DRAFT
ISO/IEC 18033-2: Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric Ciphers

Editor: Victor Shoup

January 15, 2004

Editor's note: The following items still need to be addressed:

- **The editor needs to convert to ISO format.**
- **Someone needs to take a close look at the ASN1 syntax.**
- **A final decision needs to be reached as to which schemes are included in the standard.**

Contents

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75% of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this part of ISO/IEC 18033 may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 18033-2 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 27, *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*

Annex A of this part of ISO/IEC 18033 is for information only. Annex B forms a normative part of this part if ISO/IEC 18033. Annex C of this part of ISO/IEC 18033 is for information only.

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 ciphertext format for several specific asymmetric ciphers (although conforming systems may choose to use alternative formats for storing and transmitting ciphertexts).

A normative annex (Annex B) 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 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;
- *EPOC-2*, *HIME(R)*: two schemes based on the hardness of factoring.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions for this part of ISO/IEC 18033. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. For undated references, the latest edition of the normative document referred to applies. Members of IEC and ISO maintain registers of currently valid International Standards.

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

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

ISO/IEC 10116, *Information technology — Security techniques — Modes of operation for an n-bit block cipher.*

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

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

ISO/IEC 11770, *Information technology — Security techniques — Key management.*

ISO/IEC 15946-1, *Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 1: General.*

ISO/IEC 18031, *Information technology — Security techniques — Random bit generation.*

ISO/IEC 18032, *Information technology — Security techniques — Prime number generation.*

ISO/IEC 18033-1, *Information technology — Security techniques — Encryption algorithms — Part 1: General.*

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

3 Definitions

For the purposes of this part of ISO/IEC 18033, the following definitions apply; where appropriate, forward references are given to clauses which contain more detailed definitions and/or further elaboration.

3.1 asymmetric cipher: cipher based on asymmetric cryptographic techniques whose public transformation is used for encryption and whose private transformation is used for decryption [ISO/IEC 18033-1]. *(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 18033-1].

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 18033-1]. *(See Clauses 7, 8.1.)*

3.4 bit: one of the two symbols `‘0’` or `‘1’`. *(See Clause 5.2.1.)*

3.5 bit string: an ordered sequence of bits. *(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 plaintext, i.e., a string of bits of a defined length, to yield a block of ciphertext [ISO/IEC 18033-1]. *(See Clause 6.4.)*

Note. In this part of ISO/IEC 18033, plaintext/ciphertext 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 ciphertext: data which has been transformed to hide its information content [ISO/IEC 18033-1].

3.10 concrete group: an 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. *(See Clause 10.1.)*

3.11 cryptographic hash function: a function that maps octets 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. *(See Clause 6.1.)*

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

3.13 decryption: reversal of the corresponding encryption [ISO/IEC 18033-1].

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 ciphertext, i.e., to hide the information content of the data [ISO/IEC 18033-1].

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. *(See Clause 5.3.)*

3.17 encryption algorithm: process which transforms plaintext into ciphertext [ISO/IEC 18033-1].

3.18 encryption option: an 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 ciphertext. *(See Clauses 7, 8.1.)*

3.19 field: the 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: a group such that the underlying set of elements is finite, and such that the underlying binary operation is commutative.

3.21 finite field: a field such that the underlying set of elements is finite.

3.22 group: the 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: an asymmetric cipher that combines both asymmetric and symmetric cryptographic techniques.

3.24 key: a sequence of symbols that controls the operation of a cryptographic transformation (e.g., encryption, decryption) [ISO/IEC 18033-1].

3.25 key derivation function: a function that maps octets 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. *(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. *(See Clause 8.1.)*

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

3.28 label: an 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 ciphertext in a non-malleable way. *(See Clauses 7, 8.2.)*

3.29 length: (1) The *length of a bit string* is the number of bits in the string. *(See Clause 5.2.1.)* (2) The *length of an octet string* is the number of octets in the string. *(See Clause 5.2.2.)* (3) The *length in bits of a non-negative integer* n is the number of bits in its binary representation, i.e., $\lceil \log_2(n+1) \rceil$. *(See Clause 5.2.4.)* (4) The *length in octets of a non-negative integer* n is the number of digits in its representation base 256, i.e., $\lceil \log_{256}(n+1) \rceil$. *(See Clause 5.2.5.)*

3.30 message authentication code (MAC): the string of bits which is the output of a MAC algorithm [ISO/IEC 9797-1]. *(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: an 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].

(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. *(See Clause 5.2.2.)*

3.33 octet string: An ordered sequence of *octets*. *(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 18033-1].

3.35 prefix free set: a set S of bit/octet strings such that there do not exist strings $x, y \in S$ 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 18033-1]. *(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 18033-1]. *(See Clauses 7, 8.1.)*

3.39 secret key: key used with symmetric cryptographic techniques by a specified set of entities [ISO/IEC 18033-1].

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

Throughout this part of ISO/IEC 18033, the following symbols and notation are used; 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 $\lceil -5.3 \rceil = -5$.
$[a \mathinner{..} b]$	the interval of integers from a to b, including both a and b.
$[a \mathinner{..} b)$	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 *(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. *(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 $x_1, \ldots, x_l$, in the given order. *(See Clauses 5.2.1, 5.2.2.)*
$x \parallel y$	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. *(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 $a = b = 0$).
$a \mid b$	a 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 $r \in [0 \mathinner{..} n)$ 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 $b \in [0 \mathinner{..} n)$ 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. *(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 may 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 an ordered 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 an ordered 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, while $\langle x, y \rangle$ 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 $e \ge 1$, explicit data for F consists of p and e, along with a "multiplication table," which is a matrix $T = (T_{ij})_{1 \le i,j \le e}$, where each T_{ij} is an e-tuple over $[0 \mathinner{..} p)$.

The set of elements of F is the set of all e-tuples over $[0 \mathinner{..} p)$. 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

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

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

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

then $d \cdot a = c$, where

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

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

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

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

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 θ_i denote the e-tuple over F' whose ith 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 [Len91].

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 15496-1, as well as [BSS99], 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 $y' = \mathit{FE2IP}_F(y)$, 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 $z' = \mathit{FE2IP}_F(z)$, and where f is the largest non-negative integer such that 2^f divides $\mathit{FE2IP}_F(1_F)$.

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 $\mathit{EC2OSP}_E$ and $\mathit{OS2ECP}_E$ 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 $\mathit{EC2OSP}_E(P, \mathit{fmt})$ 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 $\mathit{EP} = \langle\, \mathit{Oct}(0)\,\rangle$.
- If $P = (x, y) \neq \mathcal{O}$, with compressed form $(x, \tilde{y})$, then

$$\mathit{EP} = \langle\, H\,\rangle \,\|\, X \,\|\, Y,$$

  where

  - H is a single octet of the form $\mathit{Oct}(4U + C \cdot (2 + \tilde{y}))$, 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 $\mathit{FE2OSP}_F(x)$;
  - Y is the octet string $\mathit{FE2OSP}_F(y)$ 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 $\mathit{OS2ECP}_E(\mathit{EP})$ takes as input an octet string *EP*. If there exists a point P on the curve E and a format specifier *fmt* such that $\mathit{EC2OSP}_E(P, \mathit{fmt}) = \mathit{EP}$, 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 $\mathit{OS2ECP}_E(\mathit{EP})$ 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 only defined for inputs not exceeding a given length, a hash function in this part of ISO/IEC 18033 is defined to **fail** in this case.

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

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

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

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

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

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

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

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

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 A.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 9797-2, with the following provisos:

- For MAC the algorithms described in 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 9797-2 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 9797-2 are only defined for inputs not exceeding a given length, a MAC algorithm in this part of ISO/IEC 18033 is defined to **fail** in this case.

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 A.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 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 A.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

$\mathit{SC1.KeyLen} = \mathit{BC.KeyLen}$.

The function $\mathit{SC1.Encrypt}(k, M)$ works as follows.

1. Set $\mathit{padLen} = \mathit{BC.BlockLen} - (|M| \bmod \mathit{BC.BlockLen})$.
2. Let $P_1 = \mathit{Oct}(\mathit{padLen})$.
3. Let P_2 be the octet string formed by repeating the octet P_1 a total of padLen times (so $|P_2| = \mathit{padLen}$).
4. Let $M' = M \,\|\, P_2$.
5. Parse M' as $M'_1 \,\|\, \cdots \,\|\, M'_l$, where for $1 \le i \le l$, M'_i is an octet string of length $\mathit{BC.BlockLen}$.
6. Let c_0 be the octet string consisting of $\mathit{BC.BlockLen}$ copies of the octet $\mathit{Oct}(0)$, and for $1 \le i \le l$, let $c_i = \mathit{BC.Encrypt}(k, M'_i \oplus c_{i-1})$.
7. Let $c = c_1 \,\|\, \cdots \,\|\, c_l$
8. Output c.

The function $\mathit{SC1.Decrypt}(k, c)$ works as follows.

1. If $|c|$ is not a non-zero multiple of $\mathit{BC.BlockLen}$, then **fail**.
2. Parse c as $c = c_1 \,\|\, \cdots \,\|\, c_l$, where for $1 \le i \le l$, c_i is an octet string of length $\mathit{BC.BlockLen}$. Also, let c_0 be the octet string consisting of $\mathit{BC.BlockLen}$ copies of the octet $\mathit{Oct}(0)$.
3. For $1 \le i \le l$, let $M'_i = \mathit{BC.Decrypt}(k, c_i) \oplus c_{i-1}$.
4. Let P_1 be the last octet of M'_l, and let $\mathit{padLen} = \mathit{BS2IP}(P_1)$.
5. If $\mathit{padLen} \notin [1 \mathinner{..} \mathit{BC.BlockLen}]$, then **fail**.
6. Check that the last padLen octets of M'_l are equal to P_1; if not, then **fail**.
7. Let M''_l be the octet string consisting of the first $\mathit{BC.BlockLen} - \mathit{padLen}$ octets of M'_l.
8. Set $M = M'_1 \,\|\, \cdots \,\|\, M'_{l-1} \,\|\, M''_l$.
9. 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 $\mathit{SC2.KeyLen}$ is equal to the value of the system parameter KeyLen.

The function $\mathit{SC2.Encrypt}(k, M)$ works as follows.

1. Set $\mathit{mask} = \mathit{KDF}(k, |M|)$.
2. Set $c = \mathit{mask} \oplus M$.
3. Output c.

The function $\mathit{SC2.Decrypt}(k, c)$ works as follows.

1. Set $\mathit{mask} = \mathit{KDF}(k, |c|)$.
2. Set $M = \mathit{mask} \oplus c$.
3. Output M.

7 Asymmetric ciphers

An asymmetric cipher AC consists of three algorithms:

- A key generation algorithm $\mathit{AC.KeyGen}()$, that outputs a public-key/private-key pair (PK, pk). The structure of PK and pk depends on the particular cipher.
- An encryption algorithm $\mathit{AC.Encrypt}(PK, L, M, opt)$ 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 $\mathit{AC.Decrypt}(pk, L, C)$ 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 [RS91], and which is equivalent to a notion of "non-malleability" defined in [DDN00]). 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 A.6, 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 A.6, 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 may be made available to all parties who wish to send encrypted messages to the owner, while the private key should 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;
- an asymmetric cipher from the family *EPOC-2* of ciphers described in Clause 12.3.
- a bounded-plaintext-length asymmetric cipher from the family *HIME(R)* of ciphers described in Clause 13.3.

Note. As each of *HC*, *RSAES*, *EPOC-2*, 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 that for 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, 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, 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 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 A.7.

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,

$$\mathit{DEM.Decrypt}(K, L, \mathit{DEM.Encrypt}(K, L, M)) = M.$$

8.2.1 Degenerate types of data encapsulation mechanisms

Two 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 A.8.

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.

1. Compute $(K, C_0) = \mathit{KEM.Encrypt}(PK, \mathit{opt})$.
2. Compute $C_1 = \mathit{DEM.Encrypt}(K, L, M)$.
3. Set $C = C_0 \,\|\, C_1$.
4. 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.

1. 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.
2. Compute $K = \mathit{KEM.Decrypt}(pk, C_0)$.
3. Compute $M = \mathit{DEM.Decrypt}(K, L, C_1)$
4. Output M.

Note. The security of *HC* is discussed in Annex A.10. 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.

1. Parse K as $K = k \| k'$, where k and k' are octet strings such that $|k| = \mathit{SC.KeyLen}$ and $|k'| = \mathit{MA.KeyLen}$.
2. Compute $c = \mathit{SC.Encrypt}(k, M)$.
3. Let $T = c \| L \| \mathit{I2OSP}(8 \cdot |L|, 8)$.
4. Compute $MAC = MA.eval(k', T)$.
5. Set $C_1 = c \| MAC$.
6. 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.

1. Parse K as $K = k \| k'$, where k and k' are octet strings such that $|k| = \mathit{SC.KeyLen}$ and $|k'| = \mathit{MA.KeyLen}$.
2. If $|C_1| < MA.MACLen$, then **fail**.
3. Parse C_1 as $C_1 = c \| MAC$, where c and MAC are octet strings such that $|MAC| = MA.MACLen$.
4. Let $T = c \| L \| \mathit{I2OSP}(8 \cdot |L|, 8)$.
5. Compute $MAC' = MA.eval(k', T)$.
6. If $MAC \neq MAC'$, then **fail**.

7. Compute $M = SC.Decrypt(k, c)$.
8. Output M.

Note. A detailed discussion of the security of this construction is found in Annex A.9. 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.

1. Parse K as $K = k \,\|\, k'$, where k and k' are octet strings such that $|k| = SC.KeyLen$ and $|k'| = MA.KeyLen$.
2. Compute $c = SC.Encrypt(k, M)$.
3. Let $T = c \,\|\, L$.
4. Compute $MAC = MA.eval(k', T)$.
5. Set $C_1 = c \,\|\, MAC$.
6. 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.

1. Parse K as $K = k \,\|\, k'$, where k and k' are octet strings such that $|k| = \mathit{SC.KeyLen}$ and $|k'| = \mathit{MA.KeyLen}$.
2. If $|C_1| < \mathit{MA.MACLen}$, then **fail**.
3. Parse C_1 as $C_1 = c \,\|\, \mathit{MAC}$, where c and MAC are octet strings such that $|\mathit{MAC}| = \mathit{MA.MACLen}$.
4. Let $T = c \,\|\, L$.
5. Compute $\mathit{MAC}' = \mathit{MA.eval}(k', T)$.
6. If $\mathit{MAC} \neq \mathit{MAC}'$, then **fail**.
7. Compute $M = \mathit{SC.Decrypt}(k, c)$.
8. Output M.

Note 1. A detailed discussion of the security of this construction is found in Annex A.9. 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 $\mathit{DEM3.KeyLen} = \mathit{MsgLen} + \mathit{MA.KeyLen}$.

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.

1. Parse K as $K = k \,\|\, k'$, where k and k' are octet strings such that $|k| = \mathit{MsgLen}$ and $|k'| = \mathit{MA.KeyLen}$.
2. Compute $c = k \oplus M$.
3. Let $T = c \,\|\, L$.
4. Compute $MAC = \mathit{MA.eval}(k', T)$.
5. Set $C_1 = c \,\|\, MAC$.
6. 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.

1. Parse K as $K = k \,\|\, k'$, where k and k' are octet strings such that $|k| = \mathit{MsgLen}$ and $|k'| = \mathit{MA.KeyLen}$.
2. If $|C_1| \neq \mathit{MsgLen} + \mathit{MA.MACLen}$, then **fail**.
3. Parse C_1 as $C_1 = c \,\|\, MAC$, where c and MAC are octet strings such that $|c| = \mathit{MsgLen}$ and $|MAC| = \mathit{MA.MACLen}$.
4. Let $T = c \,\|\, L$.
5. Compute $MAC' = \mathit{MA.eval}(k', T)$.
6. If $MAC \neq MAC'$, then **fail**.
7. Compute $M = k \oplus c$.
8. Output M.

Note 1. A detailed discussion of the security of this construction is found in Annex A.9. 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 [ElG85].

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

  It is required that μ is prime. For some cryptographic schemes, it is further required that $\gcd(\mu, \nu) = 1$.
- $\mathcal{E}(\mathbf{a}, \mathit{fmt})$ 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 $\mathit{fmt}, \mathit{fmt}'$, we have $\mathcal{E}(\mathbf{0}, \mathit{fmt}) = \mathcal{E}(\mathbf{0}, \mathit{fmt}')$.

  - 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 $\mathit{fmt}, \mathit{fmt}'$, if $(\mathbf{a}, \mathit{fmt}) \neq (\mathbf{a}', \mathit{fmt}')$, and if either $\mathbf{a} \neq \mathbf{0}$ or $\mathbf{a}' \neq \mathbf{0}$, then $\mathcal{E}(\mathbf{a}, \mathit{fmt}) \neq \mathcal{E}(\mathbf{a}', \mathit{fmt}')$.

  An octet string x is called a *valid encoding* of a group element $\mathbf{a} \in \mathcal{H}$ if $x = \mathcal{E}(\mathbf{a}, \mathit{fmt})$ 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 $\mathcal{E}(\mathbf{a}, \mathit{fmt}) = x$ 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 μ 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 $\{\mathcal{E}(\mathbf{a}_1, \mathit{fmt}_1), \ldots, \mathcal{E}(\mathbf{a}_m, \mathit{fmt}_m)\}$ 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 $\mathit{fmt}_i = \mathit{fmt}_j$. 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 A.11.

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 μ, 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 $\mathit{FE2OSP}_F$, 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 $\mathit{OS2FEP}_F$, and **fails** if $\mathit{OS2FEP}_F$ **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 μ be the order of $\mathcal{G}$, and ν 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 $\mathit{EC2OSP}_E$ and $\mathit{OS2ECP}_E$. 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 $\mathit{FE2OSP}_F(0_F)$, and if $P = (x, y) \neq \mathcal{O}$, where $x, y \in F$, then the output is $\mathit{FE2OSP}_F(x)$. 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 [ABR99, ABR01].

10.2.1 System parameters

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

- Γ — 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 $\mathit{CheckMode} = 0$, 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.

1. Generate a random number $x \in [1 \mathinner{.\,.} \mu)$.
2. Compute $\mathbf{h} = x \cdot \mathbf{g}$.
3. Output the public key:
    - $\mathbf{h}$ — a non-zero element of $\mathcal{G}$.
4. Output the private key:
    - x — an integer in the set $[1 \mathinner{.\,.} \mu)$

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.

1. Generate a random number $r \in [1 \mathinner{.\,.} \mu)$.
2. If $\mathit{OldCofactorMode} = 1$, then set $r' = r \cdot \nu \bmod \mu$; otherwise, set $r' = r$.
3. Compute $\tilde{\mathbf{g}} = r \cdot \mathbf{g}$ and $\tilde{\mathbf{h}} = r' \cdot \mathbf{h}$.
4. Set $C_0 = \mathcal{E}(\tilde{\mathbf{g}}, \mathit{fmt})$.
5. If $\mathit{SingleHashMode} = 1$, then let Z be the null octet string; otherwise, let $Z = C_0$.
6. Set $\mathit{PEH} = \mathcal{E}'(\tilde{\mathbf{h}})$.
7. Set $K = \mathit{KDF}(Z \parallel \mathit{PEH}, \mathit{KeyLen})$.
8. 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 $x \in [1 \mathinner{\ldotp\ldotp} \mu)$, and a ciphertext C_0. It runs as follows.

1. Set $\tilde{\mathbf{g}} = \mathcal{D}(C_0)$; this step **fails** if C_0 is not a valid encoding of an element of $\mathcal{H}$.
2. If $\mathit{CheckMode} = 1$, test if $\tilde{\mathbf{g}} \in \mathcal{G}$; if not, then **fail**.
3. If $\mathit{CofactorMode} = 1$ or $\mathit{OldCofactorMode} = 1$, set $\hat{\mathbf{g}} = \nu \cdot \tilde{\mathbf{g}}$; otherwise, set $\hat{\mathbf{g}} = \tilde{\mathbf{g}}$.
4. If $\mathit{CofactorMode} = 1$, then set $\hat{x} = \nu^{-1}x \bmod \mu$; otherwise, set $\hat{x} = x$.
5. Compute $\tilde{\mathbf{h}} = \hat{x} \cdot \hat{\mathbf{g}}$.
6. If $\tilde{\mathbf{h}} = \mathbf{0}$, then **fail**.
7. If $\mathit{SingleHashMode} = 1$, then let Z be the null octet string; otherwise, let $Z = C_0$.
8. Set $\mathit{PEH} = \mathcal{E}'(\tilde{\mathbf{h}})$.
9. Set $K = \mathit{KDF}(Z \parallel \mathit{PEH}, \mathit{KeyLen})$.
10. Output the secret key K.

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

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

Note 3. When using $\mathit{SingleHashMode} = 1$, even if Γ 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 A.12.

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 [FO99].

10.3.1 System parameters

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

- Γ — 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.

1. Generate a random number $x \in [0 \mathinner{..} \mu)$.
2. Compute $\mathbf{h} = x \cdot \mathbf{g}$.
3. Output the public key:
    - $\mathbf{h}$ — an element of $\mathcal{G}$.
4. Output the private key:
    - x — an integer in the set $[0 \mathinner{..} \mu)$.

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.

1. Generate a random octet string *seed* of length *SeedLen*.
2. Compute
$$t = KDF(I0 \,\|\, seed, \lceil \log_{256} \mu \rceil + 16 + KeyLen),$$
an octet string of length $\lceil \log_{256} \mu \rceil + 16 + KeyLen$.
3. Parse t as $t = u \,\|\, K$, where u and K are octet strings such that $|u| = \lceil \log_{256} \mu \rceil + 16$ and $|K| = KeyLen$.
4. Compute $r = OS2IP(u) \bmod \mu$.

