RSAES-OAEP Encryption Scheme

Algorithm specification and supporting documentation

RSA Laboratories
RSA Security Inc.
20 Crosby Drive
Bedford, MA 01730 USA

rsa-labs@rsasecurity.com

This document is a revised version of RSA Laboratories' submission of the RSAES-OAEP encryption scheme to the NESSIE project. The document contains an algorithm specification of the RSAES-OAEP encryption scheme together with supporting documentation including a security discussion. The document is for informational purposes only and does not specify a standard of any kind. See the latest version of PKCS #1 (www.rsalabs.com/pkcs/pkcs-1) for implementation guidelines.

Comments on the document are encouraged. E-mail correspondence to

rsa-labs@rsasecurity.com .

Chapter 1
Description of RSAES-OAEP

1.1 Introduction

RSAES-OAEP is a public-key encryption scheme combining the RSA algorithm [39] with the *Optimal Asymmetric Encryption Padding* (OAEP) method. The inventors of RSA are Ronald L. Rivest, Adi Shamir, and Leonard Adleman, while the inventors of OAEP are Mihir Bellare and Phillip Rogaway [4], with enhancements by Don B. Johnson and Stephen M. Matyas [27].

The RSA algorithm is discussed in Section 1.2, while the EME-OAEP encoding method is given in Section 1.3. The RSAES-OAEP encryption scheme is defined in Section 1.4. Note that the greater part of Sections 1.2-1.4 is taken from PKCS #1 v2.0 [37].

1.1.1 Notation

The following notation will be used.

An *octet* is the eight-bit representation of an integer with the leftmost bit being the most significant bit; this integer is the *value* of the octet.

An *octet string* is an ordered sequence of octets, where the first octet is the leftmost.

GCD(a,b)	greatest common divisor of the two nonnegative integers a and b
LCM(a,b)	least common multiple of the two nonnegative integers a and b
$\lfloor x \rfloor$	the real number x rounded down to the closest integer
$\lceil x \rceil$	the real number x rounded up to the closest integer
$X \| Y$	concatenation of octet strings X and Y
$X \oplus Y$	bitwise exclusive-or of octet strings X and Y

1.1.2 Data conversion primitives

Throughout this document, messages are octet strings, while the RSA primitives act on integers. For this reason, we must specify how to convert an octet string into a nonnegative integer and vice versa. This is done via two primitives I2OSP and OS2IP. I2OSP (Integer-to-Octet-String primitive) produces an octet string of a desired length from an integer; the string may be viewed as the base-256 representation of the integer. OS2IP (Octet-String-to-Integer primitive) is the inverse of I2OSP.

1.1.2.1 I2OSP

$\mathrm{I2OSP}(x, l)$

Input:	x	nonnegative integer to be converted
	l	intended length of the resulting octet string
Output:	X	corresponding octet string of length l
Errors:	"integer too large"	

1. If $x \geq 256^l$, output "integer too large" and stop.

2. Write the integer x in its unique l-digit representation base 256:

$$x = x_{l-1}256^{l-1} + x_{l-2}256^{l-2} + \ldots + x_1 256 + x_0,$$

where $0 \leq x < 256$ (one or more leading digits will be zero if $x < 256^{l-1}$).

3. Let the octet X_i have the integer value x_{l-1} for $1 \leq i \leq l$. Output the octet string $X = X_1 X_2 \ldots X_l$.

1.1.2.2 OS2IP

$\mathrm{OS2IP}(X)$

Input:	X	octet string to be converted
Output:	x	corresponding nonnegative integer

1. Let $X_1 X_2 \ldots X_n$ be the octets of X from first to last, and let X_i have the integer value x_{l-i} for $1 \leq i \leq l$.

2. Let $x = x_{l-1}256^{l-1} + x_{l-2}256^{l-2} + \ldots + x_1 256 + x_0$.

3. Output x.

1.2 The RSA algorithm

The RSA algorithm was invented by Ronald L. Rivest, Adi Shamir, and Leonard Adleman in 1977. The security of the algorithm is based on the hardness of factoring a large composite number and computing e^{th} roots modulo a composite number for a specified odd integer e.

An RSA public key consists of a pair (n, e) of integers, where n is the *modulus* and e is the *public exponent*. The modulus n is a large composite number (a bit length of at least 1024 is the current recommended size), while the public exponent e is normally a small prime such as 3, 17, or 65537. In this specification, the modulus is the product of two distinct primes. For a discussion of the case of three or more prime factors (so-called "multi-prime" RSA), see Appendix A at the end of this document.

An RSA private key may have one of two different representations. Both representations contain information about an integer d satisfying

$$(x^e)^d \equiv x \pmod{n}$$

for all integers x. This means that the private key can be used to solve the equation

$$x^e \equiv c \pmod{n}$$

in x, i.e., compute the e^{th} root of c modulo n.

The RSA encryption primitive RSAEP takes as input the public key (n, e) and a positive integer $m < n$ (a *message representative*) and returns the integer $c = m^e \bmod n$. The RSA decryption primitive RSADP takes as input the private key and an integer $c < n$ (a *ciphertext representative*) to return the integer $m = c^d \bmod n$, where d is an integer with the above specified properties.

