179239d9096520034
```

### C.4.3 Test vector

```
-------------
ACE-KEM
-------------
```



 

```
Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=128
CofactorMode=0

-----

Group=ECModp-Group:

p = 0xfffffffffffffffffffffffffffffffeffffffffffffffff

a = 0xfffffffffffffffffffffffffffffffefffffffffffffffc

b = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

mu = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831

nu = 0x01

g(x) = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

g(y) = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811

-----

Public Key

g'(x) = 0x5a9d4f57936977adcade30ca2350d00096bab728d97499a8

g'(y) = 0xb521a9a56bac905bdf8673a9e83a25ded725bf7a53631b90

c(x) = 0x48dd5e86ac11435b355f9e42ddf6c4509d4d00ed4dc7eb83

c(y) = 0xc4f840332c46a887c58f7e0731ec0f4b11433ea220ee078f

d(x) = 0x603a3be96761734ec5a11096686ec2d252ce79ebc4b9dd5d

d(y) = 0x7aa5a1a995563856c3eb8b03e7c40157009f86e03793dd35

h(x) = 0x28437b3ff9b4371d4eeabf4ca150a5366eb8b950ab779072

h(y) = 0x6569c7ce2e2020768c9ee52e7100e46a06c81365821d2b13

-----

Private Key

w = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3a4

x = 0x083d4ac64f1960a9836a84f91ca211a185814fa43a2c8e44

y = 0xb9a4fa5c33ec1bfa66fa146b9514f3e4d2b023da873d4cbb

z = 0xd8b41a0eb3f5f88ce888aed452af12a8e096873e563a9203
```



 

```
------------------------------
Trace for ACE-KEM encrypt
------------------------------

Encoding format = compressed_fmt

r = 0x9658ad41da2d788ddec09a0265990ccbe903be34126c26a9

u(x) = 0xfd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb02944

u(y) = 0x07eb4a06d8c64b8032a60394736c4d645003bcf412516fdf

u'(x) = 0x83123745fa28135677da40c250bb4254bd0cba6a1c2e2585

u'(y) = 0x6bdf0ade4befa54a9ed1aa7cd9831383a8d17ed3498a19df

h~(x) = 0x456af30e1cbacbb6d069244aa8d1f191ff3ebacdcfaf539b

h~(y) = 0x3c9a22e32c801a9ec37d9e8d6b8a90e5a41ba007204cb4ff

EU = 0x03fd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb02944

EU' = 0x0383123745fa28135677da40c250bb4254bd0cba6a1c2e2585

alpha = 0xf3af4f830f0cdb0f2c3dd05a2ceca58edb37c97f

r' = 0x8088d4e192dc432148f02aa124d31f0d0ea82c0ab3fb96ea

v(x) = 0x7f0963883bed2203445a315a3d5ca1bb68d3ec74ede13f4f

v(y) = 0x37a45b48bde10a956a0f19fbdf9b2796d33c2be5330b7cf9

EV = 0x037f0963883bed2203445a315a3d5ca1bb68d3ec74ede13f4f

PEH = 0x456af30e1cbacbb6d069244aa8d1f191ff3ebacdcfaf539b

C0 = 0x03fd5dd4aa91d2c67b57bfd32f103e5432605f8b903fb029440383123745fa281
     35677da40c250bb4254bd0cba6a1c2e2585037f0963883bed2203445a315a3d5ca1
     bb68d3ec74ede13f4f

K = 0xd29e265d98f2b3051f2f516ac3cbb96852bec0518bc82ba8660bc5d406a4c82fcd
    dc311d935f847963f7a8ea8c0e661109d4bb18306d868aa2a70fcade78d51b0a9468
    b309a59ca8d33774caf4966adc156a27243d2added6ee47551eb26f0b9c68c0715e5
    d8751ba4ec02e959bbb8b3278468228d2695156ae59f01eca85b58
```

### C.4.4 Test vector

```
-------------
ACE-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=128
CofactorMode=0

-----
```



 

```
Group=ECGF2-Group:

p = 0x080000000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd

mu = 0x04000000000000000000292fe77e70c12a4234c33

nu = 0x01

g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1
-----

Public Key

g'(x) = 0x052248912facadbe4995dc17e15c2760dca33bef9c

g'(y) = 0x0132e6b3cdf5a6fc94af4bcff2320c1e673e2897df

c(x) = 0x0537639a8b5c088e9c4960986961fc0e7c531df742

c(y) = 0x0733205990c58c743f14aed5550fa5f9a44af020e7

d(x) = 0x013344cd624a8d3af7b38fc6103d795792d951d2a6

d(y) = 0xb47079579331c06ae15065d4cf0b436a20c77f6e

h(x) = 0x059adc6998e2b481aa7d65739ae772187fcc94a933

h(y) = 0x03294c9d5168906f47fe504d5121542a8962fa945b
-----

Private Key

w = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c902

x = 0xa9836a84a1583f601a2f9b2b2432a0aff42c84e8

y = 0x02140a3d998770496c5cbec836b6e8d38e47cc0575

z = 0x02f179878e0f7ef84d45966f119bc634d0f246beec

--------------------------------
Trace for ACE-KEM encrypt
--------------------------------