5. Compute $\tilde{\mathbf{g}} = r \cdot \mathbf{g}$ and $\tilde{\mathbf{h}} = r \cdot \mathbf{h}$.
6. Set $EG = \mathcal{E}(\tilde{\mathbf{g}}, fmt)$ and $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.
7. Set $SeedMask = KDF(I1 \,\|\, EG \,\|\, PEH, SeedLen)$.
8. Set $MaskedSeed = seed \oplus SeedMask$.
9. Set $C_0 = EG \,\|\, MaskedSeed$.
10. 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 $x \in [0 \mathinner{..} \mu)$, and a ciphertext C_0. It runs as follows.

1. Parse C_0 as $C_0 = EG \,\|\, MaskedSeed$, EG and $MaskedSeed$ are octet strings such that $|MaskedSeed| = SeedLen$; this step **fails** if $|C_0| < SeedLen$.
2. Set $\tilde{\mathbf{g}} = \mathcal{D}(EG)$; this step fails if EG is not a valid encoding of a group element.
3. Compute $\tilde{\mathbf{h}} = x \cdot \tilde{\mathbf{g}}$.
4. Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.
5. Set $SeedMask = KDF(I1 \,\|\, EG \,\|\, PEH, SeedLen)$.
6. Set $seed = MaskedSeed \oplus SeedMask$.
7. Compute
$$t = KDF(I0 \,\|\, seed, \lceil \log_{256} \mu \rceil + 16 + KeyLen),$$
an octet string of length $\lceil \log_{256} \mu \rceil + 16 + KeyLen$.
8. Parse t as $t = u \,\|\, K$, where u and K are octet strings such that $|u| = \lceil \log_{256} \mu \rceil + 16$ and $|K| = KeyLen$.
9. Compute $r = OS2IP(u) \bmod \mu$.
10. Compute $\bar{\mathbf{g}} = r \cdot \mathbf{g}$.
11. Test if $\bar{\mathbf{g}} = \tilde{\mathbf{g}}$; if not, then **fail**.
12. Output the secret key K.

Note. A discussion of the security of this scheme can be found in Annex A.13.

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 [CS98, CS01].

10.4.1 System parameters

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

- Γ — 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.

1. Generate random numbers $w, x, y, z \in [0 \mathinner{.\,.} \mu)$.
2. Compute the group elements
$$\mathbf{g}' = w \cdot \mathbf{g},\ \ \mathbf{c} = x \cdot \mathbf{g},\ \ \mathbf{d} = y \cdot \mathbf{g},\ \ \mathbf{h} = z \cdot \mathbf{g}.$$
3. Output the public key:
    - $\mathbf{g}', \mathbf{c}, \mathbf{d}, \mathbf{h}$ — elements of $\mathcal{G}$.
4. Output the private key:
    - w, x, y, z — integers in the set $[0 \mathinner{.\,.} \mu)$.

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.

1. Generate a random number $r \in [0 \mathinner{..} \mu)$.
2. Compute group elements
$$\mathbf{u} = r \cdot \mathbf{g},\ \mathbf{u}' = r \cdot \mathbf{g}',\ \tilde{\mathbf{h}} = r \cdot \mathbf{h}.$$
3. Compute the octet strings
$$EU = \mathcal{E}(\mathbf{u}, \mathit{fmt}),\ EU' = \mathcal{E}(\mathbf{u}', \mathit{fmt}).$$
4. Compute the integer
$$\alpha = \mathit{OS2IP}(\mathit{Hash.eval}(EU \parallel EU')).$$
5. Compute the integer
$$r' = \alpha \cdot r \bmod \mu.$$
6. Compute the group element
$$\mathbf{v} = r \cdot \mathbf{c} + r' \cdot \mathbf{d}.$$
7. Set $EV = \mathcal{E}(\mathbf{v}, \mathit{fmt})$.
8. Set $PEH = \mathcal{E}'(\tilde{\mathbf{h}})$.
9. Set $C_0 = EU \parallel EU' \parallel EV$.
10. Set $K = \mathit{KDF}(EU \parallel PEH, \mathit{KeyLen})$.
11. 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

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

and a ciphertext C_0. It runs as follows.

1. Parse C_0 as $C_0 = EU \parallel EU' \parallel EV$, 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.
2. Check that $\{EU, EU', EV\}$ is a consistent set of valid encodings; if not, then **fail**.

3. 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

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

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

5. Compute the integer

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

6. Compute the integer

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

7. Test if

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

   If not, then **fail**.

8. Compute the group element

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

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

10. Set $K = \mathit{KDF}(\mathit{EU} \,\|\, \mathit{PEH}, \mathit{KeyLen})$.

11. Output the secret key K.

For security reasons, it is recommended that an implementation reveal no information about the cause of the error in Step 7. 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 ν 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 1 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 2 may be omitted. Then the test in Step 7 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 $\mathcal{E}(w \cdot \hat{\mathbf{u}}, \mathit{fmt}) = \mathit{EU}'$ and $\mathcal{E}(t \cdot \hat{\mathbf{u}}, \mathit{fmt}) = \mathit{EV}$.

Note 3. A detailed discussion of the security of this scheme can be found in Annex A.14.

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 [RSA78].

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, 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, α, n) takes as input

- an octet string X,
- a positive integer α, and
- a positive integer n,

and outputs an octet string. It runs as follows:

1. Check if $|X| = \mathcal{L}(n)$; if not, then **fail**.
2. Set $x = \mathit{OS2IP}(X)$.
3. Check if $x < n$; if not, then **fail**.
4. Set $y = x^{\alpha} \bmod n$.
5. Set $Y = \mathit{I2OSP}(y, \mathcal{L}(n))$.
6. Output Y.

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

$$\mathit{RSATransform}(\mathit{RSATransform}(X, e, n), d, n) = X.$$

11.3 RSA encoding mechanisms

An RSA encoding mechanism *REM* specifies two algorithms:

- $\mathit{REM.Encode}(M, L, \mathit{ELen})$ 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.
- $\mathit{REM.Decode}(E, L)$ takes as input an octet string E and a label L. It attempts to find a plaintext M such that $\mathit{REM.Encode}(M, L, |E|) = E$. 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 $\mathit{REM.Encode}(M, L, \mathit{ELen})$ is invoked, the condition $|M| \leq \mathit{ELen} - \mathit{REM.Bound}$ 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 $\mathit{Oct}(0)$.

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 [BR94].

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

$$\mathit{REM1.Bound} = 2 \cdot \mathit{Hash.len} + 2.$$

11.3.2.2 Encoding function

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

1. Check that $|M| \le \mathit{ELen} - 2 \cdot \mathit{Hash.len} - 2$; if not, then **fail**.
2. Let *pad* be the octet string of length $\mathit{ELen} - |M| - 2 \cdot \mathit{Hash.len} - 2$ consisting of a sequence of $\mathit{Oct}(0)$ octets.
3. Generate a random octet string *seed* of length *Hash.len*.
4. Set $\mathit{check} = \mathit{Hash.eval}(L)$.
5. Set $\mathit{DataBlock} = \mathit{check} \,\|\, \mathit{pad} \,\|\, \langle\, \mathit{Oct}(1) \,\rangle \,\|\, M$.
6. Set $\mathit{DataBlockMask} = \mathit{KDF}(\mathit{seed}, \mathit{ELen} - \mathit{Hash.len} - 1)$.
7. Set $\mathit{MaskedDataBlock} = \mathit{DataBlockMask} \oplus \mathit{DataBlock}$.
8. Set $\mathit{SeedMask} = \mathit{KDF}(\mathit{MaskedDataBlock}, \mathit{Hash.len})$.
9. Set $\mathit{MaskedSeed} = \mathit{SeedMask} \oplus \mathit{seed}$.
10. Set $E = \langle\, \mathit{Oct}(0) \,\rangle \,\|\, \mathit{MaskedSeed} \,\|\, \mathit{MaskedDataBlock}$.
11. Output E.

11.3.2.3 Decoding function

The algorithm $\mathit{REM1.Decode}(E, L)$ runs as follows.

1. Let $\mathit{ELen} = |E|$.
2. Check if $\mathit{ELen} \ge 2 \cdot \mathit{Hash.len} + 2$; if not, then **fail**.
3. Set $\mathit{check} = \mathit{Hash.eval}(L)$.

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

5. Set $\mathit{SeedMask} = \mathit{KDF}(\mathit{MaskedDataBlock}, \mathit{Hash.len})$.

6. Set $\mathit{seed} = \mathit{MaskedSeed} \oplus \mathit{SeedMask}$.

7. Set $\mathit{DataBlockMask} = \mathit{KDF}(\mathit{seed}, \mathit{ELen} - \mathit{Hash.len} - 1)$.

8. Set $\mathit{DataBlock} = \mathit{MaskedDataBlock} \oplus \mathit{DataBlockMask}$.

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

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

11. If $\mathit{check}' \neq \mathit{check}$, $X \neq \mathit{Oct}(0)$, or $T \neq \mathit{Oct}(1)$, then **fail**.

12. Output M.

For security reasons, it is essential that an implementation reveal no information about the cause of the error in Step 11. 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 $\mathit{RSAKeyGen}()$ must always be greater than $\mathit{REM.Bound}$.

11.4.2 Key generation

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

1. Compute $(n, e, d) = \mathit{RSAKeyGen}()$.
2. Output the public key PK:
    - n — a positive integer.
    - e — a positive integer.
3. 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 $\mathit{RSAES.MaxMsgLen}(PK)$ is $\mathcal{L}(n) - \mathit{REM.Bound}$.

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 $\mathcal{L}(n) - \mathit{REM.Bound}$, and
- no encryption option.

It runs as follows:

1. Set $E = \mathit{REM.Encode}(M, L, \mathcal{L}(n))$.
2. Set $C = \mathit{RSATransform}(E, e, n)$.
3. 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:

1. Set $E = \mathit{RSATransform}(C, d, n)$; note that this step may **fail**.
2. Set $M = \mathit{REM.Decode}(E, L)$; note that this step may **fail**.
3. Output M.

Note. The security of *RSAES* is discussed in Annex A.16.

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:

1. Compute $(n, e, d) = \mathit{RSAKeyGen}()$.
2. Output the public key PK:
   - n — a positive integer.
   - e — a positive integer.
3. 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:

1. Generate a random number $r \in [0 \mathinner{\ldotp\ldotp} n)$.
2. Set $R = \mathit{I2OSP}(r, \mathcal{L}(n))$.
3. Set $C_0 = \mathit{RSATransform}(R, e, n)$.
4. Compute $K = \mathit{KDF}(R, \mathit{KeyLen})$.
5. 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:

1. Set $R = \mathit{RSATransform}(C_0, d, n)$; note that this step may **fail**.
2. Compute $K = \mathit{KDF}(R, \mathit{KeyLen})$.
3. Output the secret key K.

Note. The security of *RSA-KEM* is discussed in Annex A.17.

12 EPOC-based ciphers

This clause describes a family of ciphers based on the EPOC transform. The cipher *EPOC-2* is described in Clause 12.3.

Note 1. These schemes are based on the work of Okamoto and Uchiyama [OU98] and Fujisaki and Okamoto [FO99].

Note 2. While this cipher may be classified as a hybrid cipher, it is not a generic hybrid cipher, in the sense of Clause 8.

12.1 EPOC key generation algorithms

For a positive integer l, an l-bit EPOC key generation algorithm *EPOCKeyGen* is a probabilistic algorithm that takes no input, and outputs positive integers (p, q, n, g, h, w), where

- p is a prime, with $2^{l-1} \leq p < 2^l$,
- q is a prime, with $2^{l-1} \leq q < 2^l$, and $p \neq q$,
- $n = p^2 q$,
- $g \in [1 \mathinner{..} n)$ such that $\gcd(g, n) = 1$ and $g^{p-1} \not\equiv 1 \pmod{p^2}$,
- $h = g^n \bmod n$, and
- $w = ((g^{p-1} \bmod p^2) - 1)/p$.

The output distribution of an l-bit EPOC 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 EPOC-based ciphers, these schemes are parameterized in terms of *EPOCKeyGen*; i.e., *EPOCKeyGen* 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 EPOC encoding mechanisms

An EPOC encoding mechanism *EEM* specifies two algorithms:

- *EEM.Encode*(M, L, l, l') takes as input an octet strings M and L, and non-negative integers l and l', and outputs (f, r, C), where f and r are non-negative integers, and C is an octet string.

  This algorithm may **fail** if the lengths of M or L, or the values l or l', exceed some (large) implementation-defined bounds.

- *EEM.Decode*(C, L, f, l, l') takes as input octet strings C and L, and non-negative integers f, l, and l', and outputs (M, r'), where M is an octet string and r' is a non-negative integer.

  This algorithm may **fail** under some circumstances.

12.2.1 Allowable EPOC encoding mechanisms

The only EPOC encoding mechanism allowed in this part of ISO/IEC 18033 is *EEM1*, described in Clause 12.2.2.

12.2.2 *EEM1*

12.2.2.1 System parameters

EEM1 is a family of EPOC encoding mechanisms, parameterized by the following system parameters:

- *Hash* — a cryptographic hash function, as described in §6.1;
- *KDF* — a key derivation function, as described in §6.2;
- *KDF′* — a key derivation function, as described in §6.2;
- *StreamMode* — one of two values: 0 or 1;
- *SC* — a symmetric cipher, as described in §6.5 (only needed if $\mathit{StreamMode} = 0$).

12.2.2.2 Encoding function

The algorithm $\mathit{EEM1.Encode}(M, L, l, l')$ runs as follows:

```
1. Let $\mathit{seedLen} = \lfloor (l-1)/8 \rfloor$.
2. Let $\mathit{rlen} = \lceil l'/8 \rceil$.
3. Generate a random octet string seed of length seedLen.
4. If $\mathit{StreamMode} = 1$, then
     • Set $C = M \oplus \mathit{KDF}(\mathit{seed}, |M|)$;
   otherwise,
     • set $K = \mathit{KDF}(\mathit{seed}, \mathit{SC.KeyLen})$;
     • set $C = \mathit{SC.Encrypt}(K, M)$.
5. Set $\mathit{DB} = M \,\|\, \mathit{seed} \,\|\, C \,\|\, L$.
6. Set $H = \mathit{Hash.eval}(\mathit{DB})$.
7. Set $T = \mathit{KDF}'(H, \mathit{rLen})$.
8. Set $f = \mathit{OS2IP}(\mathit{seed})$.
9. Set $r = \mathit{OS2IP}(T) \bmod 2^{l'}$.
10. Output $(f, r, C)$.
```

12.2.2.3 Decoding function

The algorithm $\mathit{EEM1.Decode}(C, L, f, l, l')$ runs as follows:

1. Let $\mathit{seedLen} = \lfloor (l-1)/8 \rfloor$.
2. Let $\mathit{rlen} = \lceil l'/8 \rceil$.
3. Set $\mathit{seed} = \mathit{I2OSP}(f, \mathit{seedLen})$; note that this step may **fail**.
4. If $\mathit{StreamMode} = 1$, then
   - set $M = C \oplus \mathit{KDF}(\mathit{seed}, |C|)$;

   otherwise,
   - set $K = \mathit{KDF}(\mathit{seed}, \mathit{SC.KeyLen})$;
   - set $M = \mathit{SC.Decrypt}(K, C)$.
5. Set $\mathit{DB} = M \,\|\, \mathit{seed} \,\|\, C \,\|\, L$.
6. Set $H = \mathit{Hash.eval}(\mathit{DB})$.
7. Set $T = \mathit{KDF}'(H, \mathit{rLen})$.
8. Set $r' = \mathit{OS2IP}(T) \bmod 2^{l'}$.
9. Output (M, r').

12.3 *EPOC-2*

12.3.1 System parameters

EPOC-2 is a family of asymmetric ciphers, parameterized by the following system parameters:

- l — a positive integer;
- l' — a positive integer satisfying $l' \geq 2l + 1$;
- *EPOCKeyGen* — an l-bit EPOC key generation algorithm, as described in §12.1;
- *EEM* — an EPOC encoding mechanism, as described in §12.2.

12.3.2 Key generation

The algorithm *EPOC-2.KeyGen* takes no input, and runs as follows:

1. Compute $(p, q, n, g, h, w) = \mathit{EPOCKeyGen}()$.
2. Output the public key PK:
   - n, g, h — positive integers.
3. Output the private key pk:
   - n, g, h, p, q, w — positive integers.

12.3.3 Encryption

The algorithm *EPOC-2.Encrypt* takes as input

- a public key, consisting of positive integers n, g, h,
- a label L,
- a plaintext M, and
- no encryption option.

It runs as follows:

1. Set $(f, r, C_1) = \mathit{EEM.Encode}(M, L, l, l')$.
2. Set $c_0 = g^f h^r \bmod n$.
3. Set $C_0 = \mathit{I2OSP}(c_0, \mathcal{L}(n))$.
4. Set $C = C_0 \parallel C_1$.
5. Output C.

12.3.4 Decryption

The algorithm *EPOC-2.Decrypt* takes as input

- a private key, consisting of positive integers n, g, h, p, q, w,
- a label L, and
- a ciphertext C.

It runs as follows:

1. Parse C as $C = C_0 \parallel C_1$, where C_0 and C_1 are octet strings such that $|C_0| = \mathcal{L}(n)$; this step **fails** if $|C| < \mathcal{L}(n)$.
2. Set $c_0 = \mathit{OS2IP}(C_0)$.
3. Check that $c_0 < n$; if not, then **fail**.
4. Set $c_0' = c_0^{p-1} \bmod p^2$.
5. Check that $p \mid (c_0' - 1)$; if not, then **fail**.
6. Set $a = (c_0' - 1)/p$.
7. Set $f = aw^{-1} \bmod p$.

8. Set $(M, r') = \mathit{EEM.Decode}(C_1, L, f, l, l')$ (note: this step may **fail**).
9. Check that $c_0 \bmod q = g^f h^{r' \bmod (q-1)} \bmod q$; if not, then **fail**.
10. Output M.

For security reasons, it is essential that an implementation reveal no information that would allow an observer to distinguish between a failure at step 8 and at step 9.

Note. A discussion of the security of this scheme can be found in Annex A.18.

13 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 13.3.

13.1 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.

13.2 HIME encoding mechanisms

A HIME encoding mechanism *HEM* specifies two algorithms:

- $\mathit{HEM.Encode}(M, L, \mathit{ELen}, \mathit{KLen})$ 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 $1 \leq \mathit{KLen} \leq 8$. It outputs an octet string E of length ELen.
- $\mathit{HEM.Decode}(E, L, \mathit{KLen})$ takes as input an octet string E , a label L, and a positive integer KLen. It attempts to find a plaintext M such that $\mathit{HEM.Encode}(M, L, |E|, \mathit{KLen}) = E$. It returns M if such an M exists, and otherwise **fails**.

13.2.1 Allowable HIME encoding mechanisms

The only HIME encoding mechanism allowed in this part of ISO/IEC 18033 is *HEM1*, described below in Clause 13.2.2.

13.2.2 *HEM1*

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

Note. *HEM1* is based on the OAEP construction of Bellare and Rogaway [BR94].

13.2.2.1 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

$$\mathit{HEM1.Bound} = 2 \cdot \mathit{Hash.len} + 2.$$

13.2.2.2 Encoding function

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

1. Check that $|M| \leq \mathit{ELen} - 2 \cdot \mathit{Hash.len} - 2$; if not, then **fail**.
2. Let *pad* be the octet string of length $\mathit{ELen} - |M| - 2 \cdot \mathit{Hash.len} - 2$ consisting of a sequence of $\mathit{Oct}(0)$ octets.
3. Generate a random octet string *seed* of length $\mathit{Hash.len} + 1$.
4. Clear most significant *KLen*-bit of *seed*, and set $\mathit{seed} =$ the result.
5. Set $\mathit{check} = \mathit{Hash.eval}(L)$.
6. Set $\mathit{DataBlock} = \mathit{check} \,\|\, \mathit{pad} \,\|\, \langle\, \mathit{Oct}(1) \,\rangle \,\|\, M$.
7. Set $\mathit{DataBlockMask} = \mathit{KDF}(\mathit{seed}, \mathit{ELen} - \mathit{Hash.len} - 1)$.
8. Set $\mathit{MaskedDataBlock} = \mathit{DataBlockMask} \oplus \mathit{DataBlock}$.
9. Set $\mathit{SeedMask} = \mathit{KDF}(\mathit{MaskedDataBlock}, \mathit{Hash.len} + 1)$.
10. Clear most significant *KLen*-bit of *SeedMask*, and set $\mathit{SeedMask} =$ the result.
11. Set $\mathit{MaskedSeed} = \mathit{SeedMask} \oplus \mathit{seed}$.
12. Set $E = \mathit{MaskedSeed} \,\|\, \mathit{MaskedDataBlock}$.
13. Output E.

13.2.2.3 Decoding function

The algorithm $\mathit{HEM1.Decode}(E, L, \mathit{KLen})$ runs as follows.