The public key is defined in Section 1.2.1, while the private key is defined in Section 1.2.2. The RSA primitives are described in detail in Subsections 1.2.3 and 1.2.4; they coincide with the corresponding primitives in PKCS #1 v2.0 [37], IEEE Std 1363-2000 [25], and the draft ANSI X9.44 [1]. For a suggested key generation algorithm, see Section 1.5. For the mathematical background of the RSA algorithm, see Appendix B.

1.2.1 RSA public key

For the purposes of this document, an RSA public key consists of two components:

- n — the modulus, a nonnegative integer
- e — the public exponent, a nonnegative integer

In a *valid RSA public key*, the modulus n is a product of two distinct odd primes p and q, and the public exponent e is an integer between 3 and $n - 1$ satisfying $\text{GCD}(e, p-1) = \text{GCD}(e, q-1) = 1$.

1.2.2 RSA private key

For the purposes of this document, an RSA private key may have either of two representations.

1. The first representation consists of the pair (n, d), where the components have the following meanings:

n	the modulus, a nonnegative integer
d	the private exponent, a nonnegative integer

In a *valid RSA private key* with this representation, the modulus n is the same as in the corresponding public key and is the product of two odd primes p and q, and the private exponent d is a positive integer less than n satisfying

$$e \cdot d \equiv 1 \pmod{\mathrm{LCM}(p-1, q-1)},$$

where e is the corresponding public exponent.

2. The second representation consists of a quintuple $(p, q, dP, dQ, qInv)$, where the components have the following meanings:

p	the first factor, a nonnegative integer
q	the second factor, a nonnegative integer
dP	the first factor's exponent, a nonnegative integer
dQ	the second factor's exponent, a nonnegative integer
$qInv$	the CRT coefficient (see Remark 2 below), a nonnegative integer

In a valid RSA private key with this representation, the two factors p and q are the prime factors of the modulus n, the exponents dP and dQ are positive integers less than p and q respectively satisfying

$$\begin{aligned} e \cdot dP &\equiv 1 \pmod{p-1}; \\ e \cdot dQ &\equiv 1 \pmod{q-1}, \end{aligned}$$

and the CRT coefficient $qInv$ is a positive integer less than p satisfying

$$q \cdot qInv \equiv 1 \pmod{p}.$$

Remarks.

1. The parameters in either of the representations can be efficiently computed via the Extended Euclidean Algorithm (see IEEE Std 1363-2000 [25], A.2.2).

2. The second representation is based on the Chinese Remainder Theorem (CRT); the benefit of applying CRT to RSA operations was observed by Quisquater and Couvreur [38]. See Appendices I and II for more information on CRT.

3. In IEEE Std 1363-2000, a third representation of the private key is defined. This representation consists of a triplet (p, q, d), where p and q are the prime factors of n and d is the private exponent. This representation is closely related to the second representation above in that the parameters $dP, dQ, dInv$ are easily derived from (p, q, d).

1.2.3 RSAEP

RSAEP$((n, e), m)$

Input:	(n, e)	RSA public key
	m	message representative, an integer between 0 and $n - 1$
Output:	c	ciphertext representative, an integer between 0 and $n - 1$
Errors:	"message representative out of range"	

Assumption: public key (n, e) is valid

1. If the message representative m is not between 0 and $n - 1$, output "message representative out of range" and stop.

2. Let $c = m^e \bmod n$.

3. Output c.

1.2.4 RSADP

RSADP(K, c)

Input: K RSA private key, where K has one of the following forms:

1. a pair (n, d)
2. a quintuple $(p, q, dP, dQ, qInv)$

c ciphertext representative, an integer between 0 and $n - 1$

Output: m message representative, an integer between 0 and $n - 1$

Errors: "ciphertext representative out of range"

Assumption: private key K is valid

1. If the ciphertext representative c is not between 0 and $n - 1$, output "ciphertext representative out of range" and stop.

2. If the first form (n, d) of K is used:

   2.1 Let $m = c^d \bmod n$.

   Else, if the second form $(p, q, dP, dQ, qInv)$ of K is used:

   2.2 Let $m_1 = c^{dP} \bmod p$.

   2.3 Let $m_2 = c^{dQ} \bmod q$.

   2.4 Let $h = (m_1 - m_2) \cdot qInv \bmod p$.

   2.5 Let $m = m_2 + q \cdot h$.

3. Output m.

1.3 The EME-OAEP encoding method

An encoding method for encryption consists of an *encoding operation* and a *decoding operation*. In this document, the encoding method is EME-OAEP, which was invented by Mihir Bellare and Phillip Rogaway in 1994, with enhancements given by Don B. Johnson and Stephen M. Matyas in 1996. The encoding operation EME-OAEP-ENCODE (Section 1.3.1) maps a message M to an encoded message EM of a specified length using some "encoding parameters" (an octet string P), a hash function, and a mask generation function. The decoding operation EME-OAEP-DECODE (Section 1.3.2) maps an encoded message EM back to a message; the encoding and decoding operations are inverses.

Assuming that the mask generation function in EME-OAEP has appropriate properties, and the key size is sufficiently large, RSAES-OAEP is semantically secure against adaptive chosen ciphertext attacks; see Section 2.2 of this document for more information.