Encoding format = uncompressed_fmt

r = 0x015897ecb2c932fa1bb876e25442682b342fab391c

u(x) = 0x05cf2e1de9dcf32160bef47df954851b52a226f463

u(y) = 0x06c65878cff713a57fa53bbfc87497ac73067ed3aa
```



 

```
u'(x) = 0x04783f61a7493d83d76b8178c0935a1830b8708ea8

u'(y) = 0x02aa698207027836dd768207089af0ee1b556aa9d3

h~(x) = 0x0b420ea755ce20f5fa8ea1015d0d2cbf5860767f

h~(y) = 0x055fe3d3d923afdb92c3e44a1e9ae34c249b7f3eb1

EU = 0x0405cf2e1de9dcf32160bef47df954851b52a226f46306c65878cff713a57fa53
     bbfc87497ac73067ed3aa

EU' = 0x0404783f61a7493d83d76b8178c0935a1830b8708ea802aa698207027836dd76
      8207089af0ee1b556aa9d3

alpha = 0x4a159752a3b5fad5725dce4b7a626e93021de7d5

r' = 0x8aeed29f26765252b9b6fa8e7419c3db8b2766aa

v(x) = 0x01452f7abbd59e15c528aa67738c03829a4facb9d3

v(y) = 0x0374bb51467dc126d5af50e6360f29b8a1427d01c9

EV = 0x0401452f7abbd59e15c528aa67738c03829a4facb9d30374bb51467dc126d5af5
     0e6360f29b8a1427d01c9

PEH = 0x000b420ea755ce20f5fa8ea1015d0d2cbf5860767f

C0 = 0x0405cf2e1de9dcf32160bef47df954851b52a226f46306c65878cff713a57fa53
     bbfc87497ac73067ed3aa0404783f61a7493d83d76b8178c0935a1830b8708ea802
     aa698207027836dd768207089af0ee1b556aa9d30401452f7abbd59e15c528aa677
     38c03829a4facb9d30374bb51467dc126d5af50e6360f29b8a1427d01c9

K = 0x472984597505cf1aec33eeb7477b7546ab14490e65106fce3842a55adbc6aa9828
    e0be5b74785fdf3583023352961ae5d49827a61898e458e4b5b4571472ec6fa05558
    fe870d2954814d49b8560f0d02b039398a5bbd8742d37a463a4056488db1bae29b89
    c5a532e16a4ca8dcd3ab0a9d1fd4a1c42ab27c031a81dc1e53b9ba
```

### C.4.5 Test vector

```
-------------
ACE-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=128
CofactorMode=0

-----

Group=ECGF2-Group:

p = 0x080000000000000000000000000000000000000c9

a = 0x01

b = 0x020a601907b8c953ca1481eb10512f78744a3205fd
```



 

```
mu = 0x040000000000000000000292fe77e70c12a4234c33

nu = 0x01

g(x) = 0x03f0eba16286a2d57ea0991168d4994637e8343e36

g(y) = 0xd51fbc6c71a0094fa2cdd545b11c5c0c797324f1

-----

Public Key

g'(x) = 0x052248912facadbe4995dc17e15c2760dca33bef9c

g'(y) = 0x0132e6b3cdf5a6fc94af4bcff2320c1e673e2897df

c(x) = 0x0537639a8b5c088e9c4960986961fc0e7c531df742

c(y) = 0x0733205990c58c743f14aed5550fa5f9a44af020e7

d(x) = 0x013344cd624a8d3af7b38fc6103d795792d951d2a6

d(y) = 0xb47079579331c06ae15065d4cf0b436a20c77f6e

h(x) = 0x059adc6998e2b481aa7d65739ae772187fcc94a933

h(y) = 0x03294c9d5168906f47fe504d5121542a8962fa945b

-----

Private Key

w = 0x028d2d26a73f713d3f9d0d5b8ce30d76f4d151c902

x = 0xa9836a84a1583f601a2f9b2b2432a0aff42c84e8

y = 0x02140a3d998770496c5cbec836b6e8d38e47cc0575

z = 0x02f179878e0f7ef84d45966f119bc634d0f246beec


---------------------------------
Trace for ACE-KEM encrypt
---------------------------------

Encoding format = compressed_fmt

r = 0x015897ecb2c932fa1bb876e25442682b342fab391c

u(x) = 0x05cf2e1de9dcf32160bef47df954851b52a226f463

u(y) = 0x06c65878cff713a57fa53bbfc87497ac73067ed3aa

u'(x) = 0x04783f61a7493d83d76b8178c0935a1830b8708ea8

u'(y) = 0x02aa698207027836dd768207089af0ee1b556aa9d3

h~(x) = 0x0b420ea755ce20f5fa8ea1015d0d2cbf5860767f
```



 