1. Let $\mathit{ELen} = |E|$.
2. Set $\mathit{check} = \mathit{Hash.eval}(L)$.
3. Parse E as $E = \mathit{MaskedSeed} \,\|\, \mathit{MaskedDataBlock}$, where $\mathit{MaskedSeed}$ and $\mathit{MaskedDataBlock}$ are octet strings such that $|\mathit{MaskedSeed}| = \mathit{Hash.len} + 1$, and $|\mathit{MaskedDataBlock}| = \mathit{ELen} - \mathit{Hash.len} - 1$.
4. Set $\mathit{SeedMask} = \mathit{KDF}(\mathit{MaskedDataBlock}, \mathit{Hash.len} + 1)$.
5. Clear most significant KLen-bit of $\mathit{SeedMask}$, and set $\mathit{SeedMask} =$ the result.
6. Set $\mathit{seed} = \mathit{MaskedSeed} \oplus \mathit{SeedMask}$.
7. Set $\mathit{DataBlockMask} = \mathit{KDF}(\mathit{seed}, \mathit{ELen} - \mathit{Hash.len} - 1)$.
8. Set $\mathit{DataBlock} = \mathit{MaskedDataBlock} \oplus \mathit{DataBlockMask}$.
9. Parse $\mathit{DataBlock}$ as $\mathit{DataBlock} = \mathit{check}' \,\|\, M'$, where check' and M' are octet strings such that $|\mathit{check}'| = \mathit{Hash.len}$ and $|M'| = \mathit{ELen} - 2 \cdot \mathit{Hash.len} - 1$.
10. Let $M' = \langle M_1, M_2, \ldots, M_l \rangle$, where $M_1, M_2, \cdots, M_l$ are octets, and $l = \mathit{ELen} - 2 \cdot \mathit{Hash.len} - 1$; also, let m be the largest positive integer such that $m \leq l$ and $M_1 = M_2 = \cdots M_{m-1} = \mathit{Oct}(0)$, and let T denote the octet M_m and let M denote the octet string $\langle M_{m+1}, \ldots, M_l \rangle$.
11. If $\mathit{check}' \neq \mathit{check}$, most significant KLen-bit of $\mathit{seed} \neq$ bit string of 0, or $T \neq \mathit{Oct}(1)$, then **fail**.
12. Output M.

For security reasons, it is essential that an implementation reveal no information about the cause of the error in Step 11. In particular, an implementation should output the same error message at the same time, regardless of the cause of error.

13.3 *HIME(R)*

13.3.1 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 §13.1;
- *HEM* — a HIME encoding mechanism, as described in §13.2.

13.3.2 Key generation

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

1. Compute $(p, q, n) = \mathit{HIMEKeyGen}()$.
2. Output the public key PK:
   - n — a positive integer.
3. Output the private key pk:
   - n, p, q — positive integers.

13.3.3 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 $\mathcal{L}(n) - \mathit{HEM.Bound}$, and
- no encryption option.

It runs as follows:

1. Set $k = 8 \cdot \mathcal{L}(n) -$(bit length of n)$+1$.
2. Set $E = \mathit{HEM.Encode}(M, L, \mathcal{L}(n), k)$.
3. Set $e = \mathit{OS2IP}(E)$.
4. Set $c = e^2 \bmod n$.
5. Set $C = \mathit{I2OSP}(c, \mathcal{L}(n))$.
6. Output C.

13.3.4 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:

1. Set $c = \mathit{OS2IP}(C)$.
2. Set $k = 8 \cdot \mathcal{L}(n)-$(bit length of n)+1.
3. Set $z = p^{-1} \bmod q$.
4. Set $c_p = c \bmod p$, and $c_q = c \bmod q$.
5. Set $\alpha_1 = c_p^{\frac{p+1}{4}} \bmod p$, and $\alpha_2 = p - \alpha_1$.
6. Set $\beta_1 = c_q^{\frac{q+1}{4}} \bmod q$ and $\beta_2 = q - \beta_1$.
7. Set
    - 7.1. $u_0^{(1)} = \alpha_1$, and $u_1^{(1)} = (\beta_1 - u_0^{(1)})z \bmod q$.
    - 7.2. $u_0^{(2)} = \alpha_1$, and $u_1^{(2)} = (\beta_2 - u_0^{(2)})z \bmod q$.
    - 7.3. $u_0^{(3)} = \alpha_2$, and $u_1^{(3)} = (\beta_1 - u_0^{(3)})z \bmod q$.
    - 7.4. $u_0^{(4)} = \alpha_2$, and $u_1^{(4)} = (\beta_2 - u_0^{(4)})z \bmod q$.
8. For i from 1 to 4 do:
    - 8.1. Set $v_1^{(i)} = u_0^{(i)} + u_1^{(i)}p$.
    - 8.2. For t from 2 to d do:
        - 8.2.1. Set $u_t^{(i)} = \left((c - {v_{t-1}^{(i)}}^2 \bmod p^t q)/(p^{t-1}q)\right)(2u_0^{(i)})^{-1} \bmod p$.
        - 8.2.2. Set $v_t^{(i)} = v_{t-1}^{(i)} + u_t^{(i)}p^{t-1}q$.
    - 8.3. Set $x_i = u_0^{(i)} + u_1^{(i)}p + \sum_{t=2}^{d} u_t^{(i)}p^{t-1}q$.
9. For i from 1 to 4, set $X_i = \mathit{I2OSP}(x_i, \mathcal{L}(n))$.
10. If there exists a *unique* i such that $\mathit{HEM.Decode}(X_i, L, k)$ does not **fail**, and $x_i^2 \bmod n = c$, then, for such i, set $M = \mathit{HEM.Decode}(X_i, L, k)$, otherwise **fail**.
11. Output M.

Note. A discussion of the security of this scheme can be found in Annex A.19.

A Security considerations (informative annex)

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 take 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 [BKR94], 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.

Some of the proofs of security are in the so-called "random oracle" model, which was first formalized in [BR93], 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 [CGH98]. Nevertheless, a random oracle proof does rule out a broad class of attacks.

A.1 MAC algorithms

This section 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 9797-2, 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.

A.2 Block ciphers

This section 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.

A.3 Symmetric ciphers

This section 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.

A.4 Security of *SC1*

This section 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 [BDJR97], where it is shown to be secure against a "multiple plaintext" attack, as discussed above, assuming BC is a pseudo-random permutation (see Annex A.2). The cipher *SC1* uses a fixed initial value; nevertheless, it is easy to adapt the proof of security in [BDJR97] to show that *SC1* is secure against "single plaintext" attacks, which is adequate for the constructions in this document.

Note that the paper [Vau02] presents some attacks on *SC1*. However, the attacks in [Vau02] 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.

A.5 Security of *SC2*

This section 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.

A.6 Asymmetric ciphers

This section 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 [RS91]. Here, the definition in [RS91] 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 [DDN91, DDN98]). 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 [Can00, CS01, BDPR98, DDN91, DDN98] for more on notions of security for asymmetric ciphers.

See [NY90] 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 [GM84]. 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.

A.6.1 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.

A.6.2 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 section 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 $Adv'_{AC}(A)$.

A.7 Key encapsulation mechanisms

This section 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:

1. Run the encryption algorithm, generating a pair (K^*, C_0^*).
2. Generate a random octet string $\tilde{K}$ of length *KEM.KeyLen*.
3. Choose $b \in \{0, 1\}$ at random.

4. 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.

A.7.1 Benign malleability

This section 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 A.6.2.

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 $Adv'_{KEM}(A)$.

A.8 Data encapsulation mechanisms

This section 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.

A.9 Security of *DEM1*, *DEM2*, and *DEM3*

This section 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 A.3 and *MA* satisfies the definition of security in Annex A.1, then *DEM1* satisfies the definition of security in Annex A.8.

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, [CS01] 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 [CS01].

A.10 Security of *HC*

This section 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 A.7 and DEM satisfies the definition of security in Annex A.8, then HC satisfies the definition of security in Annex A.6.

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, [CS01] 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 [CS01]. If KEM is benignly malleable (see Annex A.7.1), then one can easily show that HC is also benignly malleable (see Annex A.6.2) with the same security bound as above.

A.11 Intractability assumptions related to concrete groups

This section 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.

A.11.1 The Computational Diffie-Hellman problem

The Computational Diffie-Hellman (CDH) problem for Γ is as follows. On input $(x\mathbf{g}, y\mathbf{g})$, where $x, y \in [0 \mathinner{..} \mu)$, 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 $[0 \mathinner{..} \mu)$.

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 [Sho97], it is shown how to take a $CDH_\Gamma[t, l]$-adversary A with $\epsilon = AdvCDH_\Gamma(A)$, and a given value of δ, 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.

A.11.2 The Decisional Diffie-Hellman problem

The Decisional Diffie-Hellman (DDH) problem for Γ 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 [Bon98, MW00, NR97] for further discussion of the DDH and related problems.

A.11.3 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 Γ. 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 [OP01] for more details about this assumption.

A.12 Security of *ECIES-KEM*

This section 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 Γ (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 $\mathit{SingleHashMode} = 0$ and

$$\mathit{CheckMode} + \mathit{CofactorMode} + \mathit{OldCofactorMode} > 0.$$

Then if A is a $\mathit{ECIES\text{-}KEM}[t, q]$-adversary that makes at most q' random oracle queries, then we have

$$\mathit{Adv}_{\mathit{ECIES\text{-}KEM}}(A) = O(\mathit{AdvGapCDH}_\Gamma(A')),$$

where

- A' is a $\mathit{GapCDH}_\Gamma[t', O(q')]$-adversary, where $t' \approx t$.

This bound is essentially proved in [CS01], at least for the case where $\mathit{CheckMode} = 1$ 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 $\mathit{SingleHashMode} = 1$ and

$$\mathit{CheckMode} + \mathit{CofactorMode} + \mathit{OldCofactorMode} > 0.$$

In this case, *ECIES-KEM* is no longer secure against adaptive chosen ciphertext attacks, but it is benignly malleable (see Annex A.7.1). If A is a $\mathit{ECIES\text{-}KEM}[t, q]$-adversary that makes at most q' random oracle queries, then we have

$$\mathit{Adv}'_{\mathit{ECIES\text{-}KEM}}(A) = O(\mathit{AdvGapCDH}_\Gamma(A')),$$

where

- A' is a $\mathit{GapCDH}_\Gamma[t', O(q \cdot q')]$-adversary, where $t' \approx t$.

Besides satisfying only a weaker definition of security, this reduction is not as tight as in the case where $\mathit{SingleHashMode} = 0$. Also, the quality of the reduction degrades even further with $\mathit{SingleHashMode} = 1$ when one considers the multi-plaintext model formally defined in [BBM00], whereas the reduction does not degrade significantly when $\mathit{SingleHashMode} = 0$.

If

$$\mathit{CheckMode} + \mathit{CofactorMode} + \mathit{OldCofactorMode} = 0,$$

then in both of the above estimates, the term

$$\mathit{AdvGapCDH}_\Gamma(A'),$$

must be replaced by

$$\nu \cdot \mathit{AdvGapCDH}_{\Gamma}(A').$$

Therefore, this selection of system parameters should only be used when ν 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 [ABR99, ABR01] for details.

A.13 Security of *PSEC-KEM*

This section 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 Γ (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

$$\mathit{Adv}_{\mathit{PSEC\text{-}KEM}}(A) = O(\mathit{AdvCDH}_{\Gamma}(A') + (q + q')(\mu^{-1} + 2^{-\mathit{SeedLen}})),$$

where A' is a $\mathit{AdvCDH}_{\Gamma}[t', O(q + q')]$-adversary, with $t' \approx t$.

This bound is proven in [Sho01b].

Also, the security does not significantly degrade in the multi-plaintext model formally defined in [BBM00].

A.14 Security of *ACE-KEM*

This section 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 Γ (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

$$\mathit{Adv}_{\mathit{ACE\text{-}KEM}}(A) = O(\ \mathit{AdvDDH}_{\Gamma}(A_1) + \mathit{Adv}_{\mathit{Hash}}(A_2) + \mathit{Adv}_{\mathit{KDF}}(A_3) + q \cdot \mu^{-1}\ ),$$

where:

- A_1, A_2, A_3 denote adversaries that run in time essentially the same as A.

- $\mathit{Adv}_{\mathit{Hash}}(A_2)$ denotes the probability that an adversary A_2, given encodings $\mathit{EU1}^*$ and $\mathit{EU2}^*$ of two independent, random elements in $\mathcal{G}$, can find encodings $\mathit{EU1}$ and $\mathit{EU2}$ of elements in $\mathcal{G}$, such that $(\mathit{EU1}, \mathit{EU2}) \neq (\mathit{EU1}^*, \mathit{EU2}^*)$, but

  $$\mathit{Hash.eval}(\mathit{EU1} \parallel \mathit{EU2}) = \mathit{Hash.eval}(\mathit{EU1}^* \parallel \mathit{EU2}^*).$$

  If the group supports multiple encodings, the adversary can choose the format it wants when $\mathit{EU1}^*$ and $\mathit{EU2}^*$ are generated; furthermore, the adversary may choose to use the same or different formats in its choice of $\mathit{EU1}$ and $\mathit{EU2}$; however, $\mathit{EU1}^*$ and $\mathit{EU2}^*$ must be consistent encodings, and the same holds for $\mathit{EU1}$ and $\mathit{EU2}$.

  If $\mathit{CofactorMode} = 1$, then the adversary may choose $\mathit{EU1}$ 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.

- $\mathit{Adv}_{\mathit{KDF}}(A_3)$ 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 $\mathit{EU1}$ be an encoding of $\mathbf{u}_1$. Let R be a random octet string of length KeyLen. The first distribution is $(R, \mathit{EU1})$, and the second is $(\mathit{KDF}(\mathit{EU1} \parallel \mathcal{E}'(\tilde{\mathbf{h}}), \mathit{KeyLen}), \mathit{EU1})$.

The reader is referred to [CS01] for a detailed proof for the case where $\mathit{CofactorMode} = 0$ 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 [CS01] that *ACE-KEM* is no less secure than *ECIES-KEM*, in the sense that for any $\mathit{ACE\text{-}KEM}[t, q]$-adversary A, there exists a $\mathit{ECIES\text{-}KEM}[t', q]$-adversary A' such that $t' \approx t$ and $\mathit{Adv}_{\mathit{ECIES\text{-}KEM}}(A') \approx \mathit{Adv}_{\mathit{ACE\text{-}KEM}}(A)$. The proof in [CS01] is only for the case where $\mathit{CofactorMode} = 0$ 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 [CS01] 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 [CS01] is only for the case where $\mathit{CofactorMode} = 0$ 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 7. 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*.

A.15 The RSA inversion problem

This section 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 $\mathit{RSAKeyGen}()$, along with random $x \in [0 \mathinner{..} n)$, compute y such that $y^e \equiv x \pmod{n}$. For any given algorithm A and any given RSA key generation algorithm $\mathit{RSAKeyGen}$, $\mathit{Adv}_{\mathit{RSAKeyGen}}(A)$ 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 $\mathit{RSAKeyGen}[t]$-*adversary.*

The RSA assumption for $\mathit{RSAKeyGen}$ is the assumption $\mathit{Adv}_{\mathit{RSAKeyGen}}(A)$ is negligible for any efficient algorithm A.

A.16 Security of *RSAES*

This section discusses the security of the bounded-plaintext-length asymmetric cipher *RSAES*, defined in Clause 11.4.

The paper [BR94] 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 [BR94] that the resulting scheme satisfies a technical property called "plaintext awareness," assuming the underlying permutation is indeed one way. However, as pointed out in [Sho01a], 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 A.6). Moreover, it is proven in [Sho01a] 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 [BR94] does not establish this.

In [Sho01a], it is shown that *RSAES* is secure if the encryption exponent e is very small (e.g., $e = 3$). This result was generalized in [FOPS01] to general encryption exponent. It should be pointed out, however, that the security reduction in [FOPS01] 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 [Sho01a] for small encryption exponent 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 11; if an attacker can obtain such information from a decryption oracle, then the scheme can be easily broken, as described in [Man01].

A.17 Security of *RSA-KEM*

This section 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 $\mathit{RSAKeyGen}$, such that the output (n, e, d) always satisfies $n \geq \mathit{nBound}$, and for any $\mathit{RSA\text{-}KEM}[t, q]$-adversary A, we have

$$\mathit{Adv}_{\mathit{RSA\text{-}KEM}}(A) \leq \mathit{Adv}_{\mathit{RSAKeyGen}}(A') + q/\mathit{nBound},$$

where

- A' is a $\mathit{RSAKeyGen}[t']$-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 [Sho01b].

This security reduction is quite tight, unlike those for *RSAES* discussed above in Annex A.16. Moreover, in the multi-plaintext model formally defined in [BBM00], 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 [Man01].

A.18 Security of *EPOC-2*

It can be shown that in the random oracle model, where the functions *KDF*, KDF', and *Hash* used in *EEM1* are viewed as random oracles, that *EPOC-2* is secure against adaptive chosen ciphertext attack, assuming that

- factoring numbers of the form $n = p^2q$, as output by *EPOCKeyGen*, is computationally infeasible, and
- the symmetric cipher *SC* used in *EEM1* is secure in the sense defined in Annex A.3.

The proof of security can be derived from applying the general results of Fujisaki and Okamoto [FO99] to the asymmetric cipher defined in Okamoto and Uchiyama [OU98]. The security reduction can in fact be shown to be significantly tighter than that implied by the general security proof in [FO99], as will be discussed in a forthcoming version of [FO99].

As pointed out in Clause 12.3.4, care must be taken in the implementation of the decryption algorithm not to reveal the cause of certain errors; otherwise, the scheme is susceptible to a *key recovery* attack, as described in [Den02].

A.19 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 [NSS01, HIM02] — note that [HIM02] corrects several mistakes in [NSS01].

B ASN1 Syntax for Object Identifiers (normative annex)

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.

[Editor's note: This is still a "rough draft," and really needs to be closely looked at; also, the "version history" is included just for time being, for information only. Please contact me at `shoup@cs.nyu.edu` for the original ASCII source file.]

```
-- Version history

-- version 1, Karol Gorski
--      first draft, KG

-- version 1.1, Phil Griffin
--      corrections to ASN.1 syntax and extensions to specify supported
--  algorithms

-- version 2, Karol Gorski
--      changed order of some definition concerning epoc2 and eem
--      changed hash function oids
--      added block cipher oids
--      removed the noIv definition
--      added prime order field specification
--      changed explicit finite field specification to a choice

-- version 2.1, Phil Griffin
--      corrections to ASN.1 syntax and comments
--
--      Notice that "rsaes", "epoc2" and "genericHybrid" are defined
--      but are not referenced by any information object or in any
--      information object set.
--
--      Added context specific tag to eliminate a duplicate tag
--      in type EciesKemParameters. Here the "keyDerivationFunction"
--      and optional "group" component shared the same tag.
--
--      Added a context specific tag on the "groupParameters" component
--      of type "Group", as this optional component references a CHOICE
--      type which must be disabiguated with extra tagging. I also needed
--      to add one tag to these two choice alternatives. For reasons of
--      style, I chose to add tags to both choice alternatives as I think
--      this makes life easier for hand coders of this protocol.
--
--      Added context specific tag to eliminate a duplicate tag
--      in type PsecKemParameters. Here the "keyDerivationFunction"
--      and optional "group" component shared the same tag.
--
--      Added context specific tag to eliminate a duplicate tag
--      in type AceKemParameters. Here the "keyDerivationFunction"
--      and optional "group" component shared the same tag.
--
-- version 3, Karol Gorski
--      added HIME-R specifications
--      added public key specifications for all algorithms
--      changed some definitions to align with other standards
--      changed definition of Group to a CHOICE
--      added optional hash code length parameter to hash functions
--      included X9.62 syntax for finite fields and elliptic curve
--              parameters
--
-- version 4, Karol Gorski
--  removed the subjectPublicKeyInfo definition,
--  added normative comment concerning the encoding of public keys in
--     the subjectPublicKeyInfo structure
--  adopted the X9.62 style of definitions of finite fields and
--     their representations
--  added a general irreducible polynomial basis type for
--     characteristic two finite field
--  added an odd characteristic field type
--  added tags to KemPublicKey structure (a CHOICE) to facilitate
--     decoding
--  cleaned up old comments
--  checked syntax using OSS ASN.1 tools
--
-- version 4a, Karol Grski
--  added normative comment to explain choice of optimal normal
--     bases
--  generalised encoding of odd characteristic fields to permit
--     future extensions to bases other than polynomial
--  reformatted to not more than 70 characters per line


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

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

-- EXPORTS All; --

-- IMPORTS None; --


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

-- 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-eem                  OID ::= {is18033-2 epoc-encoding-method(7)}
id-hem                  OID ::= {is18033-2 himer-encoding-method(8)}
id-ft                   OID ::= {is18033-2 field-type(9)}

-- Asymmetric ciphers --

id-ac-rsaes             OID ::= { id-ac rsaes(1) }
id-ac-generic-hybrid    OID ::= { id-ac generic-hybrid(2) }
id-ac-epoc2             OID ::= { id-ac epoc2(3) }
id-ac-himer             OID ::= {id-ac himer(4) }

-- 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 psec(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) }

-- epoc encoding methods --
```

```
id-eem-eem1             OID ::= { id-eem eem1(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 DER-encoded value
-- 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-ac-rsaes PARMS RsaesParameters
}

RsaesPublicKey ::= RSAPublicKey

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

RsaesParameters ::= RsaEncodingMethod

RsaEncodingMethod ::= AlgorithmIdentifier {{ RSAemAlgorithms }}


RSAemAlgorithms ALGORITHM ::= {
        { OID id-rem-rem1 PARMS Rem1Parameters },

        ...  -- Expect additional algorithms --
}

Rem1Parameters ::= SEQUENCE {
        hashFunction            HashFunction,
        keyDerivationFunction   KeyDerivationFunction
}


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

-- EPOC-2 asymmetric cipher

epoc2 ALGORITHM ::= {
        OID id-ac-epoc2 PARMS Epoc2Parameters
}

Epoc2PublicKey ::= SEQUENCE {
        n       INTEGER,
        g       INTEGER,
        h       INTEGER
}

Epoc2Parameters::=SEQUENCE{
        l               INTEGER,
        lPrime          INTEGER,
        encodingMethod  Epoc2EncodingMethod
}

Epoc2EncodingMethod ::= AlgorithmIdentifier {{ EemAlgorithms }}

EemAlgorithms ALGORITHM ::= {
        { OID id-eem-eem1 PARMS Eem1Parameters },

        ...  -- Expect additional algorithms --
}

Eem1Parameters ::= SEQUENCE {
        hashFunction            HashFunction,
        keyDerivationFunction   KeyDerivationFunction,
        keyDerivationFunction2  KeyDerivationFunction,
        symmetricCipher         SymmetricCipher,
        streamMode              BOOLEAN
}

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

-- 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
}

GenericHybridPublicKey ::= KemPublicKey

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

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