The EME-OAEP encoding operation introduces some randomness (20 octets in the case of SHA-1), so that different applications of the encoding operation to the same message will produce different encoded messages. Furthermore, a deterministic padding operation including the hash of the encoding parameters is applied to the message to be encoded. In this manner, the encoded message will have some structure that can be verified by the decoding operation. For example, the decoding operation is able to extract the hash of the encoding parameters from the encoded message.

The EME-OAEP encoding operation is described in Section 1.3.1, while the decoding operation is treated in Section 1.3.2. EME-OAEP is defined as in PKCS #1 v2.0 [37] and coincides with EME1 in IEEE Std 1363-2000 [25] and EME-OAEP in the draft ANSI X9.44 [1], except that EME1 outputs an integer of a desired bit length rather than an octet string.

EME-OAEP is parameterized by the choice of hash function *Hash* and mask generation function *MGF*. In this document, *MGF* is based on *Hash* as defined in Section 1.3.3; this definition coincides with that of PKCS #1 v2.0, IEEE Std 1363-2000 and the draft ANSI X9.44. In PKCS #1 v2.0 and the draft ANSI X9.44, the recommended hash function is SHA-1 [18], while IEEE Std 1363-2000 recommends SHA-1 and RIPEMD-160 [17]. Future revisions of PKCS #1 may suggest other hash functions and define new mask generation functions.

1.3.1 OAEP encoding operation

EME-OAEP-ENCODE$(M, P, emLen)$

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function
Input:	M	message to be encoded, an octet string of length at most $emLen - 1 - 2hLen$ (*mLen* denotes the length in octets of the message)
	P	encoding parameters, an octet string

	emLen	intended length in octets of the encoded message, at least $2hLen + 1$
Output:	EM	encoded message, an octet string of length *emLen*
Errors:	"message too long"; "parameter string too long"	

1. If the length of P is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1) then output "parameter string too long" and stop.
2. If $mLen > emLen - 2hLen - 1$, output "message too long" and stop.
3. Generate an octet string PS consisting of $emLen - mLen - 2hLen - 1$ zero octets. The length of PS may be 0.
4. Let $pHash = Hash(P)$, an octet string of length *hLen*.
5. Concatenate $pHash, PS$, the message M, and other padding to form a data block DB as $DB = pHash \| PS \| 01 \| M$.
6. Generate a random octet string *seed* of length *hLen*.
7. Let $dbMask = MGF(seed, emLen - hLen)$.
8. Let $maskedDB = DB \oplus dbMask$.
9. Let $seedMask = MGF(maskedDB, hLen)$.
10. Let $maskedSeed = seed \oplus seedMask$.
11. Let $EM = maskedSeed \| maskedDB$.
12. Output EM.

Remark. The EME-OAEP encoding operation is illustrated in Figure C.1 at the end of this document.

1.3.2 OAEP decoding operation

EME-OAEP-DECODE(EM, P)

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
	MGF	mask generation function
Input:	EM	encoded message, an octet string of length at least $2hLen + 1$ (*emLen* denotes the length in octets of EM)

	P	encoding parameters, an octet string
Output:	m	recovered message, an octet string of length at most $emLen - 1 - 2hLen$
Errors:	"decoding error"	

1. If the length of P is greater than the input limitation for the hash function ($2^{61} - 1$ octets for SHA-1) then output "decoding error" and stop.

2. If $emLen < 2hLen + 1$, output "decoding error" and stop.

3. Let *maskedSeed* be the first *hLen* octets of EM and let *maskedDB* be the remaining $emLen - hLen$ octets.

4. Let $seedMask = MGF(maskedDB, hLen)$.

5. Let $seed = maskedSeed \oplus seedMask$.

6. Let $dbMask = MGF(seed, emLen - hLen)$.

7. Let $DB = maskedDB \oplus dbMask$.

8. Let $pHash = Hash(P)$, an octet string of length *hLen*.

9. Separate DB into an octet string *pHash'* consisting of the first *hLen* octets of DB, a (possibly empty) octet string PS consisting of consecutive zero octets following *pHash'*, and a message M as
$$DB = pHash' \| PS \| 01 \| M.$$
If there is no 01 octet to separate PS from M, output "decoding error" and stop.

10. If *pHash'* does not equal *pHash*, output "decoding error" and stop.

11. Output M.

Remark. The EME-OAEP decoding operation is illustrated in Figure C.2 at the end of this document.

1.3.3 Mask generation function

A mask generation function takes an octet string of variable length and a desired output length as input, and outputs an octet string of the desired length. There may be restrictions on the length of the input and output octet strings, but such bounds are generally very large. Mask generation functions are deterministic; the octet string output is completely determined by the input octet string. The output of a mask generation function should be pseudorandom, that is, it should be infeasible to predict, given one part of the output but not the input, another part of the output. The provable security of RSAES-OAEP relies on the random nature of the output of the mask generation function, which in turn relies on the random nature of the underlying hash function.