```
-- 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 --
}

KemPublicKey ::= CHOICE {
        eciesKemPublicKey       [0] EciesKemPublicKey,
        psecKemPublicKey        [1]     PsecKemPublicKey,
        aceKemPublicKey         [2] AceKemPublicKey,
        rsaKemPublicKey         [3] RsaKemPublicKey,

        ... -- expect additional choices
}

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

-- 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)
```

```
Ripemd128HashLength     ::= INTEGER (1..128)
Ripemd160HashLength     ::= INTEGER (1..160)
Sha1HashLength          ::= INTEGER (1..160)
Sha256HashLength        ::= INTEGER (1..256)
Sha384HashLength        ::= INTEGER (1..384)
Sha512HashLength        ::= INTEGER (1..512)
WhirlpoolHashLength     ::= INTEGER (1..512)


MacAlgorithm ::= AlgorithmIdentifier {{ MACAlgorithms }}

MACAlgorithms ALGORITHM ::= {

        ...  -- Expect additional algorithms --
}

HashFunction ::= AlgorithmIdentifier {{ HashAlgorithms }}


-- the ISO hash functions may be parameterised with the length of
-- the hash code
HashAlgorithms ALGORITHM ::= {
        -- iso identifiers
        { OID id-dhf-ripemd128  PARMS Ripemd128HashLength }             |
        { OID id-dhf-ripemd160  PARMS Ripemd160HashLength }             |
        { OID id-dhf-sha1       PARMS Sha1HashLength }                  |
        { OID id-dhf-sha256     PARMS Sha256HashLength }                |
        { OID id-dhf-sha384     PARMS Sha384HashLength }                |
        { OID id-dhf-sha512     PARMS Sha512HashLength }                |
        { OID id-dhf-whirlpool  PARMS WhirlpoolHashLength }             |

        -- nist identifiers
        { OID id-sha1           }  |
        { OID id-sha256         }  |
        { OID id-sha384         }  |
        { OID id-sha512         }  ,

        ...  -- 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 }}

BlockAlgorithms ALGORITHM ::= {
        { OID id-bc64-misty1    }               |
        { OID id-bc64-tdea      }               |
        { OID id-bc128-aes      }               |
        { OID id-bc128-camellia }               |
        { OID id-bc128-seed     }               |
        { OID id-bc128-rc6      }               ,

        ...  -- Expect additional algorithms --
}

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

-- Useful external object identifiers --

-- hash functions

-- ISO identifiers
is10118-3               OID ::= {iso(1) standard(0) is10118(10118) part3(3)}
id-dhf                  OID ::= {is10118-3 algorithm(0)}
id-dhf-ripemd160        OID ::= {id-dhf ripemd160(49)}
id-dhf-ripemd128        OID ::= {id-dhf ripemd128(50)}
id-dhf-sha1             OID ::= {id-dhf sha1(51)}
id-dhf-sha256           OID ::= {id-dhf sha256(52)}
id-dhf-sha512           OID ::= {id-dhf sha512(53)}
id-dhf-sha384           OID ::= {id-dhf sha384(54)}
id-dhf-whirlpool        OID ::= {id-dhf whirlpool(55)}

-- NIST identifiers
id-sha1         OID ::= { iso(1) identified-organization(3) oiw(14)
                        secsig(3) algorithms(2) 26 }
id-sha256       OID ::= { joint-iso-itu-t (2) country (16) us (840)
                        organization (1) gov (101) csor (3)
                        nistalgorithm (4) hashalgs (2) 1 }
id-sha384       OID ::= { joint-iso-itu-t (2) country (16) us (840)
                        organization (1) gov (101) csor (3)
                        nistalgorithm (4) hashalgs (2) 2 }
id-sha512       OID ::= { joint-iso-itu-t (2) country (16) us (840)
                        organization (1) gov (101) csor (3)
                        nistalgorithm (4) hashalgs (2) 3 }
```

```
-- block ciphers
is18033-3               OID ::= {iso(1) standard(0) is18033(18033) part3(3)}
-- need to change the following 2 identifiers in 18033-3
id-bc64                 OID ::= {is18033-3 block-cipher-64-bit(1) }
id-bc128                OID ::= {is18033-3 block-cipher-128-bit(2) }
id-bc64-tdea            OID ::= {id-bc64 tdea(1) }
id-bc64-misty1          OID ::= {id-bc64 misty1(2) }
id-bc128-aes            OID ::= {id-bc128 aes(1) }
id-bc128-camellia       OID ::= {id-bc128 camellia(2) }
id-bc128-seed           OID ::= {id-bc128 seed(3) }
id-bc128-rc6            OID ::= {id-bc128 rc6(4) }


-- 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  -- AlgorithmObjectIdentifiers --
```

C Test Vectors (informative annex)

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 = 0x6434363064303334306635613764353336437396365356365353962356333737

k' = 0x3863323837346633333330653033653032303536

c = 0x0745c5f99ad56fe3ae4ebbeddc5385493cf67a8fa3e3fcdda5d8c82308a8e2b04c
    a4ac32241b1036f20fbe1f3aed19a3

T = 0x0745c5f99ad56fe3ae4ebbeddc5385493cf67a8fa3e3fcdda5d8c82308a8e2b04c
    a4ac32241b1036f20fbe1f3aed19a374657374000000000000000020

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
                          73206d61696e6c79206f6e20746865206c61696e

Label in ASCII = "test"

Label as octet string = 0x74657374

k = 0x6434363064303334306635613764353333643739636535636535396235633737

k’ = 0x3863323837346633333330653033653032303536

c = 0xae747466b1f160cf196d2ebe16ac9a70b6ff57c614436cf3de67ea38324f275791
    164cfcaea866b0024db7

T = 0xae747466b1f160cf196d2ebe16ac9a70b6ff57c614436cf3de67ea38324f275791
    164cfcaea866b0024db77465737400000000000000020

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 = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a920704560513898d9
    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 = 0xfffffffffffffffffffffffffffffffeffffffffffffffff

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 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 = 0xfffffffffffffffffffffffffffffffeffffffffffffffff

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 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 = 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 = 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 = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a920704560513898d9
    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 = 0xfffffffffffffffffffffffffffffffeffffffffffffffff

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 = 0xfffffffffffffffffffffffffffffffeffffffffffffffff

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 = 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 = 0x5e769d3a6fc9b82acf30800c8afe9631c2b9a1bdee398fd0a920704560513898d9
    4e40f3f6fc6a773249d63fc74bba14ceadc203b49f2344a6a22a0a8904c60b

mu = 0xdf0235fe94e74d2d70dbbc887389e5af9ec9ccd7

nu = 0x9e89f7f68e9a2e44b68affab0e53d03763d829685af48fa6405ce08865be6c7ee
     7221781300459df024b33e2

-----

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 = 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 = 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 = 0x0800000000000000000000000000000000000000c9

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 = 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 = 0x0800000000000000000000000000000000000000c9

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

h~(y) = 0x055fe3d3d923afdb92c3e44a1e9ae34c249b7f3eb1

EU = 0x0305cf2e1de9dcf32160bef47df954851b52a226f463
```

```
EU’ = 0x0204783f61a7493d83d76b8178c0935a1830b8708ea8

alpha = 0xd8e475b97184ee436903685198f494fbaa979816

r’ = 0x0267bffb82048609976b545bc4311c57e0869cf07c

v(x) = 0x06904e3cfb1b97cda28216f7caeca93fb005122cd3

v(y) = 0x05d0ae5a5d32e563575bfe4f59a2e5a18151163070

EV = 0x0306904e3cfb1b97cda28216f7caeca93fb005122cd3

PEH = 0x000b420ea755ce20f5fa8ea1015d0d2cbf5860767f

CO = 0x0305cf2e1de9dcf32160bef47df954851b52a226f4630204783f61a7493d83d76
     b8178c0935a1830b8708ea80306904e3cfb1b97cda28216f7caeca93fb005122cd3

K = 0xa0e34391d4fd70e6e780d7edb112ab475d88d3cd9782fd6365aca96b67cfeee964
    1bf7ee8176ec16db20623729a5001ec8e69779ecb3d25e9d128fe22aa4fc3056a032
    969279bb2eeaa2af3e9e5708e8b2b92d2d3f8932adeac7181c7ae03b663883fac467
    e54579cc7531dd3226fd94504c8a8bb60c2ad8cdb2aca4ef8664c9
```

C.5 Test vectors for *RSAES*

C.5.1 Test vector

```
===============
RSAES
===============
Rem=Rem1(Hash=Sha1(), Kdf=Kdf1(Hash=Sha1()))

-----

Public Key

n = 1096769317767533941413945645147207342367965840228428205076139459783O
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

e = 65537

-----

Private Key

n = 10967693177675339414139456451472073423679658402284282050761394597830
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

d = 36604719910171765415791435519332386590192588649642812654882582974250
    35561116224175300745978157072171393796773377539442551140602428629298
    12363545353590295192906382395640986479188889136284619444866954451931
    90809948446596269042966714133764211707743041789247544284660831177834
    9289048921023267935264351364735548141

===============================
Trace for RSAES encrypt
===============================

Message in ASCII = " This is a test message !!!"

Message as octet string = 0x205468697320697320612074657374206d6573736167
                          6520212121

Label in ASCII = "Label"

Label as octet string = 0x4c6162656c

seed = 0xd6e168c5f256a2dcff7ef12facd390f393c7a88d

DataBlock = 0x74341e3c271df3c784e595b804b1f90be0f80429000000000000000000
            000000000000000000000000000000000000000000000000000000000000
            00000000000000000000000000000000000000000120546869732069732 0
            612074657374206d65737361676520212121

DataBlockMask = 0xc325ebbb41a82551d5d0ad4834870a05ef3918c8caae38873f07dc
                a43127a4dee36a6ca5970f6c06926037de7df79c4915d83ff705821d
                2c46a1fa7bb81b73e27176feb7fd3a45e40b843f1aaebccb1ef4fa7e
                e3b9b491a342f43eaaa435efded41e0a3a6ec2eff1f2ed95

MaskedDataBlock = 0xb711f58766b5d696513538f03036f30e0fc11ce1caae38873f07
                  dca43127a4dee36a6ca5970f6c06926037de7df79c4915d83ff705
                  821d2c46a1fa7bb81b73e27176feb7fd3a45e40b843f1aaebccb1f
                  d4ae168aca94f8d062951edec1469bfeb97b79490fa58ad1d3ccb4

SeedMask = 0x281d7cb2d7d5531ed1f9382152d9be9a89a1df09

MaskedSeed = 0xfefc14772583f1c22e87c90efe0a2e691a667784

E = 0x00fefc14772583f1c22e87c90efe0a2e691a667784b711f58766b5d696513538f0
    3036f30e0fc11ce1caae38873f07dca43127a4dee36a6ca5970f6c06926037de7df7
    9c4915d83ff705821d2c46a1fa7bb81b73e27176feb7fd3a45e40b843f1aaebccb1f
    d4ae168aca94f8d062951edec1469bfeb97b79490fa58ad1d3ccb4

C = 0x4712734b1d3c9e43bc8ca30f4d93c88b6273075cb59a63ed2de383cf1a719afc42
    99919813f3b775153ef66121fea89821e6ef57427cbb03628884db2aed8e980bce93
    1205efdd3d6ee2e2ffc32a8266176ceee26dda7e3ed664c70c97c21187e97e1ccafa
    0c1b2e504552ff81d2aa683d89c77b37e9f7818aaf09b7fb585daf
```

C.5.2 Test vector

```
===============
RSAES
===============
Rem=Rem1(Hash=Sha1(), Kdf=Kdf2(Hash=Sha1()))

-----

Public Key

n = 1096769317767533941413945645147207342367965840228428205076139459783
    0409892052941241561970885131442367148322550031719583343578917449141
    78718642603750662788855742326532564254343296113773973874542733322600
    3651562339652352922811469386523033747515254261027325307114304734669
    0365642884618438728252895009596756788538 1

e = 65537

-----

Private Key

n = 1096769317767533941413945645147207342367965840228428205076139459783
    0409892052941241561970885131442367148322550031719583343578917449141
    78718642603750662788855742326532564254343296113773973874542733322600
    3651562339652352922811469386523033747515254261027325307114304734669
    0365642884618438728252895009596756788538 1

d = 3660471991017176541579143551933238659019258864964281265488258297425
    0355611162241753007459781570721713937967733775394425511406024286292
    98123635453535902951929063823956409864791888891362846194448669544519
    31908099484465962690429667141337642117077430417892475442846608311778
    34928904892102326793526435136473548141

===============================
Trace for RSAES encrypt
===============================

Message in ASCII = " This is a test message !!!"

Message as octet string = 0x205468697320697320612074657374206d6573736167
                          6520212121

Label in ASCII = "Label"

Label as octet string = 0x4c6162656c
```

```
seed = 0xd6e168c5f256a2dcff7ef12facd390f393c7a88d

DataBlock = 0x74341e3c271df3c784e595b804b1f90be0f80429000000000000000000
            000000000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000001205468697320697320
            612074657374206d65737361676520212121

DataBlockMask = 0xcaae38873f07dca43127a4dee36a6ca5970f6c06926037de7df79c
                4915d83ff705821d2c46a1fa7bb81b73e27176feb7fd3a45e40b843f
                1aaebccb1ef4fa7ee3b9b491a342f43eaaa435efded41e0a3a6ec2ef
                f1f2ed951285c5776e259a31024b20beab5cfa02db497746

MaskedDataBlock = 0xbe9a26bb181a2f63b5c23166e7db95ae77f7682f926037de7df7
                  9c4915d83ff705821d2c46a1fa7bb81b73e27176feb7fd3a45e40b
                  843f1aaebccb1ef4fa7ee3b9b491a342f43eaaa435efded41e0a3b
                  4e96879881cdfc61a5a4571a40e945222645cdd83d9d67fb685667

SeedMask = 0x0bfaec4d57584c957e242aa0ef72860f3e109d42

MaskedSeed = 0xdd1b8488a50eee49815adb8f43a116fcadd735cf

E = 0x00dd1b8488a50eee49815adb8f43a116fcadd735cfbe9a26bb181a2f63b5c23166
    e7db95ae77f7682f926037de7df79c4915d83ff705821d2c46a1fa7bb81b73e27176
    feb7fd3a45e40b843f1aaebccb1ef4fa7ee3b9b491a342f43eaaa435efded41e0a3b
    4e96879881cdfc61a5a4571a40e945222645cdd83d9d67fb685667

C = 0x7e72db6f8d55e9ef81e7486a891dd6f3399cd6275f817cf2978a64577fc276e8a8
    b0108d42d671867e22fd76ee2b59cca834a548aeb7b8f1e635ad719a9530b435d2bc
    8d2b15eeb2e162e9573d9765bcc9e4fbededdf6f1ef277aed2449214ffcb998734e1
    d1ba948e84e79f67d2c2a441509899222de4131819718bde30c471
```

C.5.3 Test vector

```
===============
RSAES
===============
Rem=Rem1(Hash=Sha256(outlen=20), Kdf=Kdf1(Hash=Sha256(outlen=20)))

-----

Public Key

n = 1096769317767533941413945645147207342367965840228428205076139459783O
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

e = 65537
```

```
-----

Private Key

n = 10967693177675339414139456451472073423679658402284282050761394597830
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

d = 36604719910171765415791435519332386590192588649642812654882582974250
    35561116224175300745978157072171393796773377539442551140602428629298
    12363545353590295192906382395640986479188889136284619444866954451931
    90809948446596269042966714133764211707743041789247544284660831177834
    9289048921023267935264351364735 48141

================================
Trace for RSAES encrypt
================================

Message in ASCII = " This is a test message !!!"

Message as octet string = 0x2054686973206973206120746573742 06d6573736167
                          6520212121

Label in ASCII = "Label"

Label as octet string = 0x4c6162656c

seed = 0xd6e168c5f256a2dcff7ef12facd390f393c7a88d

DataBlock = 0x0e66373f45dcf3dd656151e519f7ee5e3d558d9c000000000000000000
            000000000000000000000000000000000000000000000000000000000000
            000000000000000000000000000000000000000001205468697320697320
            612074657374206d6573736167652021212 1

DataBlockMask = 0x0742ba966813af75536bb6149cc44fc256fd6406df79665bc31dc5
                a62f70535e52c53015b9d37d412ff3c1193439599e1b628774c50d9c
                cb78d82c425e4521ee47b8c36a4bcffe8b8112a89312fc04420a39de
                99223890e74ce10378bc515a212b97b8a6447ba6a8870278

MaskedDataBlock = 0x09248da92dcf5ca8360ae7f18533a19c6ba8e99adf79665bc31d
                  c5a62f70535e52c53015b9d37d412ff3c1193439599e1b628774c5
                  0d9ccb78d82c425e4521ee47b8c36a4bcffe8b8112a89312fc0443
                  2a6db6f05118f9946c80230cd9222e0146f2cbd5251cc388a62359

SeedMask = 0x6f0195f38eed2417aa6eb7a365245073e58711db

MaskedSeed = 0xb9e0fd367cbb86cb5510468cc9f7c0807640b956

E = 0x00b9e0fd367cbb86cb5510468cc9f7c0807640b95609248da92dcf5ca8360ae7f1
```

```
    8533a19c6ba8e99adf79665bc31dc5a62f70535e52c53015b9d37d412ff3c1193439
    599e1b628774c50d9ccb78d82c425e4521ee47b8c36a4bcffe8b8112a89312fc0443
    2a6db6f05118f9946c80230cd9222e0146f2cbd5251cc388a62359

C = 0x04652a946c1b2a9cade87c46f1995f1a531008cd04bc10d46c850094234c856dd8
    57140a46f9d4d059b5e184dc5f57baac374655911eba712d0b2cd4a92af5ebcfd5ef
    cdc484b8236e85f237c2eec163fe836ad4c002d6604fe0021f4b3835028c98af97e3
    c37c646227e1c488bff9bfd4dc430ac04b4aadc7a9cf6d335e4913
```

C.5.4 Test vector

```
===============
RSAES
===============
Rem=Rem1(Hash=Sha256(outlen=20), Kdf=Kdf2(Hash=Sha256(outlen=20)))

-----

Public Key

n = 10967693177675339414139456451472073423679658402284282050761394597830
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

e = 65537

-----

Private Key

n = 10967693177675339414139456451472073423679658402284282050761394597830
    40989205294124156197088513144236714832255003171958334357891744914178
    71864260375066278885574232653256425434296113773973874542733322600365
    15623396523529228114693865230337475152542610273253071143047346690365
    6428846184387282528950095967567885381

d = 36604719910171765415791435519332386590192588649642812654882582974250
    35561116224175300745978157072171393796773377539442551140602428629298
    12363545353590295192906382395640986479188889136284619444866954451931
    90809948446596269042966714133764211707743041789247544284660831177834
    928904892102326793526435136473548141

================================
Trace for RSAES encrypt
================================

Message in ASCII = " This is a test message !!!"
Message as octet string = 0x2054686973206973206120746573742064657373616 7
                          6520212121
```

```
Message as octet string = 0x20546869732069732061207465737420 6d6573736167
```

```
Public Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

e = 65537

-----

Private Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

d = 32023135558599481863153745244741739956797835803921402370443497280464
    79396037520308981353808895461806395564474639124525446044708705259675
    840210989546479265


-------------------------------
Trace for RSA-KEM encrypt
-------------------------------

r = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4

R = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4
C0 = 0x4603e5324cab9cef8365c817052d954d44447b1667099edc69942d32cd594e4ff
     cf268ae3836e2c35744aaa53ae201fe499806b67dedaa26bf72ecbd117a6fc0

K = 0x5f8de105b5e96b2e490ddecbd147dd1def7e3b8e0e6a26eb7b956ccb8b3bdc1ca9
    75bc57c3989e8fbad31a224655d800c46954840ff32052cdf0d640562bdfadfa263c
    fccf3c52b29f2af4a1869959bc77f854cf15bd7a25192985a842dbff8e13efee5b7e
    7e55bbe4d389647c686a9a9ab3fb889b2d7767d3837eea4e0a2f04
```

C.6.2 Test vector

```
-------------
RSA-KEM
-------------

Kdf=Kdf2(Hash=Sha1())
keylen=128

-----
```

```
Public Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

e = 65537

-----

Private Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

d = 32023135558599481863153745244741739956797835803921402370443497280464
    79396037520308981353808895461806395564474639124525446044708705259675
    840210989546479265


-------------------------------
Trace for RSA-KEM encrypt
-------------------------------

r = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4

R = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4
C0 = 0x4603e5324cab9cef8365c817052d954d44447b1667099edc69942d32cd594e4ff
     cf268ae3836e2c35744aaa53ae201fe499806b67dedaa26bf72ecbd117a6fc0

K = 0x0e6a26eb7b956ccb8b3bdc1ca975bc57c3989e8fbad31a224655d800c46954840f
    f32052cdf0d640562bdfadfa263cfccf3c52b29f2af4a1869959bc77f854cf15bd7a
    25192985a842dbff8e13efee5b7e7e55bbe4d389647c686a9a9ab3fb889b2d7767d3
    837eea4e0a2f04b53ca8f50fb31225c1be2d0126c8c7a4753b0807
```

C.6.3 Test vector

```
-------------
RSA-KEM
-------------

Kdf=Kdf1(Hash=Sha256(outlen=20))
keylen=128

-----
```

```
Public Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

e = 65537

-----

Private Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

d = 32023135558599481863153745244741739956797835803921402370443497280464
    79396037520308981353808895461806395564474639124525446044708705259675
    840210989546479265


-------------------------------
Trace for RSA-KEM encrypt
-------------------------------

r = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4

R = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4
C0 = 0x4603e5324cab9cef8365c817052d954d44447b1667099edc69942d32cd594e4ff
     cf268ae3836e2c35744aaa53ae201fe499806b67dedaa26bf72ecbd117a6fc0

K = 0x09e2decf2a6e1666c2f6071ff4298305e2643fd510a2403db42a8743cb989de86e
    668d168cbe604611ac179f819a3d18412e9eb45668f2923c087c12fee0c5a0d2a8aa
    70185401fbbd99379ec76c663e875a60b4aacb1319fa11c3365a8b79a44669f26fb5
    55c80391847b05eca1cb5cf8c2d531448d33fbaca19f6410ee1fcb
```

C.6.4 Test vector

```
-------------
RSA-KEM
-------------

Kdf=Kdf2(Hash=Sha256(outlen=20))
keylen=128

-----
```

```
Public Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

e = 65537

-----

Private Key

n = 58881133325026912517619364310092848849666407571798023374905464783262
    38537107326596800820237597139824869184990638749556269785797065508097
    452399642780486933

d = 32023135558599481863153745244741739956797835803921402370443497280464
    79396037520308981353808895461806395564474639124525446044708705259675
    840210989546479265


------------------------------
Trace for RSA-KEM encrypt
------------------------------

r = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4

R = 0x032e45326fa859a72ec235acff929b15d1372e30b207255f0611b8f785d7643741
    52e0ac009e509e7ba30cd2f1778e113b64e135cf4e2292c75efe5288edfda4
C0 = 0x4603e5324cab9cef8365c817052d954d44447b1667099edc69942d32cd594e4ff
     cf268ae3836e2c35744aaa53ae201fe499806b67dedaa26bf72ecbd117a6fc0

K = 0x10a2403db42a8743cb989de86e668d168cbe604611ac179f819a3d18412e9eb456
    68f2923c087c12fee0c5a0d2a8aa70185401fbbd99379ec76c663e875a60b4aacb13
    19fa11c3365a8b79a44669f26fb555c80391847b05eca1cb5cf8c2d531448d33fbac
    a19f6410ee1fcb260892670e0814c348664f6a7248aaf998a3acc6
```

C.7 Test vectors for *HC*

Combining a *KEM* with a *DEM* is fairly straightforward, but enumerating all the different possible combinations would be quite tedious and lengthy. We give just one test vector here as an illustration.