$MGF(Z, l)$

Options:	*Hash*	hash function (*hLen* denotes the length in octets of the hash function output)
Input:	Z	seed from which mask is generated, an octet string
	l	intended length in octets of the mask, at most $2^{32}hLen$
Output:	*mask*	mask, an octet string of length l
Errors:	"mask too long"	

Steps:

1. If $l > 2^{32}hLen$, output "mask too long" and stop.

2. Let T be the empty octet string.

3. For $i = 0$ to $\lceil l/hLen \rceil - 1$, do

    3.1 Convert i to an octet string C of length 4 with the primitive I2OSP:

    $$C = \text{I2OSP}(i, 4).$$

    3.2 Concatenate the hash of the seed Z and C to the octet string T:

    $$T = T \| Hash(Z \| C)$$

4. Output the leading l octets of T as the octet string *mask*.

1.4 The RSAES-OAEP encryption scheme

1.4.1 Encryption operation

RSAES-OAEP-ENCRYPT$((n, e), M, P)$

Input:	(n, e)	recipient's RSA public key
	M	message to be encrypted, an octet string of length at most $k - 2 - 2hLen$, where k is the length in octets of the modulus n and $hLen$ is the length in octets of the hash function output for EME-OAEP.
	P	encoding parameters, an octet string that may be empty
Output:	C	ciphertext, an octet string of length k
Errors:	"message too long"	

Assumption: public key (n, e) is valid

1. Apply the EME-OAEP encoding operation (see Section 1.3.1) to the message M and the encoding parameters P to produce an encoded message EM of length $k - 1$ octets:

$$EM = \text{EME-OAEP-ENCODE}(M, P, k - 1).$$

If the encoding operation outputs "message too long," then output "message too long" and stop.

2. Convert the encoded message EM to an integer message representative m (see Section 1.1.2.2):

$$m = \text{OS2IP}(EM).$$

3. Apply the RSAEP encryption primitive (Section 1.2.3) to the public key (n, e) and the message representative m to produce an integer ciphertext representative c:

$$c = \text{RSAEP}((n, e), m).$$

4. Convert the ciphertext representative c to a ciphertext C of length k octets (see Section 1.1.2.1):

$$C = \text{I2OSP}(c, k).$$

5. Output the ciphertext C.

1.4.2 Decryption operation

RSAES-OAEP-DECRYPT(K, C, P)

Input: K recipient's RSA private key

	C	ciphertext to be decrypted, an octet string of length k, where k is the length in octets of the modulus n
	P	encoding parameters, an octet string that may be empty
Output:	M	message, an octet string of length at most $k - 2 - 2hLen$, where $hLen$ is the length in octets of the hash function output for EME-OAEP
Errors:	"decryption error"	

Assumption: private key K is valid

1. If the length of the ciphertext C is not k octets, output "decryption error" and stop.

2. Convert the ciphertext C to an integer ciphertext representative c (see Section 1.1.2.2):

$$c = \mathrm{OS2IP}(C).$$

3. Apply the RSADP decryption primitive (Section 1.2.4) to the private key K and the ciphertext representative c to produce an integer message representative m:

$$m = \mathrm{RSADP}(K, c).$$

   If RSADP outputs "ciphertext representative out of range," then output "decryption error" and stop.

4. Convert the message representative m to an encoded message EM of length $k - 1$ octets (see Section 1.1.2.1):

$$EM = \mathrm{I2OSP}(m, k-1).$$

   If I2OSP outputs "integer too large," then output "decryption error" and stop.

5. Apply the EME-OAEP decoding operation (see Section 1.3.2) to the encoded message EM and the encoding parameters P to recover a message M:

$$M = \text{EME-OAEP-DECODE}(EM, P).$$

   If the decoding operation outputs "decoding error," then output "decryption error" and stop.

6. Output the message M.

Remark. It is important that the errors in steps 4 and 5 are indistinguishable, otherwise an adversary may be able to extract useful information from the type of error occurred. In particular, the error messages in steps 4 and 5 must be identical. Moreover, the execution time of the decryption operation must not reveal whether an error has occurred. One way of achieving this is as follows: In case of error in step 4, proceed to step 5 with EM set to a string of zero octets. Error message information is used to mount a chosen ciphertext attack on PKCS #1 v1.5 encrypted messages in [5].

1.5 An algorithm for generating an RSA key pair

In this section, we give an algorithm for generating an RSA key pair. In Section 1.5.1, we describe the key generation algorithm RSAKG, which produces a key pair with a modulus of a desired bit length and a predetermined public exponent. The algorithm RSAKG will need two primes. Each of these primes is produced from a subroutine PG, which takes an interval and the public exponent as input and returns a random prime p belonging to the interval, such that $p-1$ is relatively prime to the public exponent. The prime generation algorithm PG is given in Section 1.5.2.

1.5.1 Key generation

RSAKG(L, e)

Input:	L	the desired bit length for the modulus
	e	the public exponent, an odd integer greater than 1
Output:	K	a valid private key in either of the two representations described in Section 1.2.2 with a modulus of bit length L
	(n, e)	a valid public key

1. Generate a prime p such that $\lfloor 2^{(L-1)/2} \rfloor + 1 \leq p \leq \lceil 2^{L/2} \rceil - 1$ and such that $\text{GCD}(p, e) = 1$ using PG with input $(\lfloor 2^{(L-1)/2} \rfloor + 1, \lceil 2^{L/2} \rceil - 1, e)$.

2. Generate a prime q such that $\lfloor 2^{(L-1)/2} \rfloor + 1 \leq q \leq \lceil 2^{L/2} \rceil - 1$ and such that $\text{GCD}(q, e) = 1$ using PG with input $(\lfloor 2^{(L-1)/2} \rfloor + 1, \lceil 2^{L/2} \rceil - 1, e)$.

3. Put $n = pq$. The public key is (n, e).

4. Compute the parameters necessary for the private key K (see Section 1.2.2).

5. Output the public key and the private key.

Remark. There are other possibilities for the choice of intervals. For example, in Annex A to IEEE Std 1363-2000 [25], the interval for p is $[2^{M-1}, 2^M - 1]$, where $M = \lceil L/2 \rceil$, and the interval for q is chosen in a way that makes the product of p and q an L-bit number. Contrary to the choice of intervals in the algorithm RSAKG above, this approach may give primes p and q with different bit lengths.

1.5.2 Generating a random prime within a given interval

The prime generation algorithm suggested in this section uses the probabilistic Miller-Rabin primality test and does not check whether the primes are "strong" or not -- with an overwhelming probability, the generated numbers will indeed be strong primes. See Annex A in IEEE Std 1363-2000 [25] for an algorithm giving a proof that a number is a prime and for an algorithm generating primes that are provably strong.

PG(r, s, e)

Input: r the lower bound for the prime to be generated

s the upper bound for the prime to be generated

e an odd positive integer

Output: p an odd random prime uniformly chosen from the interval $[r, s]$ such that GCD$(p-1, e) = 1$

1 Generate a random odd integer p uniformly from the interval $[r-2, s-2]$.

2 Set $p \leftarrow p + 2$.

3 If p is divisible by any of the, say, 2000 smallest primes, return to step 2.

4 If GCD$(p-1, e) \neq 1$, then return to step 2.

5 Let v, w be such that w is odd and $p - 1 = w2^v$.

6 Choose a positive integer t such that the primality test in step 8 is successful with a sufficiently large probability (see Remark 1 below).

7 Set $i \leftarrow 1$.

8 While $i \leq t$ do

8.1 Generate a random integer a uniformly from the interval $[1, p-1]$.

8.2 Set $b \leftarrow a^w \bmod p$.

8.3 If $b = 1$ or $b = p - 1$, then go to step 8.6.

8.4 Set $j \leftarrow 0$.

8.5 While $b \neq p - 1$ do

8.5.1 Set $j \leftarrow j + 1$.

8.5.2 If $j = v$, then return to step 2 (p is composite).

8.5.3 Set $b \leftarrow b^2 \bmod p$ ($= a^{w2^j} \bmod p$).

8.5.4 If $b = 1$, then return to step 2 (p is composite).

8.6 Set $i \leftarrow i + 1$.

9 If p is greater than s, return to step 1.

10 Output p.

Remarks.

1. The Miller-Rabin primality test in step 8 is a probabilistic test, which means that there is a small chance that an error occurs, i.e., a random composite number passes the test. For a randomly chosen candidate this error has been estimated in [16]; to achieve an error probability less than 2^{-100} for a random k-bit integer, it suffices to choose the parameter t in accordance with the following table.

k	256	342	384	410	512	683	768	1024
t	17	12	11	10	8	6	5	4

$342 = \lceil 1024/3 \rceil$, $410 = \lceil 2048/5 \rceil$ and $683 = \lceil 2048/3 \rceil$ are included for completeness, being typical values in the multi-prime setting (see Appendix A).

2. Step 8.5 in the Miller-Rabin primality test is based on the following facts: If p is a prime, then $a^{p-1} \equiv 1 \pmod{p}$. If, in addition, t is even and satisfies $a^t \equiv 1 \pmod{p}$, then $a^{t/2} \equiv \pm 1 \pmod{p}$.

3. To check whether p is divisible by any of the smallest primes $p_1, p_2, \ldots, p_{2000}$, we have to compute the remainder r_i when p is divided by p_i; if $r_i = 0$ for some i, then p is not a prime. Yet, we need to perform this trial division only once for each prime p_i; when p is replaced with $p + 2$, just replace r_i with $(r_i + 2) \bmod p_i$.

4. Instead of increasing the value of p in steps of 2 until a prime is found, one may generate a brand new random integer each time a number p proves to be composite. In this manner, the prime eventually found will be chosen uniformly at random; an ostensible drawback with incremental search is that the prime eventually found will no longer be uniformly chosen (primes preceded by a big "gap" containing no primes will be chosen with a larger probability than other primes). Yet, Brandt and Damgard [2] have shown that this bias does not affect security. Moreover, with the alternative approach, we cannot simplify our computations in the way described in Remark 3.