C.7.1 Test vector

```
=============
Hybrid Cipher
=============
-------------
DEM1
-------------

SC=SC1(BC=AES(keylen=32))
MAC=HMAC(Hash=Sha1(), keylen=20, outlen=20)

-------------
ACE-KEM
-------------

Kdf=Kdf1(Hash=Sha1())
Hash=Sha1()
Keylen=52
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 HC encrypt
==================================

-------------------------------
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
    533d79ce5ce59b5c778c2874f3330e03e02056


-------------------------------
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 = 0x94a6b23344a026db8e3f2669562ad8fc06a529befb032d89a192a460d0340f5a

k’ = 0x7d533d79ce5ce59b5c778c2874f3330e03e02056

c = 0x4e11a54ddf582716b4d46b75adcd446a173ca235b70a944901d2e6f8a583a01993
    bfebf63d92496654e5fe271784a310

T = 0x4e11a54ddf582716b4d46b75adcd446a173ca235b70a944901d2e6f8a583a01993
    bfebf63d92496654e5fe271784a3107465737400000000000000020

MAC = 0xd4a406ce2e48b63c3d054b91c354b4eeb4a16941

C1 = 0x4e11a54ddf582716b4d46b75adcd446a173ca235b70a944901d2e6f8a583a0199
     3bfebf63d92496654e5fe271784a310d4a406ce2e48b63c3d054b91c354b4eeb4a1
     6941
```