5. A practical method for generating a number p nearly uniformly from an interval $[r, s]$ is as follows. Let M be the bitlength of s and let m be a suitably chosen number, for example, $m = 32$. Generate an $(M + m)$-bit random integer x uniformly from the interval $[0, 2^{M+m} - 1]$ and put

$$p = (x \bmod (s - r + 1)) + r.$$

The extra m bits of x are included to make p more uniformly distributed over the interval $[r, s]$; any of the most probable integers (which are at the beginning of the interval) will be chosen with probability at most $1 + 2^{-m}$ times the probability for any of the least probable integers (which are at the end of the interval).

Chapter 2

Security properties

The security of RSAES-OAEP depends on the security of the underlying RSA encryption and decryption primitives, RSAEP and RSADP (described in Subsections 1.2.3 and 1.2.4), and the security of the OAEP encoding method (described in Section 1.3).

The security of the encryption and decryption primitives will be described in Section 2.1. The advantage of the technique that is generically known as OAEP (*Optimal Asymmetric Encryption Padding* [4]) is that under one model of analysis -- the so-called *random oracle model* -- the security of RSAES-OAEP can be tightly related to the security of RSAEP/RSADP. This allows us to consider the security of RSAES-OAEP in Section 2.2.

2.1 RSA encryption and decryption primitive

Over the years many different researchers have considered the security of RSAEP/RSADP. Boneh [6] gives an excellent survey of the main attacks which we summarize here. In some cases, the discussion of the private exponent d refers to the inverse of e mod $(p-1)(q-1)$ as opposed to the alternative definition given in this document; knowledge of either is of course sufficient to compromise security.

1. Taking e^{th} roots of c modulo n when the factorization of n is unknown.

   This is an open problem [10] and there are currently no practical techniques for achieving this when typical parameter choices are made. Although the RSA problem of taking e^{th} roots modulo n is not known to be equivalent to factoring the modulus, factorization is the only method known for solving the problem in the general case. Boneh and Venkatesan [10] have shown that if there is an algebraic reduction from factoring to e^{th} roots in time T, then it is possible to factor in (roughly) time $2^e T$. This means that, for very small e (say, less than 64), if factoring is hard, then the problems are not equivalent (at least via algebraic reductions). For larger e (for instance, $e = 2^{16} + 1$), there still might be an efficient reduction. However, see further notes below for possible methods of determining the private key d, and hence solving the problem as well as factoring the modulus, when sufficient information about the private key is leaked.

2. Factoring n and then taking e^{th} roots of c modulo n.

   Trends in the effectiveness of factoring integers are carefully collated and scrutinized by the cryptographic community. Progress over past years has been gradual but steady. Under a variety of models it is possible to provide a range of predictions for the continued resistance of an RSA modulus n to a factoring attack [33, 43].

   The most recent factorization of an RSA modulus was RSA-512, a 512-bit RSA modulus [40, 45]. It is possible to use this emperical evidence as a base point from which to make estimates for the likely security of RSA moduli of different sizes. While there are a variety of comparisons available [33, 43] which sometimes offer divergent views, there seems to be a general consensus that the security offered by 1024-bit RSAEP/RSADP is roughly equivalent to that offered by 80-bit symmetric key cryptography in terms of computational effort[1]. Note that the user can freely choose appropriate parameter choices to give a level of protection appropriate to the user's own risk assessment and key lengths of 2048-bit and higher offer an increasingly significant margin for security. Recent proposals to use an opto-electronic device TWINKLE to speed up part of the factoring process [32] are unlikely to have any significant impact at the recommended parameter choices today [42].

3. Two users sharing a common modulus.

   Two users should never share the same modulus n, even if they use different encryption/decryption exponent pairs. Systems that allow users to share moduli are using RSAEP/RSADP inappropriately.

4. Using a small private exponent d.

   It may be tempting to use a small private exponent d for reasons of efficiency. It has been shown [8, 47] that a basic implementation of RSAEP/RSADP can be susceptible to attack if $d < n^{0.292}$. It is conjectured that this might continue to be the case if $d < n^{0.5}$. A small private exponent d should not be used.

5. Using a low public exponent e.

   Some progress has been made [13] on exploiting the use of a low public exponent. While there is no particular attack within the context of RSAES-OAEP that compromises the security of the public exponent $e = 3$, more conservative users may prefer to use other public exponents such as $e = 17$ or $e = 2^{16} + 1$ while still retaining a very competitive performance for encryption. Also, as noted further in Annex D.4.3.4 of IEEE Std 1363-2000 [25], a larger public exponent can provide an additional level of defense in the case that the underlying random number generation fails in an implementation of the OAEP method, undermining the security properties offered by that method.

6. Broadcasting the same message to multiple users.

   It has been known for some time that it can be unsafe to broadcast the same message to different users if no padding, or a very simple padding scheme is used [22]. Application of [13] allows improvements to this original work [22] to be made. The application of EME-OAEP as the padding scheme prior to encryption is sufficient to resist these attacks.

7. Sending related messages to the same user.

   For small e it can be possible to recover simply-related messages that are encrypted under the same public-key [14]. Extensions showed some practical applications of this work when small amounts of random padding are used prior to encrypting with RSAEP. In [15] an attack is described that applies to a case where the plaintext ends by sufficiently many zeroes, and two or more ciphertexts corresponding to the same plaintext are available. The application of EME-OAEP as the padding scheme prior to encryption is sufficient to resist these attacks.