C.8 Test vectors for *EPOC-2*

```
==============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.1 ( 1152 bits )
```

```
===============================================================================

******** EPOC-2 Private Key ********

p = 0xf622e39d0510d9bdae7f02e457a833fe9e37b59929ea938f0b1984
    0f183ea01037ce053fc98609c9b52ebde468c003c5
q = 0xbbf752fc80599253a26e18a2105d462f99bb7b458d1d54d3042c8d
    c075f67884324dfbed34b8dd0fe637753a38d08165
w = 0xdb928872ee7566e674a165ff650a322eff82a0382be26f46bd1017
    a2a59e1f66e8cba0e39cc06e55c322f854aa7340c6

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xadc2ac34c36b4b914c682cbc0d5fb52e18cb170069bec7d5e1b863
    1a36c54f4c01b8e0ad6976dc1a8ac0e978b1571495760273d4c5cdcd
    85d8b1fd6a13c8687ee83ae5e182df9862830a471c61fb63a45c7708
    6dca95e1b849442daf80975e0ece7b00d78538b748f7161c22935c97
    9e29d8b29a024df9afb314fb1ecfe5f7cb9632696023a18bb45b5e85
    87186d3e5d
g = 0x02
h = 0x9b65e85d0f1ce25857f2329596cecf9286460072cbf221e79819ae
    228ed40f5bee941f46f1a7d6cee6a8fc16e512eaec764597575bbd25
    2264249380dbe091753c1896c910b9c70118c501d780369d30266fad
    daf604738af906af7414b14781d2eaf7af12c69491c1f6f913e27c65
    4e7a6f52e0a00ba035f730ad299602692d4bc25dd5a2bb13a0c8fe76
    e8e71baa3f

******** EPOC-2 Encryption Data ********

M = 0x0e3135ce4877c82c2a1f28428093a96a
L = ( null string )
seed = 0x000000000000000000000000000000000000000000000000000
       0000000000000000000000000000000000000000000
K = KDF(seed, SC.KeyLen) = 0x71ceb77b5c33163f81ad90d17a6b7594
C = SC.Encrypt(K, M) = 0xb2d933dbf6dc49d89a46ab47ce727157b60
                       f6e81a44dcf1c801e225e261e8b0f
H = Hash(DB) = 0xf5db56f9461d217e64bad228176679efc46dce4a
T = KDF'(H, rLen) = 0x6895b6f48be7a93d834a15ec0364bb27a3fce9
                    6bafe15a0362e703a16f7c64b9c7f6b0a8f2efbd
                    a66863abfca728ff8309544a1d5453567913728f
                    d653d4e77d8a09557d0c23c2c27a733d348d3592
                    0c762351e870454eb7cd46fe009d3adf8d0107db
                    1a98ff96ac50cbc1293ec03180dacae4c1503817
```

```
                   00
f = 0x000000000000000000000000000000000000000000000000000000
    0000000000000000000000000000000000000000
r = 0x6895b6f48be7a93d834a15ec0364bb27a3fce96bafe15a0362e703
    a16f7c64b9c7f6b0a8f2efbda66863abfca728ff8309544a1d545356
    7913728fd653d4e77d8a09557d0c23c2c27a733d348d35920c762351
    e870454eb7cd46fe009d3adf8d0107db1a98ff96ac50cbc1293ec031
    80dacae4c150381700

C0 = 0x92ce9d30b218c9962bdf6deb1699382d1fa2a7be3bc07c5068371
     aab4a841b7df4d67ad76f2ae70ad39d7fa8dde0bd45a85d76050e92
     11ad55b03bd729c92973dde3d9499194cc7b2093f9c1868fc3d6092
     76252f087dfccc543f61478fc0fc13e62ea7e1da57077532f0d9f3f
     f1bd6dfe42b7dc2946a8fd9a2835f00b4a3b42469a7f946728c0e96
     b152a4a4f7d16b0
C1 = 0xb2d933dbf6dc49d89a46ab47ce727157b60f6e81a44dcf1c801e2
     25e261e8b0f

C = C0||C1 = 0x92ce9d30b218c9962bdf6deb1699382d1fa2a7be3bc07
             c5068371aab4a841b7df4d67ad76f2ae70ad39d7fa8dde0
             bd45a85d76050e9211ad55b03bd729c92973dde3d949919
             4cc7b2093f9c1868fc3d609276252f087dfccc543f61478
             fc0fc13e62ea7e1da57077532f0d9f3ff1bd6dfe42b7dc2
             946a8fd9a2835f00b4a3b42469a7f946728c0e96b152a4a
             4f7d16b0b2d933dbf6dc49d89a46ab47ce727157b60f6e8
             1a44dcf1c801e225e261e8b0f

=============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.2 ( 1152 bits )
=============================================================================

******** EPOC-2 Private Key ********

p = 0xebb009072ab385e6907b6b2e5d551809b10d1c23e586674f7ae7be
    3149a8bf29300c7105630f1ef947a626a923642e73
q = 0x9873274558b92a11f8925f320d8397f43ac2c8de7535851a6541a2
    d2fa7e0603e1eaae98c0843d054ab5e2d4c8f3878b
w = 0xd0b7f4647df7edef29a6a245ac437a30965fbff077dd77c0c7879f
    b83074a2298d9ab6b44a355b330e4e3d8701ab1128

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0x81379f8036da3ed30d2a47ad21870a70222fd91529c20a55319474
```

```
    1707430d7f4f7a0f206e01fded8a7d85c69f81ca203e024493985aa4
    b36cd22a40904ae21c9b34734c0fa3edb39cb49fb0d78afad1c16227
    965100a75217a2d26bd8d074a7fbb8da92f9349276e2f6f94d7111f1
    ba0aa2bfea0a5feeb126802d6aac40f362c98531a4066a706de14f44
    7a09b5c7c3
g = 0x02
h = 0x3511042779eea4fa7c661d3f537690e90b9875b954c94f414324e5
    50dd965748756ac939f7c47f13591cf17acc42b8f7bfcb2dca43da73
    a2655e2bab74d554158ec98e60f71a79746b8021d4547420a0b5970e
    f5f9bf58d016de640e5f30d5e38a60d407015a217104f4949b24528d
    85bb974c96f3ac8a9faea8eb06c859408a6bc6abd0b45f5b8a37a4b2
    05b7ed1427

******** EPOC-2 Encryption Data ********

M = 0xfc898a7cf9f94370f5b3451f1128c926
L = ( null string )
seed = 0xdb3b02ca134ee8f4d08a7ec7b3a946e0ca752daa2246eb9e4d2
       66452b3e662e787c10f2d6d4de7161c30173d776934
K = KDF(seed, SC.KeyLen) = 0x6ca45a262e131ae32459f42147fed204
C = SC.Encrypt(K, M) = 0xcd862851c3cbeef42005399d576d6fca67f
                         f9468eb4a13a4433be9cb11fd6f0b
H = Hash(DB) = 0x4e59ade1dba394472a6a7688816c2200164f95b6
T = KDF'(H, rLen) = 0x122b760b3ab27e58fdfba7745a1e337536a3d5
                    bf923a6d6518ccc0aef3d907f0c9ddef8054e4b0
                    24941cdf442522df9778a5a3c4de9bbb28a8699c
                    87dfb390300ef9eabcea0161212d4c17be96bad6
                    4023ba11e081318909e41fca443697f9a49568cd
                    cd4c1f940530c7142ff7f4aebe75e50c75a8f60c
                    eb
f = 0xdb3b02ca134ee8f4d08a7ec7b3a946e0ca752daa2246eb9e4d2664
    52b3e662e787c10f2d6d4de7161c30173d776934
r = 0x122b760b3ab27e58fdfba7745a1e337536a3d5bf923a6d6518ccc0
    aef3d907f0c9ddef8054e4b024941cdf442522df9778a5a3c4de9bbb
    28a8699c87dfb390300ef9eabcea0161212d4c17be96bad64023ba11
    e081318909e41fca443697f9a49568cdcd4c1f940530c7142ff7f4ae
    be75e50c75a8f60ceb

C0 = 0x593c9d98817c68c89e5ba23904dbcaa8bae39d90fc7cddbc0c016
     c894ad7ba05400bbb4525aec048820afa1634102c1b7903a61b9dbe
     65966932d237d94801864c27214e567064b4bc67d3d5caf3f896623
     2041a0ecec9997c2b09c3844cb5a346474f1b6c1fe1893089b98e96
     e9c488f1e269c42e82115c23757b96c855ac2b868e7683e25684183
     04ccfdbeb51b45d
C1 = 0xcd862851c3cbeef42005399d576d6fca67ff9468eb4a13a4433be
     9cb11fd6f0b

C = C0||C1 = 0x593c9d98817c68c89e5ba23904dbcaa8bae39d90fc7cd
             dbc0c016c894ad7ba05400bbb4525aec048820afa163410
             2c1b7903a61b9dbe65966932d237d94801864c27214e567
             064b4bc67d3d5caf3f8966232041a0ecec9997c2b09c384
             4cb5a346474f1b6c1fe1893089b98e96e9c488f1e269c42
             e82115c23757b96c855ac2b868e7683e2568418304ccfdb
             eb51b45dcd862851c3cbeef42005399d576d6fca67ff946
             8eb4a13a4433be9cb11fd6f0b

==============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.3 ( 1152 bits )
==============================================================================

******** EPOC-2 Private Key ********

p = 0xebb009072ab385e6907b6b2e5d551809b10d1c23e586674f7ae7be
    3149a8bf29300c7105630f1ef947a626a923642e73
q = 0x9873274558b92a11f8925f320d8397f43ac2c8de7535851a6541a2
    d2fa7e0603e1eaae98c0843d054ab5e2d4c8f3878b
w = 0xd0b7f4647df7edef29a6a245ac437a30965fbff077dd77c0c7879f
    b83074a2298d9ab6b44a355b330e4e3d8701ab1128

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0x81379f8036da3ed30d2a47ad21870a70222fd91529c20a55319474
    1707430d7f4f7a0f206e01fded8a7d85c69f81ca203e024493985aa4
    b36cd22a40904ae21c9b34734c0fa3edb39cb49fb0d78afad1c16227
    965100a75217a2d26bd8d074a7fbb8da92f9349276e2f6f94d7111f1
    ba0aa2bfea0a5feeb126802d6aac40f362c98531a4066a706de14f44
    7a09b5c7c3
g = 0x02
h = 0x3511042779eea4fa7c661d3f537690e90b9875b954c94f414324e5
    50dd965748756ac939f7c47f13591cf17acc42b8f7bfcb2dca43da73
    a2655e2bab74d554158ec98e60f71a79746b8021d4547420a0b5970e
    f5f9bf58d016de640e5f30d5e38a60d407015a217104f4949b24528d
    85bb974c96f3ac8a9faea8eb06c859408a6bc6abd0b45f5b8a37a4b2
    05b7ed1427

******** EPOC-2 Encryption Data ********

M = 0x808ec102dee0c930c3265b9297f64215
L = ( null string )
seed = 0x9a5db6fd3ccec8242a93496573cf1fef9aef982bb66a8bd2324
       39c28878415bdae11380f21b06f0d0b79a2c5170342
K = KDF(seed, SC.KeyLen) = 0x03bf02285e514238f16750ab892e544f
C = SC.Encrypt(K, M) = 0x948b9e1c9a55d62998c9ae3dd0edafc33f3
                       2f5c8e87916091edf00c67b470c81
```

```
H = Hash(DB) = 0x8afc9b710e25938d2b513f06fd5dd86f566b562f
T = KDF'(H, rLen) = 0x2d5b371c1b5e1eca40fc6e90f1e8eb8be2c409
                    9fff5d3c8dc9d3172250d6b09d8b89cb4787db60
                    e7a9801e7ed2804aa595664d5a811284566fcdae
                    536949c11e3518db02bf6581643c045e6d4f5980
                    e8e791e78fb2f9cc375e77e44bbb909f4519e330
                    50ed5a6014d98fb953c408b4c88952667e7db9b1
                    d5
f = 0x9a5db6fd3ccec8242a93496573cf1fef9aef982bb66a8bd232439c
    28878415bdae11380f21b06f0d0b79a2c5170342
r = 0x2d5b371c1b5e1eca40fc6e90f1e8eb8be2c4099fff5d3c8dc9d317
    2250d6b09d8b89cb4787db60e7a9801e7ed2804aa595664d5a811284
    566fcdae536949c11e3518db02bf6581643c045e6d4f5980e8e791e7
    8fb2f9cc375e77e44bbb909f4519e33050ed5a6014d98fb953c408b4
    c88952667e7db9b1d5

C0 = 0x73a696586f7b435ab857d1f94dba061f069dee6beb9567a215d60
     ffe8765439fa7904ddc999c5735f3b6ced12fad19ddb66cb020c538
     a41b935184a4df47a39199a3fa911143085309f6dd918032b68ac8b
     e17e8714b92216b96ccf052508b4e3726fee6cde02f204d77e14ccc
     c761cee7a8e6bed58f305caee85edf5b37c5da643df733c9522ed95
     33e28ef661e16b9
C1 = 0x948b9e1c9a55d62998c9ae3dd0edafc33f32f5c8e87916091edf0
     0c67b470c81

C = C0||C1 = 0x73a696586f7b435ab857d1f94dba061f069dee6beb956
             7a215d60ffe8765439fa7904ddc999c5735f3b6ced12fad
             19ddb66cb020c538a41b935184a4df47a39199a3fa91114
             3085309f6dd918032b68ac8be17e8714b92216b96ccf052
             508b4e3726fee6cde02f204d77e14cccc761cee7a8e6bed
             58f305caee85edf5b37c5da643df733c9522ed9533e28ef
             661e16b9948b9e1c9a55d62998c9ae3dd0edafc33f32f5c
             8e87916091edf00c67b470c81

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.4 ( 1152 bits )
===============================================================================

******** EPOC-2 Private Key ********

p = 0xebb009072ab385e6907b6b2e5d551809b10d1c23e586674f7ae7be
    3149a8bf29300c7105630f1ef947a626a923642e73
q = 0x9873274558b92a11f8925f320d8397f43ac2c8de7535851a6541a2
    d2fa7e0603e1eaae98c0843d054ab5e2d4c8f3878b
w = 0xd0b7f4647df7edef29a6a245ac437a30965fbff077dd77c0c7879f
    b83074a2298d9ab6b44a355b330e4e3d8701ab1128

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
```

```
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0x81379f8036da3ed30d2a47ad21870a70222fd91529c20a55319474
    1707430d7f4f7a0f206e01fded8a7d85c69f81ca203e024493985aa4
    b36cd22a40904ae21c9b34734c0fa3edb39cb49fb0d78afad1c16227
    965100a75217a2d26bd8d074a7fbb8da92f9349276e2f6f94d7111f1
    ba0aa2bfea0a5feeb126802d6aac40f362c98531a4066a706de14f44
    7a09b5c7c3
g = 0x02
h = 0x3511042779eea4fa7c661d3f537690e90b9875b954c94f414324e5
    50dd965748756ac939f7c47f13591cf17acc42b8f7bfcb2dca43da73
    a2655e2bab74d554158ec98e60f71a79746b8021d4547420a0b5970e
    f5f9bf58d016de640e5f30d5e38a60d407015a217104f4949b24528d
    85bb974c96f3ac8a9faea8eb06c859408a6bc6abd0b45f5b8a37a4b2
    05b7ed1427

******** EPOC-2 Encryption Data ********

M = 0xa83b5191793554f15ec1479345dbc6f2
L = ( null string )
seed = 0x83254130a56af70136d3499386fcdbc1c723df982dfdee6bf6d
       fa40035a624a66d93199194dce3dca79b50cb448c28
K = KDF(seed, SC.KeyLen) = 0xe08ee24009a0e1ac2baaa984a4541e99
C = SC.Encrypt(K, M) = 0x39d9392a1ccbe70852619270f0e1547d800
                       07b69686478a36b6a8281e4e93270
H = Hash(DB) = 0xa63a73792992ff3f536d99c181b02b827fde1dd0
T = KDF'(H, rLen) = 0x5da9e60f952767df184c36b43a09f79c87462a
                    c46cf5cb40d0e0c2fd98179cb4ad1d634ea50296
                    1222bccf08f6a6584cd5d55e0291cbbcc751c3d6
                    db4dc433be6651f57b1b2a9f7dc62d1e5aee8481
                    14aa447cd2d56b9512b02df71d4f77d08700eca0
                    7b6a82b03ca574990a82efef2494d39e88015a58
                    ae
f = 0x83254130a56af70136d3499386fcdbc1c723df982dfdee6bf6dfa4
    0035a624a66d93199194dce3dca79b50cb448c28
r = 0x5da9e60f952767df184c36b43a09f79c87462ac46cf5cb40d0e0c2
    fd98179cb4ad1d634ea502961222bccf08f6a6584cd5d55e0291cbbc
    c751c3d6db4dc433be6651f57b1b2a9f7dc62d1e5aee848114aa447c
    d2d56b9512b02df71d4f77d08700eca07b6a82b03ca574990a82efef
    2494d39e88015a58ae

C0 = 0x4680682145be1250626a85ae29718e759f8346cbdf7cd1bbde0d9
     0c386bfef1f4989c043f896b9ae14fd357a2974d9274a5be014de7f
     f05d2a929268f1155c8e2c6d388ace74ea8a167ec05cac797fbbb0a
     bd167e81d90883e0f2f64d79bcbe62d3274a8344fa6e277e244d5c4
     ad11bada2cf85ba323a19b28351322a452df9f0bbeeb9708ee5cfc7
     ee0dff25963de09
```

```
C1 = 0x39d9392a1ccbe70852619270f0e1547d80007b69686478a36b6a8
     281e4e93270

C = C0||C1 = 0x4680682145be1250626a85ae29718e759f8346cbdf7cd
             1bbde0d90c386bfef1f4989c043f896b9ae14fd357a2974
             d9274a5be014de7ff05d2a929268f1155c8e2c6d388ace7
             4ea8a167ec05cac797fbbb0abd167e81d90883e0f2f64d7
             9bcbe62d3274a8344fa6e277e244d5c4ad11bada2cf85ba
             323a19b28351322a452df9f0bbeeb9708ee5cfc7ee0dff2
             5963de0939d9392a1ccbe70852619270f0e1547d80007b6
             9686478a36b6a8281e4e93270

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.5 ( 1152 bits )
===============================================================================

******** EPOC-2 Private Key ********

p = 0xee16cf5d651913c475ffe37daddf65548386fbfc6f62bb02a867e7
    349cc233983e45f03d5698d59ce75d2c6909d507c7
q = 0xe8adfed4e4db6b62879587702af49329343e0e74ed1807222d32d8
    a059c38eb1fc6923522d59d9e8c47ae9dff754544b
w = 0x91c1ce919afca1342de94ca6544a56b3e5748fcf0654d85731c63f
    adf38f3b9d0fda5397ec63b7b366e92d08220b9a34

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xc942918df0520a549f11d8bcbf41923d90fd9deb4fae248b534b6b
    45fbbf2bc217ebb0b3ccb4468429a1d8982eed4369fc27607c8e85d4
    d9961f3830261c66f73ddd568135e7124f447f736b039712614b4c7c
    52bf1f449eb2d1f146f03034539b3d46329f2d00ed44be8636f67d86
    cf9e672883f82408ff40f6a3d3fa53992e11e36e0b17315bad7f60d0
    0623d89bdb
g = 0x02
h = 0x80e6dd57b6eb6433f1ca75f07a0c6b30b50a6eac45ced8288a2e07
    e339da05613203cf3cbb0c3b4d6d8005bef7f29df0306250e8af3114
    4d0c7509a13a2eb90a964d1096f6065db552dd720da1c1464a379bc2
    e6b20efb02f1ae17ad3a04a46985d917fdf5e02efc26d7ba2949ed91
    4bfaad81418d56f60d01a8f40d05f4c351fe11b661e596fef8199771
    76174b9296

******** EPOC-2 Encryption Data ********
```

```
M = 0x018f697cd0dd8e20eabe9c288032e0b8
L = ( null string )
seed = 0xf2ae02f7ce412022ed8edc7773b5833a4ad05a2a0cd976a4870
       6ecdaa952487c1c547f24a54a9f73d70b09d2426b8b
K = KDF(seed, SC.KeyLen) = 0xc4ad414f108c46467f427b5795662580
C = SC.Encrypt(K, M) = 0x5f531a1cb7695a229f7dafb8decba11a248
                       81b7198b8328a22fbde49ce157ba3
H = Hash(DB) = 0x62381a81a0e855d4fc8ebe947f30f6aa2ece3d2f
T = KDF'(H, rLen) = 0x33e206553812827c056b1ce19d0dc39244e958
                    88bccb9336d41ffd32d8f21b46703709950d3e56
                    13204fc0764da9c3a5ada8414d9bf3bc3fe7934b
                    2e5a8fce33e892f80edd40dbf47285e10dc29446
                    e6239d69c1105ebff31906dfc9e1157e3a914590
                    13002b2a8a8a7047aeb590727c211873cc16cde1
                    7e
f = 0xf2ae02f7ce412022ed8edc7773b5833a4ad05a2a0cd976a48706ec
    daa952487c1c547f24a54a9f73d70b09d2426b8b
r = 0x33e206553812827c056b1ce19d0dc39244e95888bccb9336d41ffd
    32d8f21b46703709950d3e5613204fc0764da9c3a5ada8414d9bf3bc
    3fe7934b2e5a8fce33e892f80edd40dbf47285e10dc29446e6239d69
    c1105ebff31906dfc9e1157e3a91459013002b2a8a8a7047aeb59072
    7c211873cc16cde17e

C0 = 0x044dbecbbe80221921316f330c3dc8f458c57dfa065ae0f52b31e
     2abc5ad8384277956371d4f90c29812edbb0cfbedad25070dbebea1
     6b5b70ef59e0ce1862fe0a7e6866f0859158840f4ed42d225881194
     d3dcdd0d295a08874df37c6b6caf074050f4d532777c586f3d94806
     fc37cac1671947647001dcad1ea6215bf5708d8774407eb9eff54fb
     e2a3cb147deedba
C1 = 0x5f531a1cb7695a229f7dafb8decba11a24881b7198b8328a22fbd
     e49ce157ba3

C = C0||C1 = 0x044dbecbbe80221921316f330c3dc8f458c57dfa065ae
             0f52b31e2abc5ad8384277956371d4f90c29812edbb0cfb
             edad25070dbebea16b5b70ef59e0ce1862fe0a7e6866f08
             59158840f4ed42d225881194d3dcdd0d295a08874df37c6
             b6caf074050f4d532777c586f3d94806fc37cac16719476
             47001dcad1ea6215bf5708d8774407eb9eff54fbe2a3cb1
             47deedba5f531a1cb7695a229f7dafb8decba11a24881b7
             198b8328a22fbde49ce157ba3

=================================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.6 ( 1152 bits )
=================================================================================

******** EPOC-2 Private Key ********

p = 0xee16cf5d651913c475ffe37daddf65548386fbfc6f62bb02a867e7
    349cc233983e45f03d5698d59ce75d2c6909d507c7
q = 0xe8adfed4e4db6b62879587702af49329343e0e74ed1807222d32d8
    a059c38eb1fc6923522d59d9e8c47ae9dff754544b
```

```
w = 0x91c1ce919afca1342de94ca6544a56b3e5748fcf0654d85731c63f
    adf38f3b9d0fda5397ec63b7b366e92d08220b9a34

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xc942918df0520a549f11d8bcbf41923d90fd9deb4fae248b534b6b
    45fbbf2bc217ebb0b3ccb4468429a1d8982eed4369fc27607c8e85d4
    d9961f3830261c66f73ddd568135e7124f447f736b039712614b4c7c
    52bf1f449eb2d1f146f03034539b3d46329f2d00ed44be8636f67d86
    cf9e672883f82408ff40f6a3d3fa53992e11e36e0b17315bad7f60d0
    0623d89bdb
g = 0x02
h = 0x80e6dd57b6eb6433f1ca75f07a0c6b30b50a6eac45ced8288a2e07
    e339da05613203cf3cbb0c3b4d6d8005bef7f29df0306250e8af3114
    4d0c7509a13a2eb90a964d1096f6065db552dd720da1c1464a379bc2
    e6b20efb02f1ae17ad3a04a46985d917fdf5e02efc26d7ba2949ed91
    4bfaad81418d56f60d01a8f40d05f4c351fe11b661e596fef8199771
    76174b9296

******** EPOC-2 Encryption Data ********

M = 0xf8949c1e35cc65ab18c7cac14624362e
L = ( null string )
seed = 0xb10de34e3f455c58704a31836cc8009bca3bb922d6ce13a5626
       d7186141ecf20d8098646206b8bc9477ddfc793dd08
K = KDF(seed, SC.KeyLen) = 0x8ec09bf4150d3a9caf8dba80b9b6df97
C = SC.Encrypt(K, M) = 0x05043c53ef7a86a29e7a6e5a14d1c333770
                       e3506130c5c1d4ab82542234938ae
H = Hash(DB) = 0x2258ed43c1b692ddd86c6cf4343123f2feff0d6d
T = KDF'(H, rLen) = 0x85a1522c0a974a4b574625694a2c5893ed1fa8
                    a53b79a1461ca03a9207f8edc8c955438353c6bc
                    590948a741905da53e87942d55c13a24cc605aaf
                    6871bbbc2457ac90f2bcc4e3ce3489fa546d063a
                    0fbf8653820ec43e6ba0fc1d21e5dc944380a108
                    a009afb8345251a4f2a683ef1185d414bfcb736e
                    64
f = 0xb10de34e3f455c58704a31836cc8009bca3bb922d6ce13a5626d71
    86141ecf20d8098646206b8bc9477ddfc793dd08
r = 0x85a1522c0a974a4b574625694a2c5893ed1fa8a53b79a1461ca03a
    9207f8edc8c955438353c6bc590948a741905da53e87942d55c13a24
    cc605aaf6871bbbc2457ac90f2bcc4e3ce3489fa546d063a0fbf8653
    820ec43e6ba0fc1d21e5dc944380a108a009afb8345251a4f2a683ef
    1185d414bfcb736e64
```

```
C0 = 0x1cfbc2e267dd774d15c6281afe9e0645edc1421bcaac79172da7d
     15e3eaefb2ac1ce8d769137b2aaaac6fab124afca85952719c44855
     2d8d759523554b55cc9e08ea03b740c4851291ff751c3a64adf42a3
     39a0e0defa7be8f5e68434769f8b53ad6cd6234921d02260a0d79f3
     e8691ed581bf57f72e8e529a540242d8273642ce3c08f52f39309aa
     57f6c8f71ee5c31
C1 = 0x05043c53ef7a86a29e7a6e5a14d1c333770e3506130c5c1d4ab82
     542234938ae

C = C0||C1 = 0x1cfbc2e267dd774d15c6281afe9e0645edc1421bcaac7
             9172da7d15e3eaefb2ac1ce8d769137b2aaaac6fab124af
             ca85952719c448552d8d759523554b55cc9e08ea03b740c
             4851291ff751c3a64adf42a339a0e0defa7be8f5e684347
             69f8b53ad6cd6234921d02260a0d79f3e8691ed581bf57f
             72e8e529a540242d8273642ce3c08f52f39309aa57f6c8f
             71ee5c3105043c53ef7a86a29e7a6e5a14d1c333770e350
             6130c5c1d4ab82542234938ae

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.7 ( 1152 bits )
===============================================================================

******** EPOC-2 Private Key ********

p = 0xee16cf5d651913c475ffe37daddf65548386fbfc6f62bb02a867e7
    349cc233983e45f03d5698d59ce75d2c6909d507c7
q = 0xe8adfed4e4db6b62879587702af49329343e0e74ed1807222d32d8
    a059c38eb1fc6923522d59d9e8c47ae9dff754544b
w = 0x91c1ce919afca1342de94ca6544a56b3e5748fcf0654d85731c63f
    adf38f3b9d0fda5397ec63b7b366e92d08220b9a34

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xc942918df0520a549f11d8bcbf41923d90fd9deb4fae248b534b6b
    45fbbf2bc217ebb0b3ccb4468429a1d8982eed4369fc27607c8e85d4
    d9961f3830261c66f73ddd568135e7124f447f736b039712614b4c7c
    52bf1f449eb2d1f146f03034539b3d46329f2d00ed44be8636f67d86
    cf9e672883f82408ff40f6a3d3fa53992e11e36e0b17315bad7f60d0
    0623d89bdb
g = 0x02
h = 0x80e6dd57b6eb6433f1ca75f07a0c6b30b50a6eac45ced8288a2e07
    e339da05613203cf3cbb0c3b4d6d8005bef7f29df0306250e8af3114
    4d0c7509a13a2eb90a964d1096f6065db552dd720da1c1464a379bc2
    e6b20efb02f1ae17ad3a04a46985d917fdf5e02efc26d7ba2949ed91
    4bfaad81418d56f60d01a8f40d05f4c351fe11b661e596fef8199771
    76174b9296

******** EPOC-2 Encryption Data ********

M = 0x2c3018b8411540f98b4e8c6e655ec5ca
L = ( null string )
seed = 0xad8dc81673fdcd8f59b11de4c65f6bae0e7742c9a99983c3ba5
       0dafbb68321399f02759cd2037e705706d433508390
K = KDF(seed, SC.KeyLen) = 0xd6fb6d7568ead08717eb7a747c63cf6a
C = SC.Encrypt(K, M) = 0xda0eec9df5d6a83cbe400c134ece89c3374
                       9aaf9e05b7ea6d0036d94d458cafe
H = Hash(DB) = 0xb458e6f6d1a75926cd707e7810b8a959aea6e313
T = KDF'(H, rLen) = 0x0d0d86d7fddd9b6d340a159b0991fa47d029a0
                    79aaa0ce49e669a898c5c576070b321e0d39d1ea
                    6e454e5b111c3d915846fb4a2f3bf848e27b600a
                    2bc631e3fdc811b80dce3be6ba1201099834301f
                    64817ae1818943205808647 8e14ad30ab2cc2aec
                    e6d37ce3718880d78863fc04616677d1f5ef03c9
                    3f
f = 0xad8dc81673fdcd8f59b11de4c65f6bae0e7742c9a99983c3ba50da
    fbb68321399f02759cd2037e705706d433508390
r = 0x0d0d86d7fddd9b6d340a159b0991fa47d029a079aaa0ce49e669a8
    98c5c576070b321e0d39d1ea6e454e5b111c3d915846fb4a2f3bf848
    e27b600a2bc631e3fdc811b80dce3be6ba1201099834301f64817ae1
    818943205808647 8e14ad30ab2cc2aece6d37ce3718880d78863fc04
    616677d1f5ef03c93f

C0 = 0xb7994e6bb7d7300e7160e087d8e9884b736c7e563704cc7f5db22
     0cb548c8ba9dde3a1c7a8f829efee3b4417f3a213ee13d96102cfc2
     c43d75f2460487d93bf5890a39ccf3b6681405cf0f5430da60d9f86
     fa52cc8857efbbdd0adb1d6fcc75285ab79fcd3a1bd52ae1bdcb222
     0c5176fe6efcf02b037810b24b97a36ff558a5e8d8af00b93098e85
     eb3b9c7e143cb3c
C1 = 0xda0eec9df5d6a83cbe400c134ece89c33749aaf9e05b7ea6d0036
     d94d458cafe

C = C0||C1 = 0xb7994e6bb7d7300e7160e087d8e9884b736c7e563704c
             c7f5db220cb548c8ba9dde3a1c7a8f829efee3b4417f3a2
             13ee13d96102cfc2c43d75f2460487d93bf5890a39ccf3b
             6681405cf0f5430da60d9f86fa52cc8857efbbdd0adb1d6
             fcc75285ab79fcd3a1bd52ae1bdcb2220c5176fe6efcf02
             b037810b24b97a36ff558a5e8d8af00b93098e85eb3b9c7
             e143cb3cda0eec9df5d6a83cbe400c134ece89c33749aaf
             9e05b7ea6d0036d94d458cafe

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.8 ( 1152 bits )
===============================================================================
******** EPOC-2 Private Key ********

p = 0xff6fe61c802466ebca41547185229da76ab23ca0523cfb63503a2e
    9419948481d26fc38914927b7133ab7e67c515923b
q = 0xfba6b13bcdf6880235e592970ea618d47fccafc760b1a2db7c3e52
    1cb4464443852a86d0ba39fe5157e0a5aa5c10d643
w = 0x3ca8779dc0c9b93528485ea15a0be6ad44dde0686f724e11ddced8
    89374d061873bc9fd9ddce2fbd2c43cacbfcbfaf80

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xfa8bb28af9017e3a815eb7c25f370ae3accfc133cd806804143078
    f9f5a1b2f0e7b1df48fe7e2ed3a0f4650d9f816b6e0ce4c31330055f
    780768eb7909fe6d42ed1d09f548ec638917537200ff4e65489c9c61
    b3f2e42c13587300caa17c9c9567a88cb353819ca075682f1b991d7f
    8fd9126571d31c1746824acbe3c8611265692439cea8996dc72a3485
    cf596c590b
g = 0x02
h = 0x6f81fe2a320707ded0f021b0aedd2fd3d8169f0ca7d46a6dbb9f6e
    ffa9dc577cf3d10a6e2db2acec405ef9109616998da47cebb9b5a9b4
    512a6dd7e73214824fda6609e7b21c5202f1d4e5731726884a132bcf
    1078b2c57accb45443a6d4e53cfeed5de4a743dcabea297d5a69eee6
    f978ae5737ff5b7244228c739e6dbc24d5b4676f7ef233523fee2d27
    1470827aa8

******** EPOC-2 Encryption Data ********

M = 0xfb7083651ec39b2dd2c28adb9b340e04
L = ( null string )
seed = 0xd4610825a1dee4e879eeb0afadb27973512cb6c036c8fe45440
       8b899371b85b24a04ed22c9b6d8cdb01b17c8a90a4a
K = KDF(seed, SC.KeyLen) = 0xb55ec04a4109c363ecf5e50b0bf87cae
C = SC.Encrypt(K, M) = 0x2d880808be3e46585fed12ee1fcaf970adf
                       57653cfb9e24aed68583455b803c2
H = Hash(DB) = 0x23074d2ef85fa047f1b16f7fb8991fd049266977
T = KDF'(H, rLen) = 0xe744d1409bc2476a37fb029b1683b01a952845
                    aeb2a34bc399e8c54218dff01ef885f2eb3cd6e5
                    800aee67358244c5d0c5eac753be936ad9886355
                    467c6be250d179b549042807a46fbed7e8cfc90f
                    cf8433821ad03a6ab368cb8de2bf7b328c5bdb85
                    5bff8eea30e7669fcba930d841d053575beacdc1
                    c3
```

```
f = 0xd4610825a1dee4e879eeb0afadb27973512cb6c036c8fe454408b8
    99371b85b24a04ed22c9b6d8cdb01b17c8a90a4a
r = 0xe744d1409bc2476a37fb029b1683b01a952845aeb2a34bc399e8c5
    4218dff01ef885f2eb3cd6e5800aee67358244c5d0c5eac753be936a
    d9886355467c6be250d179b549042807a46fbed7e8cfc90fcf843382
    1ad03a6ab368cb8de2bf7b328c5bdb855bff8eea30e7669fcba930d8
    41d053575beacdc1c3

C0 = 0x50f9b53a707bcc3911c2f1b29570078176202355253620f04c73a
     6d918ef17d93d5561d6cb0c09f96c2fee45f606ef89279ea50e7d8f
     47ced7654340852eebf65005aec84f2f6ade0c3f16f5170a057b213
     a1e47fbb261655feeccdb6dfbb51e5cf8d4b71f1e614ed2f1ba3bd5
     4b6c39fd7e3393b10d52d3f398e961e78837f9a9ae6b58973e744cb
     08e613f9b0a702c
C1 = 0x2d880808be3e46585fed12ee1fcaf970adf57653cfb9e24aed685
     83455b803c2

C = C0||C1 = 0x50f9b53a707bcc3911c2f1b29570078176202355253620
             0f04c73a6d918ef17d93d5561d6cb0c09f96c2fee45f606
             ef89279ea50e7d8f47ced7654340852eebf65005aec84f2
             f6ade0c3f16f5170a057b213a1e47fbb261655feeccdb6d
             fbb51e5cf8d4b71f1e614ed2f1ba3bd54b6c39fd7e3393b
             10d52d3f398e961e78837f9a9ae6b58973e744cb08e613f
             9b0a702c2d880808be3e46585fed12ee1fcaf970adf5765
             3cfb9e24aed68583455b803c2

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.9 ( 1152 bits )
===============================================================================

******** EPOC-2 Private Key ********

p = 0xff6fe61c802466ebca41547185229da76ab23ca0523cfb63503a2e
    9419948481d26fc38914927b7133ab7e67c515923b
q = 0xfba6b13bcdf6880235e592970ea618d47fccafc760b1a2db7c3e52
    1cb4464443852a86d0ba39fe5157e0a5aa5c10d643
w = 0x3ca8779dc0c9b93528485ea15a0be6ad44dde0686f724e11ddced8
    89374d061873bc9fd9ddce2fbd2c43cacbfcbfaf80

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xfa8bb28af9017e3a815eb7c25f370ae3accfc133cd806804143078
    f9f5a1b2f0e7b1df48fe7e2ed3a0f4650d9f816b6e0ce4c31330055f
    780768eb7909fe6d42ed1d09f548ec638917537200ff4e65489c9c61
    b3f2e42c13587300caa17c9c9567a88cb353819ca075682f1b991d7f
    8fd9126571d31c1746824acbe3c8611265692439cea8996dc72a3485
    cf596c590b
g = 0x02
h = 0x6f81fe2a320707ded0f021b0aedd2fd3d8169f0ca7d46a6dbb9f6e
    ffa9dc577cf3d10a6e2db2acec405ef9109616998da47cebb9b5a9b4
    512a6dd7e73214824fda6609e7b21c5202f1d4e5731726884a132bcf
    1078b2c57accb45443a6d4e53cfeed5de4a743dcabea297d5a69eee6
    f978ae5737ff5b7244228c739e6dbc24d5b4676f7ef233523fee2d27
    1470827aa8

******** EPOC-2 Encryption Data ********

M = 0x4600806a2093ab7d6582de22fb7040b4
L = ( null string )
seed = 0xdebce01be6ac1fb46291a99058fa6b4f866b7d579838de8f10a
       622f946055a4179fb25eb148c597f3d57d53346b070
K = KDF(seed, SC.KeyLen) = 0xa8326c3adce17be4f32f0d6dcd7b8e19
C = SC.Encrypt(K, M) = 0xe52cceffaa6232c7412666bbaf27c581802
                       fbd84e4ba2773a97983bf047eb849
H = Hash(DB) = 0xd3b84b32776a7727ddd2e97dc6576996458a88ee
T = KDF'(H, rLen) = 0x34413272aa5d25ac327d340e8a3b980c4cf0e9
                    31fe0b10a386457ff9e54e36315b30d491393e2f
                    72d107bc118c888c6fd6d651deb0b1ce1acee991
                    42287b7212c37775b7c5570211b37dd8cfbfd72a
                    3971f7a03568e3e644a831f77690a9a61c5fef62
                    ebf77b40da39c6e6fe411e3cebe34941383d3c8a
                    58
f = 0xdebce01be6ac1fb46291a99058fa6b4f866b7d579838de8f10a622
    f946055a4179fb25eb148c597f3d57d53346b070
r = 0x34413272aa5d25ac327d340e8a3b980c4cf0e931fe0b10a386457f
    f9e54e36315b30d491393e2f72d107bc118c888c6fd6d651deb0b1ce
    1acee99142287b7212c37775b7c5570211b37dd8cfbfd72a3971f7a0
    3568e3e644a831f77690a9a61c5fef62ebf77b40da39c6e6fe411e3c
    ebe34941383d3c8a58

C0 = 0x5e589df708c7b8273b715f5ae6296fcabdc383eea136793671418
     9d3a58412cb16f045c44eb563891ef1b02e075c489ffca0953b9dd4
     a6d782bc672846369d027eef3f96a4aa197a291e6e40e0f09e8f918
     8b9053c3ecc9ad1d0e0359e9d967c34399418d98bd335bfc70b5fe0
     5982aceb77bf828732cd7bca3845242474dcf72178f799f8b25dc07
     64a43cb86df60a7
C1 = 0xe52cceffaa6232c7412666bbaf27c581802fbd84e4ba2773a9798
     3bf047eb849

C = C0||C1 = 0x5e589df708c7b8273b715f5ae6296fcabdc383eea1367
             936714189d3a58412cb16f045c44eb563891ef1b02e075c
             489ffca0953b9dd4a6d782bc672846369d027eef3f96a4a
             a197a291e6e40e0f09e8f9188b9053c3ecc9ad1d0e0359e
             9d967c34399418d98bd335bfc70b5fe05982aceb77bf828
```

```
            732cd7bca3845242474dcf72178f799f8b25dc0764a43cb
            86df60a7e52cceffaa6232c7412666bbaf27c581802fbd8
            4e4ba2773a97983bf047eb849

===============================================================================
        EPOC-2 ( IFES-EPOC-EME3 ) Test Vector No.10 ( 1152 bits )
===============================================================================

******** EPOC-2 Private Key ********

p = 0xff6fe61c802466ebca41547185229da76ab23ca0523cfb63503a2e
    9419948481d26fc38914927b7133ab7e67c515923b
q = 0xfba6b13bcdf6880235e592970ea618d47fccafc760b1a2db7c3e52
    1cb4464443852a86d0ba39fe5157e0a5aa5c10d643
w = 0x3ca8779dc0c9b93528485ea15a0be6ad44dde0686f724e11ddced8
    89374d061873bc9fd9ddce2fbd2c43cacbfcbfaf80

******** EPOC-2 Public Key ********

Hash = SHA1
KDF = KDF2-SHA1
KDF' = KDF1-SHA1
StreamMode = 0
SC = SC1(BC=Camellia, BC.KeyLen=128)
l = 384 bits
l' = 960 bits
SC.KeyLen = 128 bits
n = 0xfa8bb28af9017e3a815eb7c25f370ae3accfc133cd806804143078
    f9f5a1b2f0e7b1df48fe7e2ed3a0f4650d9f816b6e0ce4c31330055f
    780768eb7909fe6d42ed1d09f548ec638917537200ff4e65489c9c61
    b3f2e42c13587300caa17c9c9567a88cb353819ca075682f1b991d7f
    8fd9126571d31c1746824acbe3c8611265692439cea8996dc72a3485
    cf596c590b
g = 0x02
h = 0x6f81fe2a320707ded0f021b0aedd2fd3d8169f0ca7d46a6dbb9f6e
    ffa9dc577cf3d10a6e2db2acec405ef9109616998da47cebb9b5a9b4
    512a6dd7e73214824fda6609e7b21c5202f1d4e5731726884a132bcf
    1078b2c57accb45443a6d4e53cfeed5de4a743dcabea297d5a69eee6
    f978ae5737ff5b7244228c739e6dbc24d5b4676f7ef233523fee2d27
    1470827aa8

******** EPOC-2 Encryption Data ********

M = 0x60336448e2321d360e1194d56a919785
L = ( null string )
seed = 0x4b963b925409302335ef677422e4ae0fefbb8dc57bea6dd3642
       b2f00fcda2fcb536cf41c24122ae73cd2c6496637d2
K = KDF(seed, SC.KeyLen) = 0xfb079a9eecbaca1db64073a3b52a64c4
C = SC.Encrypt(K, M) = 0x95c82ef8e442280c24941d5eab5a31630ae
                       c18cf43463a61fb7b5241cf75e92e
H = Hash(DB) = 0x7a33a145dd0b1d0695bbffdad1354193be408930
```

```
T = KDF'(H, rLen) = 0xe3ca74b5689ace69183b638782f20eae50f89e
                    308e3249127e0d772458bb68d81fe5aac4f1b559
                    6c33c861583f0d7746be3cb7a2c67fd359fcaace
                    62197ad99e6689b4f3abd3a23d6666ab9a294b9c
                    67cd46d2bad166fe03830f146bff48be7384c2cb
                    9ff64f9e49f987ed34f7cb3a2b6685797526b1e3
                    12
f = 0x4b963b925409302335ef677422e4ae0fefbb8dc57bea6dd3642b2f
    00fcda2fcb536cf41c24122ae73cd2c6496637d2
r = 0xe3ca74b5689ace69183b638782f20eae50f89e308e3249127e0d77
    2458bb68d81fe5aac4f1b5596c33c861583f0d7746be3cb7a2c67fd3
    59fcaace62197ad99e6689b4f3abd3a23d6666ab9a294b9c67cd46d2
    bad166fe03830f146bff48be7384c2cb9ff64f9e49f987ed34f7cb3a
    2b6685797526b1e312

C0 = 0x1e0e1ef2a8f5300255794b0cef078175afd71c46104cd82b3da92
     050fd307c04c5f256372212651e7b30f6248802aa691c353a585420
     9ff3b20b2290daf58ee4797d6964cd82b1732ac014b4a2ad3c02f60
     d298fbe82f2a05d373ee20a99ed414e98c4aa32c7e2d046927bdac6
     3b974d04a4916785d3d77586d759840f059fbc481fbe5f71eea716a
     98e06ca71879605
C1 = 0x95c82ef8e442280c24941d5eab5a31630aec18cf43463a61fb7b5
     241cf75e92e

C = C0||C1 = 0x1e0e1ef2a8f5300255794b0cef078175afd71c46104cd
             82b3da92050fd307c04c5f256372212651e7b30f6248802
             aa691c353a5854209ff3b20b2290daf58ee4797d6964cd8
             2b1732ac014b4a2ad3c02f60d298fbe82f2a05d373ee20a
             99ed414e98c4aa32c7e2d046927bdac63b974d04a491678
             5d3d77586d759840f059fbc481fbe5f71eea716a98e06ca
             7187960595c82ef8e442280c24941d5eab5a31630aec18c
             f43463a61fb7b5241cf75e92e

===============================================================================
        End of EPOC-2 ( IFES-EPOC-EME3 ) Test Vector
===============================================================================
```

C.9 Test vectors for *HIME(R)*

```
=====================================================================
               HIME(R) Test Vector No.1 (1023-bit)
=====================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1023-bit
p,q: 341-bit
d: 2

Public key
```

```
----------

n =               45 79 68 9f 45 3a 81 0f 87 bd 83 9e bd 99 3e a0
                  67 0e d9 06 dd 20 23 e6 69 51 7a fa 79 6a 77 9c
                  7f de 32 26 81 49 15 f7 7c 08 c1 ed fa 7c da 7d
                  ad be d0 51 66 11 b2 63 bd 4c 96 32 7d 5e 08 b4
                  03 a5 f9 eb 31 35 c7 87 d3 fe 5e ef 7f 7e ad 42
                  07 3a fa ad fa cb f0 36 7d 11 2e 08 b3 8f 56 4e
                  1a 6e c7 9f b7 6f 6a d6 94 9f 88 25 d8 7b 88 8d
                  9a 92 9f 0f ab 05 9d f7 04 75 08 6a 08 23 76 4f


Private key
-----------

p =               18 a0 e0 be 46 81 ae 1a 96 67 de e8 fe 53 8c 39
                  3f 47 a5 49 0e 14 aa 67 0a dc 80 2b 8a b2 8d d3
                  76 3d 07 d8 34 8a b3 23 2d 65 4f

q =               1d 52 60 8f 7d 37 84 55 85 ff 0a 67 cb 11 cf 2f
                  52 84 fc 04 b9 2f e4 b0 a5 37 16 55 c5 e1 6d 66
                  3e 6a 6b 52 8a b7 52 cb 50 42 af

Message to be encrypted
-----------------------

M =               fb e8 e3 3b 8a e5 35 a3 56 45 d8 04 89 75 8e ed
                  50 26 34 dc 5d a1 e7 84 71 a7 37 d9 84 72 81 19

Encoding parameters
-------------------

L =             (the empty string)

Step-by-step HEM1 encoding of M
-------------------------------

        check           = Hash.eval(L)
        seed            = random string of octets
        seed'           = the most significant KLen-bit cleared
                          seed
        DataBlockMask   = KDF(seed', ELen-Hash.len-1)
        MaskedDataBlock = DataBlock xor DataBlockMask
        SeedMask        = KDF(MaskedDataBlock, Hash.len+1)
        SeedMask'       = the most significant KLen-bit cleared
                          SeedMask
        MaskedSeed      = seed' xor SeedMask'

seed =            86 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                  32 12 6f a7 67
```

```
seed' =           06 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                  32 12 6f a7 67

check =           da 39 a3 ee 5e 6b 4b 0d 32 55 bf ef 95 60 18 90
                  af d8 07 09

DataBlock =       da 39 a3 ee 5e 6b 4b 0d 32 55 bf ef 95 60 18 90
                  af d8 07 09 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 01 fb e8 e3 3b 8a
                  e5 35 a3 56 45 d8 04 89 75 8e ed 50 26 34 dc 5d
                  a1 e7 84 71 a7 37 d9 84 72 81 19

DataBlockMask =   2e b9 b8 94 75 9c 38 f1 4f b1 ed d0 42 40 3b 89
                  66 f3 e2 63 75 d7 bf bd 19 eb 14 67 97 dc d7 c1
                  80 bd e2 40 ff 8c 22 16 e5 83 49 0a f0 19 af 5d
                  ec 1d 9c 51 0b f4 cf f1 05 11 8c 48 b5 3c fd 13
                  48 49 32 b7 7f 8f 81 6e 64 dc 39 70 57 63 ed d6
                  f7 4c 2e 4e 0b 1e bc bd 93 4d a6 e2 a0 29 5e 95
                  f5 0f 50 04 40 50 54 7e 6b d5 b1

MaskedDataBlock = f4 80 1b 7a 2b f7 73 fc 7d e4 52 3f d7 20 23 19
                  c9 2b e5 6a 75 d7 bf bd 19 eb 14 67 97 dc d7 c1
                  80 bd e2 40 ff 8c 22 16 e5 83 49 0a f0 19 af 5d
                  ec 1d 9c 51 0b f4 cf f1 05 11 8c 48 b5 3c fd 13
                  48 49 32 b7 7f 8f 81 6e 64 dc 38 8b bf 80 d6 5c
                  12 79 8d 18 4e c6 b8 34 e6 c3 4b b2 86 1d 82 c8
                  54 e8 d4 75 e7 67 8d fa 19 54 a8

SeedMask =        34 59 3e 07 a1 9a 32 d6 08 51 f6 22 c2 1d 90 c2
                  60 03 99 2e a9

SeedMask' =       34 59 3e 07 a1 9a 32 d6 08 51 f6 22 c2 1d 90 c2
                  60 03 99 2e a9

MaskedSeed =      32 c4 af 52 73 fc 66 12 49 98 ee 04 16 77 24 16
                  52 11 f6 89 ce

E =               32 c4 af 52 73 fc 66 12 49 98 ee 04 16 77 24 16
                  52 11 f6 89 ce f4 80 1b 7a 2b f7 73 fc 7d e4 52
                  3f d7 20 23 19 c9 2b e5 6a 75 d7 bf bd 19 eb 14
                  67 97 dc d7 c1 80 bd e2 40 ff 8c 22 16 e5 83 49
                  0a f0 19 af 5d ec 1d 9c 51 0b f4 cf f1 05 11 8c
                  48 b5 3c fd 13 48 49 32 b7 7f 8f 81 6e 64 dc 38
                  8b bf 80 d6 5c 12 79 8d 18 4e c6 b8 34 e6 c3 4b
                  b2 86 1d 82 c8 54 e8 d4 75 e7 67 8d fa 19 54 a8

The ciphertext
--------------
```

```
C =              15 73 07 76 08 64 8e c0 d3 66 b6 b5 78 7d 18 c5
                 0c 58 7f 42 ea 50 88 cc 04 c2 68 24 ea e4 8f cb
                 dd 0c c6 47 ce 93 5f 14 2b 8c eb ea b5 1f 78 4b
                 2d 9e 15 80 63 55 bb 9a ed ce 2e 23 a8 ec 7d 8b
                 a6 ae bf bb 1e 54 b7 64 e6 76 51 e9 c5 b6 e7 c2
                 e6 bc 38 fc c9 da 94 20 a9 64 d6 92 07 3a e1 8a
                 3c 66 61 f2 45 74 d5 5a 3e a4 ee b0 c2 73 ef f9
                 d9 2e 16 0f 6f d9 3c a4 74 8d 16 01 d9 36 4f e8

Step-by-step HEM1 decoding of E
-------------------------------

The intermediate values are the same as during HEM1 encoding of M.

=====================================================================
               HIME(R) Test Vector No.2 (1023-bit)
=====================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1023-bit
p,q: 341-bit
d: 2
L = (the empty string)

Public key
----------

n =              45 79 68 9f 45 3a 81 0f 87 bd 83 9e bd 99 3e a0
                 67 0e d9 06 dd 20 23 e6 69 51 7a fa 79 6a 77 9c
                 7f de 32 26 81 49 15 f7 7c 08 c1 ed fa 7c da 7d
                 ad be d0 51 66 11 b2 63 bd 4c 96 32 7d 5e 08 b4
                 03 a5 f9 eb 31 35 c7 87 d3 fe 5e ef 7f 7e ad 42
                 07 3a fa ad fa cb f0 36 7d 11 2e 08 b3 8f 56 4e
                 1a 6e c7 9f b7 6f 6a d6 94 9f 88 25 d8 7b 88 8d
                 9a 92 9f 0f ab 05 9d f7 04 75 08 6a 08 23 76 4f

Private key
-----------

p =              18 a0 e0 be 46 81 ae 1a 96 67 de e8 fe 53 8c 39
                 3f 47 a5 49 0e 14 aa 67 0a dc 80 2b 8a b2 8d d3
                 76 3d 07 d8 34 8a b3 23 2d 65 4f

q =              1d 52 60 8f 7d 37 84 55 85 ff 0a 67 cb 11 cf 2f
                 52 84 fc 04 b9 2f e4 b0 a5 37 16 55 c5 e1 6d 66
                 3e 6a 6b 52 8a b7 52 cb 50 42 af

Example 2.1
-----------
```

```
M =               d8 5a 93 45 a8 60 51 e7 30 71 62 00 56 b9 20 e2
                  19 00 58 55 a2 13 a0 f2 38 97 cd cd 73 1b 45 25
                  7c 77 7f e9

seed =            dd dd 87 71 fe c4 8b 83 a3 1e e6 f5 92 c4 cf d4
                  bc 88 17 4f 3b

C =               0e 38 31 4b 1e f6 49 fd d1 d6 1c 81 7b 21 d4 30
                  d5 b3 79 ca 3e 05 34 1d d9 53 34 d4 2a e9 7e 73
                  11 32 83 6b 71 52 91 2b 7b d2 b7 45 85 c3 c9 1f
                  c0 20 dd 34 8c fe d0 a3 f2 3c e1 4c 27 ca da 07
                  d1 8c cc a8 ad ff 6b 2c bb 85 48 2c ae 3e cb ac
                  6a 27 f9 5f f8 45 19 41 19 60 d7 1b 9e 8b 2c 34
                  0c 71 11 7c 0c 25 58 80 e9 2f 8c 02 62 51 0f 36
                  22 04 e1 9e fe 5c 73 a0 11 47 13 36 09 d2 d5 2f

Example 2.2
-----------

M =               da fb f0 38 e1 80 d8 37 c9 63 66 df 24 c0 97 b4
                  ab 0f ac 6b df 59 0d 82 1c 9f 10 64 2e 68 1a d0

seed =            cc 88 53 d1 d5 4d a6 30 fa c0 04 f4 71 f2 81 c7
                  b8 98 2d 82 24

C =               22 27 04 3b 95 59 a7 38 f8 36 5d 03 3e de 8b 51
                  62 a6 e8 c7 31 45 23 8a f6 8a 0c 8c e6 13 ab 16
                  9f a9 73 aa 32 df ec 25 15 a2 2d c8 3e f6 d0 5f
                  fa 1f 9b c0 1b 45 ae 9c da 1f 66 46 4b 90 3f 5f
                  1b 1e e0 24 57 30 49 5b db 90 2a ee b2 dd 33 51
                  19 8d 28 71 02 9e ba b7 9e 7c f3 34 49 45 62 64
                  95 4f e2 51 5e 66 ad 1c 6e d3 63 42 46 16 b1 fd
                  fb 89 5e 0b c0 9d e3 12 27 a2 ac 91 a2 31 99 d2

=====================================================================
               HIME(R) Test Vector No.3 (1023-bit)
=====================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1023-bit
p,q: 341-bit
d: 2
L = (the empty string)

Public key
----------

n =               64 46 ed 6c 4b a2 a1 ad e0 c3 6f ba 47 1c 02 8c
```

```
                  3a fc ba 00 3b 8f 8f 52 f2 1e c7 8f fb 42 33 83
                  4f a5 75 f4 ff 7d bb 1a be c8 b5 93 57 e9 69 dc
                  30 44 a5 7d 3a b9 1d bd 1a 49 2e 87 fd f7 57 21
                  c0 52 43 4f 1a 00 63 f6 c5 18 bc d1 35 93 3a f4
                  e3 c9 7b 29 30 c0 5b d1 1c bb c2 06 b4 fd b3 50
                  eb e0 5b d4 38 36 a0 ec b6 5b af f2 d0 da 1a 47
                  08 4e 8d f2 b7 a4 35 f1 6b 9e 5f 3d ad b6 62 97

Private key
-----------

p =               1c a0 e0 be 46 91 ae 1a 96 67 de e8 fe 53 8c 39
                  3f 47 a5 49 0e 14 aa 67 0a dc 80 2b 8a b2 8d d3
                  76 3d 07 d8 34 8a b3 23 2d 66 c7

q =               1f 52 60 8f 7d 37 84 55 85 ff 0a 6b cb 11 cf 2f
                  52 84 fc 04 b9 2f e4 b0 a5 37 16 55 c5 e1 6d 66
                  3e 6a 6b 52 8a b7 52 cb 50 47 c7

Example 3.1
-----------

M =               ab 3c d9 d8 8d 98 40 3b 38 b4 09 95 fd 6f f4 1a
                  1a cc 8a da

seed =            6a 90 87 4e ef ce 8f 2c cc 20 e4 f2 74 1f b0 a3
                  3a 38 48 ae c9

C =               5d c8 e9 ec 7c 33 f6 a4 5e 63 26 43 e8 b6 57 32
                  38 76 b7 17 ac dd 30 a2 fc f3 fd 8d 22 35 f1 ec
                  3e 75 ac e1 c8 e8 4e 2e dd 5a b9 6f 9e bd 48 68
                  79 3f 20 f2 af 90 60 57 da b1 2e f1 d3 07 5e 0d
                  b3 39 39 f6 50 50 1c 27 8f 59 25 4d 3d 64 81 3d
                  8c da e2 98 37 c5 76 65 53 43 15 eb 54 e9 3f ac
                  b3 53 7b 95 b7 c0 0e 92 36 38 f9 7b 4b 1d 6d 5e
                  0a bb a3 4c 8a b5 13 4b 2a 7b 47 4c 37 af 01 c3

Example 3.2
-----------

M =               ef f2 9d da 4f 2d 51 64 73 f1 10 64 c9 96 fa 26
                  56 d2 c3 c0 93 44 05 af bc de 22 2d 9d 31 7c f2
                  39 fe 8a 16

seed =            95 29 7b 0f 95 a2 fa 67 d0 07 07 d6 09 df d4 fc
                  05 c8 9d af c2

C =               41 57 55 2e 9d ab ff 3c 08 54 6e 5b c8 0d e0 07
                  c9 64 7a 09 bc e5 9c a0 8b 9b f9 f6 be 14 58 e0
                  8b 0a 1d 47 ff 2c 7d 5f 42 75 30 32 3b 1c ce dc
                  c3 ac c3 8e 43 bc ca bc 57 39 4f 29 8d 8e dd cd
                  7d 26 bf 72 1f 6a 3f c5 55 19 c9 04 cd eb 94 f4
                  3f c8 15 a6 fb fc 43 0a 0e 25 05 14 5d ac 22 b6
                  c9 a5 57 27 89 ca 0e 52 4d fb 87 91 71 fe 80 f7
                  d8 78 c4 3a b0 d7 7c 3e 66 39 c6 ac 28 c3 85 8f


======================================================================
               HIME(R) Test Vector No.4 (1344-bit)
======================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1344-bit
p,q: 448-bit
d: 2

Public key
----------

n =               87 e5 d1 89 c4 66 b7 98 0f 48 50 1f 36 10 80 2d
                  86 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                  32 12 6f 6a 0d 8c 39 22 f2 6b 35 42 80 1b 0b cf
                  fb e8 e3 3b 8a e5 35 a3 56 45 d8 04 89 75 8e ed
                  50 26 34 dc 5d a1 e7 84 71 a7 37 d9 84 72 81 19
                  92 d2 09 25 dc 4b a8 1f 7f ee 3a f5 71 cb b3 f2
                  c6 68 a6 93 56 c0 72 aa ba 4b 3e 4d 19 9a e1 84
                  07 33 e8 e4 df 9f 43 89 c4 f0 4c fc ad 99 63 67
                  70 cb d8 9c 70 a7 87 eb 73 5f 91 f6 cb fc 5a 22
                  b9 72 63 1d e2 28 3d 1e 84 9c 82 0f f6 2a d9 42
                  c4 c5 fd ea 0b 8a 66 63


Private key
-----------

p =               d1 ad c7 92 77 f0 e7 16 de bf 8f a0 42 be 80 8c
                  55 cc da 53 34 2c fd a4 60 d6 d5 68 e3 85 b7 89
                  36 b3 28 ad fa 67 9b 80 b0 ec 94 5c 9c da 67 1b
                  63 da 57 f9 e4 66 87 03

q =               ca 92 db 5a 3a 53 22 0a 4a 92 1e e6 e9 af da ce
                  12 7e a7 67 0e df df c2 68 42 a1 ad 62 79 05 1b
                  e6 82 00 c0 09 83 db a2 36 ab e1 6d 37 41 45 cd
                  b1 f6 da b2 cc 81 d8 0b


Message to be encrypted
-----------------------

M =               fb e8 e3 3b 8a e5 35 a3 56 45 d8 04 89 75 8e ed
```

```
                  50 26 34 dc 5d a1 e7 84 71 a7 37 d9 84 72 81 19
Encoding parameters
-------------------

L =             (the empty string)

Step-by-step HEM1 encoding of M
-------------------------------

        check           = Hash.eval(L)
        seed            = random string of octets
        seed'           = the most significant KLen-bit cleared
                          seed
        DataBlockMask   = KDF(seed', ELen-Hash.len-1)
        MaskedDataBlock = DataBlock xor DataBlockMask
        SeedMask        = KDF(MaskedDataBlock, Hash.len+1)
        SeedMask'       = the most significant KLen-bit cleared
                          SeedMask
        MaskedSeed      = seed' xor SeedMask'

seed =            86 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                  32 12 6f a7 67

seed' =           06 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                  32 12 6f a7 67

check =           da 39 a3 ee 5e 6b 4b 0d 32 55 bf ef 95 60 18 90
                  af d8 07 09

DataBlock =       da 39 a3 ee 5e 6b 4b 0d 32 55 bf ef 95 60 18 90
                  af d8 07 09 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                  00 00 01 fb e8 e3 3b 8a e5 35 a3 56 45 d8 04 89
                  75 8e ed 50 26 34 dc 5d a1 e7 84 71 a7 37 d9 84
                  72 81 19

DataBlockMask =   2e b9 b8 94 75 9c 38 f1 4f b1 ed d0 42 40 3b 89
                  66 f3 e2 63 75 d7 bf bd 19 eb 14 67 97 dc d7 c1
                  80 bd e2 40 ff 8c 22 16 e5 83 49 0a f0 19 af 5d
                  ec 1d 9c 51 0b f4 cf f1 05 11 8c 48 b5 3c fd 13
                  48 49 32 b7 7f 8f 81 6e 64 dc 39 70 57 63 ed d6
                  f7 4c 2e 4e 0b 1e bc bd 93 4d a6 e2 a0 29 5e 95
                  f5 0f 50 04 40 50 54 7e 6b d5 b1 7a 0d 04 a8 ca
                  27 26 10 33 dc 95 e2 e4 5b 32 08 d1 7e 13 90 61
                  9b 4b d7 c4 15 04 ea 4e fd fc b7 24 dc 0e 75 01
                  a3 19 f9
```

```
MaskedDataBlock = f4 80 1b 7a 2b f7 73 fc 7d e4 52 3f d7 20 23 19
                  c9 2b e5 6a 75 d7 bf bd 19 eb 14 67 97 dc d7 c1
                  80 bd e2 40 ff 8c 22 16 e5 83 49 0a f0 19 af 5d
                  ec 1d 9c 51 0b f4 cf f1 05 11 8c 48 b5 3c fd 13
                  48 49 32 b7 7f 8f 81 6e 64 dc 39 70 57 63 ed d6
                  f7 4c 2e 4e 0b 1e bc bd 93 4d a6 e2 a0 29 5e 95
                  f5 0f 50 04 40 50 54 7e 6b d5 b1 7a 0d 04 a8 ca
                  27 26 11 c8 34 76 d9 6e be 07 ab 87 3b cb 94 e8
                  ee c5 3a 94 33 30 36 13 5c 1b 33 55 7b 39 ac 85
                  d1 98 e0

SeedMask =        6d ad 57 05 63 99 d1 1f 44 a1 20 bd 1a ff 15 47
                  31 f0 79 59 ea

SeedMask' =       6d ad 57 05 63 99 d1 1f 44 a1 20 bd 1a ff 15 47
                  31 f0 79 59 ea

MaskedSeed =      6b 30 c6 50 b1 ff 85 db 05 68 38 9b ce 95 a1 93
                  03 e2 16 fe 8d

E =               6b 30 c6 50 b1 ff 85 db 05 68 38 9b ce 95 a1 93
                  03 e2 16 fe 8d f4 80 1b 7a 2b f7 73 fc 7d e4 52
                  3f d7 20 23 19 c9 2b e5 6a 75 d7 bf bd 19 eb 14
                  67 97 dc d7 c1 80 bd e2 40 ff 8c 22 16 e5 83 49
                  0a f0 19 af 5d ec 1d 9c 51 0b f4 cf f1 05 11 8c
                  48 b5 3c fd 13 48 49 32 b7 7f 8f 81 6e 64 dc 39
                  70 57 63 ed d6 f7 4c 2e 4e 0b 1e bc bd 93 4d a6
                  e2 a0 29 5e 95 f5 0f 50 04 40 50 54 7e 6b d5 b1
                  7a 0d 04 a8 ca 27 26 11 c8 34 76 d9 6e be 07 ab
                  87 3b cb 94 e8 ee c5 3a 94 33 30 36 13 5c 1b 33
                  55 7b 39 ac 85 d1 98 e0

The ciphertext
--------------

C =               79 67 bc c3 42 fc bb 72 e2 75 e6 68 73 bf 3c 68
                  c9 81 05 df cd 08 82 28 0d cd 0a 45 26 1c 7d 68
                  ee 46 79 6b 49 b3 66 74 81 9c c8 84 05 98 5a 44
                  8b f9 17 4d 93 c5 ce d4 fc b2 00 ff 65 e5 fd 58
                  cb 3c f1 1f 65 d8 3c 24 f2 78 c6 ba 88 44 79 32
                  86 af 8e b3 9b f5 9d 02 65 21 b8 2b 09 c5 40 05
                  92 1e a7 4c 56 87 57 4b 3a 9f c1 8b 86 c2 90 eb
                  21 34 17 bb 3b e2 b4 b4 a6 be d1 54 d8 6f 1e b4
                  d6 be cd 65 05 a6 00 23 40 31 d8 c3 a7 ce c1 03
                  b8 14 17 30 37 8a 33 05 c6 3b 1d bd ed e7 bf ac
                  6f 8d e8 22 34 90 db 7a

Step-by-step HEM1 decoding of E
-------------------------------

The intermediate values are the same as during HEM1 encoding of M.
```

```
======================================================================
               HIME(R) Test Vector No.5 (1344-bit)
======================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1344-bit
p,q: 448-bit
d: 2
L = (the empty string)

Public key
----------

n =                  87 e5 d1 89 c4 66 b7 98 0f 48 50 1f 36 10 80 2d
                     86 9d 91 55 d2 66 54 c4 41 c9 18 26 d4 6a b4 d4
                     32 12 6f 6a 0d 8c 39 22 f2 6b 35 42 80 1b 0b cf
                     fb e8 e3 3b 8a e5 35 a3 56 45 d8 04 89 75 8e ed
                     50 26 34 dc 5d a1 e7 84 71 a7 37 d9 84 72 81 19
                     92 d2 09 25 dc 4b a8 1f 7f ee 3a f5 71 cb b3 f2
                     c6 68 a6 93 56 c0 72 aa ba 4b 3e 4d 19 9a e1 84
                     07 33 e8 e4 df 9f 43 89 c4 f0 4c fc ad 99 63 67
                     70 cb d8 9c 70 a7 87 eb 73 5f 91 f6 cb fc 5a 22
                     b9 72 63 1d e2 28 3d 1e 84 9c 82 0f f6 2a d9 42
                     c4 c5 fd ea 0b 8a 66 63

Private key
-----------

p =                  d1 ad c7 92 77 f0 e7 16 de bf 8f a0 42 be 80 8c
                     55 cc da 53 34 2c fd a4 60 d6 d5 68 e3 85 b7 89
                     36 b3 28 ad fa 67 9b 80 b0 ec 94 5c 9c da 67 1b
                     63 da 57 f9 e4 66 87 03

q =                  ca 92 db 5a 3a 53 22 0a 4a 92 1e e6 e9 af da ce
                     12 7e a7 67 0e df df c2 68 42 a1 ad 62 79 05 1b
                     e6 82 00 c0 09 83 db a2 36 ab e1 6d 37 41 45 cd
                     b1 f6 da b2 cc 81 d8 0b

Example 5.1
-----------

M =                  ab 3c d9 d8 8d 98 40 3b 38 b4 09 95 fd 6f f4 1a
                     1a cc 8a da

seed =               6a 90 87 4e ef ce 8f 2c cc 20 e4 f2 74 1f b0 a3
                     3a 38 48 ae c9

C =                  71 9d 83 07 f8 75 1e b4 51 be d2 20 28 d0 6a 2e
```

```
                  e4 25 16 fe 5b 3d bd e2 3f a5 50 85 6f 06 7e 2c
                  3b f7 0a ca ae 1a 8d 36 39 13 36 4a e3 7f 9f 74
                  b1 ed 53 b7 81 97 95 9a 9b e3 c4 fd 33 46 5c 4b
                  55 53 b5 a5 ba 21 07 36 7e ba ec f9 9f 2a 15 80
                  6c 9b 8a d0 f5 70 2e 61 2c 12 26 6a 56 90 7c 9e
                  91 e9 2a d7 b2 3a 14 43 fa 95 94 b5 23 b5 96 0f
                  d5 5d 6d 8f 1b f9 fe fc bf dd 72 cf 3c bd 21 5f
                  7c dc 01 51 c6 50 04 b1 f6 ae eb 4c a8 b9 bc f4
                  f6 11 9c fd d9 09 34 14 8f 1e af 1d 54 6e 63 af
                  41 27 74 54 5f d8 19 39

Example 5.2
-----------

M =               ef f2 9d da 4f 2d 51 64 73 f1 10 64 c9 96 fa 26
                  56 d2 c3 c0 93 44 05 af bc de 22 2d 9d 31 7c f2
                  39 fe 8a 16

seed =            95 29 7b 0f 95 a2 fa 67 d0 07 07 d6 09 df d4 fc
                  05 c8 9d af c2

C =               1b 17 e0 25 92 33 25 83 0b 77 1f cf 9e 17 53 57
                  5b 6e 07 f2 59 5a c4 50 8a fd 00 58 3c b4 32 9c
                  95 7a 46 a1 4c 21 68 99 bd 35 93 d9 96 8a 1f 7f
                  e8 f4 bb 9a fe 35 89 01 5b 57 c9 0a e5 a0 00 ec
                  57 9c 0c e8 1e b4 fc 51 81 d9 fe ba e0 35 38 bd
                  60 f1 4e 6c fb 04 28 e7 43 d6 0f 26 ce a6 40 30
                  0b 7d 1f 08 d8 18 22 94 ed 3f 53 1c ab eb ee b4
                  89 f7 37 85 88 be 1b ba 8a 45 a1 ee 5e 69 fd ed
                  31 d4 18 65 5a a5 c8 ae 9d 17 4b fc c2 31 68 cb
                  e7 d5 f4 1e 5b 55 ee f1 f1 9e 1c c6 cd 9e a1 31
                  fe d9 5e d0 56 52 cc 8b

=====================================================================
               HIME(R) Test Vector No.6 (1344-bit)
=====================================================================

Hash.eval: SHA1
Hash.len: 20
KDF: KDF1-SHA1
n: 1344-bit
p,q: 448-bit
d: 2
L = (the empty string)

Public key
----------

n =               d1 10 0b fe 0e 2c f5 d0 76 12 57 dc 34 e4 13 bb
                  02 21 f7 c4 27 cd ee 41 d1 b0 78 28 ac b3 c8 80
                  cc b5 26 db ea d4 39 58 fb 71 07 e6 28 1b 73 0f
```

```
                        6c f1 64 bc 28 9f df b9 f2 9b ae 88 e4 e3 a3 38
                        c8 45 1e 70 c3 0b b1 f1 44 87 e8 49 9b 67 55 11
                        e5 ac 3d ff 50 91 05 3b 46 59 cc 3b 23 c2 99 a7
                        0f a3 ed ea e4 63 45 6e e8 35 99 90 68 c3 a4 5e
                        b7 45 47 7c 79 d8 ed ac c5 8a cc 16 9a 67 7d 96
                        ab f9 4f 7a 1b 55 3b 56 35 c0 62 37 21 0d 9d 48
                        63 c1 f1 27 64 15 b9 ba 1c ae a0 73 d2 f9 3d 32
                        11 8a 73 b2 61 bd 13 1b

Private key
-----------

p =                     f1 ad c7 92 77 f0 e7 16 de bf 8f a0 42 be 80 8c
                        55 cc da 53 34 2c fd a4 60 d6 d5 68 e3 85 b8 89
                        36 b3 28 ad fa 67 9b 80 b0 ec 94 5c 9c da 67 1b
                        63 da 57 f9 e4 cc 6f 6b

q =                     ea 92 db 5a 3a 53 22 0a 4a 92 1e e6 e9 af da ce
                        12 7e b7 67 0e df df c2 68 42 a1 ad 62 79 05 1b
                        e6 82 00 c0 09 83 db a2 36 ab e1 6d 37 41 45 cd
                        b1 f6 da b2 cc 8a 2e 73


Example 6.1
-----------

M =                     d8 5a 93 45 a8 60 51 e7 30 71 62 00 56 b9 20 e2
                        19 00 58 55 a2 13 a0 f2 38 97 cd cd 73 1b 45 25
                        7c 77 7f e9

seed =                  dd dd 87 71 fe c4 8b 83 a3 1e e6 f5 92 c4 cf d4
                        bc 88 17 4f 3b

C =                     6b ab 8d 82 90 ec 05 7c d3 e8 b2 7f e9 e5 38 d1
                        25 cd c6 13 38 7d e4 e5 f0 df 23 24 fe 58 3d 91
                        91 79 71 f5 1a 93 ae 13 29 9d 47 5d 4c bc 45 be
                        9a 8d f1 71 5c 32 c6 cc da 85 ea b6 ca ed da 1b
                        92 48 42 a8 4a f9 aa 40 9f 02 97 f9 73 74 b2 71
                        18 18 85 07 9e c4 02 1a 95 f2 18 08 28 69 bc 00
                        9d da c2 45 b3 f7 fe be 58 07 d3 65 67 59 d4 b2
                        14 f3 d2 1a e1 e5 10 49 90 f8 7d e5 70 02 13 87
                        96 b0 cc b9 15 3b 6b 2a 0f 25 52 90 a1 d4 1c 44
                        45 c1 07 ae 90 da 54 fb 3b b4 44 88 fc d6 1e 26
                        2b a7 62 ce a8 7f df 91

Example 6.2
-----------

M =                     da fb f0 38 e1 80 d8 37 c9 63 66 df 24 c0 97 b4
                        ab 0f ac 6b df 59 0d 82 1c 9f 10 64 2e 68 1a d0
```

```
seed =            cc 88 53 d1 d5 4d a6 30 fa c0 04 f4 71 f2 81 c7
                  b8 98 2d 82 24

C =               b6 dd 2f 33 ee 04 52 1f a9 17 d3 30 9e f1 2f a1
                  cb 93 de 75 4c fe a3 b1 bf e7 dd 9c a4 51 44 85
                  bf dc 57 31 c1 6d c2 78 7c 14 6d f6 f4 98 68 45
                  b8 ce aa c6 59 cf 42 12 f9 31 07 4c e6 e5 26 9e
                  62 96 06 46 db 64 88 12 42 fe 7f c7 6b 7a ed f9
                  0d 30 ca 8f 03 62 23 69 25 cb ce 11 d9 4a 76 e4
                  6e c7 bf c9 e7 b6 02 a0 f5 32 73 3b a8 b1 e6 f5
                  7c 07 45 28 bf b2 df c0 3c d7 1f 7f d6 3b 7e 4b
                  66 68 f5 89 e4 c8 d8 df 3c 0f 40 c1 04 ce b5 7b
                  6e 45 06 e9 a1 f5 bc 79 c1 40 bc e8 f2 11 dd 85
                  96 43 81 a6 ef d5 14 7b

=========================================================================
                      End of HIME(R) Test Vector
=========================================================================
```