[1] Under an equivalent-cost analysis 1024-bit RSAEP/RSADP is viewed as offering greater security than 80-bit symmetric key cryptography [43].

8. Using partial information about the private key d.

   Given the $\lceil \log_2 n/4 \rceil$ least significant bits of the private exponent d, it is possible to reconstruct all of d if $e < \sqrt{n}$ [9]. Furthermore, when a small exponent e is used, the most significant half of the bits of d can be leaked [6]. Although determining the remaining bits is of course still difficult, if the private exponent is protected by symmetric encryption, knowledge of the most significant half of the bits of d may facilitate a known-plaintext attack on the symmetric encryption method. Accordingly, it is essential that the remaining bits of the private key d should be well protected.

9. Using partial information about the factors p, q.

   Given the $\lceil \log_2 n/4 \rceil$ least significant bits of p (resp. q) or the $\lceil \log_2 n/4 \rceil$ most significant bits of p (resp. q), one can efficiently factor n [13]. The entirety of the secret primes p and q should be protected.

It is generally accepted that when RSAEP is used with appropriate parameter choices and coupled with a secure padding scheme like OAEP, then the most effective attack is to factor the modulus n. Under this assumption we can relate the security of RSAES-OAEP to the effort required to factor the underlying modulus of different sizes.

A crude estimate for the increased computing resources required beyond that for factoring RSA-512 can be derived [43] for different sizes of RSA moduli. For 1024-bit RSA moduli, the factor increase in computational power is estimated as 7×10^6 while for 2048-bit RSA the estimate is 9×10^{15}. Increases in computing power might be accounted for by some combination of the use of more machines, increasingly powerful machines, or more calendar time. The calendar time required for the factorization of RSA-512 was 3.7 months [40]. Other issues like the cost and availability of memory may also figure in deriving predictions for the future security of RSAEP/RSADP [33, 43].

2.2 RSAES-OAEP

Over the years a variety of padding methods for both RSA encryption and signature primitives have been proposed. OAEP has the benefit of coupling good security guarantees with good performance. In particular OAEP combined with RSA offers security against what is termed *adaptive chosen ciphertext attacks*; see the discussion in Section 2.2.1 below. We might contrast this with the original simple padding schemes used in previous versions[2] of PKCS #1 (v1.5 and earlier) which did not offer security against such attacks.

The OAEP construction requires the use of a *mask generation function*. It is typical to practically instantiate this function -- as we do here -- with a trusted hash function such as SHA-1.

2.2.1 The OAEP encoding method

Recently discovered shortcomings in the OAEP encoding method seemed to suggest that there are reasons to call the security of RSAES-OAEP in question. Luckily, this is not the case; RSAES-OAEP

[2] For example, the encryption scheme defined in PKCS #1 v1.5 was found to be susceptible to an attack by Bleichenbacher [5].

is provably secure in the random oracle model. In this section we give a brief outline of the background.

In 1994, Bellare and Rogaway[4] introduced a security concept that they denoted *plaintext awareness* (PA94). They proved that if an encryption primitive (e.g., RSAEP) is hard to invert without the private key, then the corresponding OAEP-based encryption scheme is plaintext-aware , meaning roughly that an adversary cannot produce a valid ciphertext without actually "knowing" the underlying plaintext. Plaintext awareness of an encryption scheme is closely related to the resistance of the scheme against *chosen ciphertext attacks*. In such attacks, an adversary is given the opportunity to send queries to an oracle simulating the decryption primitive. Using the results of these queries, the adversary attempts to decrypt a challenge ciphertext.

However, there are *two* flavors of chosen ciphertext attacks, and PA94 implies security against only one of them. The difference relies on what the adversary is allowed to do after she is given the challenge ciphertext. The *indifferent* attack scenario (denoted CCA1) does not admit any queries to the decryption oracle after the adversary is given the challenge ciphertext, whereas the *adaptive* scenario (denoted CCA2) does (except that the decryption oracle refuses to decrypt the challenge ciphertext once it is published). In 1998, Bellare and Rogaway, together with Desai and Pointcheval [3], came up with a new, stronger notion of plaintext awareness (PA98) that does imply security against CCA2.

To summarize, there have been two potential sources for misconception: that PA94 and PA98 are equivalent concepts; or that CCA1 and CCA2 are equivalent concepts. Either assumption leads to the conclusion that the Bellare-Rogaway paper implies security of OAEP against CCA2 , which it does not. OAEP has never been proven secure against CCA2; in fact, Victor Shoup [41] ingeniously demonstrated recently that such a proof does not exist in the general case. Put briefly, Shoup showed that an adversary in the CCA2 scenario who knows how to *partially* invert the encryption primitive but does not know how to invert it *completely* may well be able to break the scheme. For example, one may imagine an attacker who is able to break RSAES-OAEP if she is able to recover all but the first 20 bytes of an integer encrypted with RSAEP. Such an attacker does not need to be able to fully invert RSAEP, because she does not use the first 20 octets in her attack.