References

[ABR99] M. Abdalla, M. Bellare, and P. Rogaway. DHAES: an encryption scheme based on the Diffie-Hellman problem. Cryptology ePrint Archive, Report 1999/007, 1999. `http://eprint.iacr.org`.

[ABR01] M. Abdalla, M. Bellare, and P. Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Topics in Cryptology – CT-RSA 2001*, pages 143–158, 2001. Springer LNCS 2045.

[BBM00] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: security proofs and improvements. In *Advances in Cryptology–Eurocrypt 2000*, 2000.

[BDJR97] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: analysis of the DES modes of operation. In *38th Annual Symposium on Foundations of Computer Science*, 1997.

[BDPR98] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In *Advances in Cryptology–Crypto '98*, pages 26–45, 1998.

[BKR94] M. Bellare, J. Kilian, and P. Rogaway. On the security of cipher block chaining. In *Advances in Cryptology—Crypto '94*, pages 341–358, 1994.

[Bon98] D. Boneh. The Decision Diffie-Hellman Problem. In *Ants-III*, pages 48–63, 1998. Springer LNCS 1423.

[BR93] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[BR94] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology—Eurocrypt '94*, pages 92–111, 1994.

[BSS99] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

[Can00] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. `http://eprint.iacr.org`.

[CGH98] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited. In *30th Annual ACM Symposium on Theory of Computing*, pages 209–218, 1998.

[CS98] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology–Crypto '98*, pages 13–25, 1998.

[CS01] R. Cramer and V. Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. Cryptology ePrint Archive, Report 2001/108, 2001. `http://eprint.iacr.org`.

[DDN91] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In *23rd Annual ACM Symposium on Theory of Computing*, pages 542–552, 1991.

[DDN98] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography, 1998. Manuscript (updated, full length version of STOC paper).

[DDN00] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. *SIAM J. Comput.*, 30(2):391–437, 2000.

[Den02] A. Dent. An implementation attack against the EPOC-2 public-key cryptosystem. *Electronics Letters*, 38(9):412, 2002.

[ElG85] T. ElGamal. A public key cryptosystem and signature scheme based on discrete logarithms. *IEEE Trans. Inform. Theory*, 31:469–472, 1985.

[FO99] E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *Advances in Cryptology–Crypto '99*, pages 537–554, 1999.

[FOPS01] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In *Advances in Cryptology–Crypto 2001*, pages 260–274, 2001.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[HIM02] Self-evaluation report: HIME(R) cryptosystem, Dec. 2002. Available from `http://www.sdl.hitachi.co.jp/crypto/hime/index.html`.

[Len91] H. W. Lenstra. Finding isomorphisms between finite fields. *Math. Comp.*, 56:329–347, 1991.

[Man01] J. Manger. A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS # 1 v2.0. In *Advances in Cryptology–Crypto 2001*, pages 230–238, 2001.

[MW00] U. Maurer and S. Wolf. The Diffie-Hellman protocol. *Designs, Codes, and Cryptography*, 19:147–171, 2000.

[NR97] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th Annual Symposium on Foundations of Computer Science*, 1997.

[NSS01] M. Nishioka, H. Satoh, and K. Sakuri. Design and analysis of fast provably secure public-key cryptosystems based on a modular squaring. In *Proc. ICISC 2001, LNCS 2288*, pages 81–102, 2001.

[NY90] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd Annual ACM Symposium on Theory of Computing*, pages 427–437, 1990.

[OP01] T. Okamoto and D. Pointcheval. The gap-problems: a new class of problems for the security of cryptographic schemes. In *Proc. 2001 International Workshop on Practice and Theory in Public Key Cryptography (PKC 2001)*, 2001.

[OU98] T. Okamoto and S. Uchiyama. A new public-key cryptosystem as secure as factoring. In *Advances in Cryptology–Eurocrypt '98*, pages 308–318, 1998.

[RS91] C. Rackoff and D. Simon. Noninteractive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology–Crypto '91*, pages 433–444, 1991.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[Sho97] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology–Eurocrypt '97*, pages 256–266, 1997.

[Sho01a] V. Shoup. OAEP reconsidered. In *Advances in Cryptology–Crypto 2001*, pages 239–259, 2001.

[Sho01b] V. Shoup. A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112, 2001. `http://eprint.iacr.org/`.

[Vau02] S. Vaudenay. Security flaws induced by CBC padding - applications to SSL, IPSEC, WTLS. In *Advances in Cryptology–Eurocrypt 2002*, 2002.