Still, RSAES-OAEP *is* secure against CCA2, which was demonstrated by Fujisaki, Okamoto, Pointcheval, and Stern [19] shortly after the announcement of Shoup's result. Using clever lattice reduction techniques, they managed to show how to invert RSAEP completely given a sufficiently large part of the pre-image. This observation, combined with a proof that OAEP is secure against CCA2 if the underlying encryption primitive is hard to *partially* invert, fills the gap between what Bellare and Rogaway proved about RSAES-OAEP and what some may have believed that they proved. Somewhat paradoxically, we are hence saved by an ostensible weakness in RSAEP (i.e., the whole inverse can be deduced from parts of it). As a consequence, it makes little sense replacing OAEP with a "more secure" encoding method, because if a CCA2 adversary is able to break RSAES-OAEP, then she will be able to break RSAEP equipped with any encoding method (if maybe slightly less efficiently). For encryption primitives different from RSAEP, however, it might be worthwhile considering a stronger encoding method such as OAEP+ suggested by Shoup [41].

2.3 Basic techniques to avoid implementation weaknesses

The analytical security of RSAEP/RSADP along with RSAES-OAEP have been considered in previous section. However we still need to implement RSAES-OAEP securely.

It is possible that weaknesses could be introduced when writing RSAES-OAEP as a component of an application, or when running an application from which so-called side-channel information can be deduced.

Within an engineering environment, implementation errors can be virtually eliminated by adopting good product design and engineering practices. Adequate testing and product management throughout the development cycle are essential. With regards to running an application using RSAES-OAEP, protection of all private key information, secure memory management, and secure error handling are all needed. Issues like the source of random numbers are, of course, fundamental to the security of the implementation.

Over recent years there have been several proposals to break cryptosystems by utilizing so-called side-channel information. Examples include *timing attacks* [30], *power analysis* [31], and *fault analysis* [7].

Implementations of RSAES-OAEP can be made resistant to timing attacks and power analysis by ensuring that all the steps in the computation of a private key operation take the same amount of time or consume the same amount of power (though the environment of use will determine whether or not such attacks and such precautions are viable). A more elegant approach to providing resistance to timing attacks is to use *blinding* [28] as suggested by Ronald L. Rivest. Fault analysis is potentially applicable to an implementation of RSAEP/RSADP which uses the chinese remainder theorem [38] during decryption. However the use of OAEP padding means that this attack does not pose a threat to RSAES-OAEP.

As with most cryptographic primitives, vulnerability against side-channel attacks can sometimes be difficult to protect against at the algorithm level. Instead, it seems far more appropriate to provide resistance to such system attacks at the level they are mounted -- at the system level.

No method is specified in this document for public-key validation, that is providing assurance to a party (that is, the party generating the key pair, the party using the public key, or a neutral third party) that a candidate public key conforms to the arithmetic definition of a public key. An invalid public key might exist because of an inadvertent key generation calculation error or due to deliberate action of an adversary. Use of an invalid public key should be assumed to void all security assurances, including the inability of an adversary to decrypt a message or discover the associated private key.

As a general principle for any cryptosytem, the use of an improperly generated but otherwise valid public key (for instance, one produced from an insufficiently random source), or an improperly protected private key, may also void all security assurances. Implementation validation can mitigate these risks as well as the possibility that invalid keys are used. However, it does not provide specific assurance that a given public key is in fact valid.

Methods for determining whether an RSA public key is valid are an area of active research; see [11, 12, 21, 34, 35, 44] for some recent developments. By contrast, methods for validating public keys in discrete logarithm and elliptic curve cryptosystems are straightforward: a public key can be validated directly and without interaction with the private key holder. Annex A of IEEE Std 1363-2000 [25] gives examples of these methods.

Chapter 3

Computational efficiency

Any single set of figures for the performance of RSAES-OAEP for a given platform can be misleading. There is much variability in the performance of RSAES-OAEP depending on the processor and clocking rate, the amount of space available, the type of arithmetic procedures used, the use of dedicated hardware for some or part of the computation, and so forth.

Here we give a few key figures that give some illustration of the range of performances possible. The overhead due to OAEP is insignificant compared to the computational requirements of the private and public-key computations.

3.1 Dedicated coprocessors

source	*key-size*	*encryption*	*decryption*
		$e = 3$	*with CRT*
NetSwift-1000 [26]	1024	1 ms	5 ms
NetSwift-1000 [26]	2048	2 ms	25 ms
		$e = 17$	*with CRT*
IBM 4758 (002/003) crypto co-processor [24]	1024	2 ms	6 ms

3.2 Software

source	*key-size*	*encryption*	*decryption*
		$e = 3$	*with CRT*
C++ Celeron 450 MHz [46]	1024	1 ms	27 ms
C++ Celeron 450 MHz [46]	2048	2 ms	183 ms

3.3 Smart card coprocessors

source	*clock*	*key-size*	*encryption*	*decryption*
			$e = 2^{16} + 1$	*with CRT*
SGS-Thomson ST19KF16 [23]	10 MHz	1024	5 ms	110 ms
	10 MHz	2048	100 ms	780 ms
Philips P83W8532 [23]	5 MHz	1024	25 ms	160 ms
	5 MHz	2048	54 ms	1100 ms
Siemens SLE66CX160S [23]	5 MHz	1024	24 ms	230 ms
	5 MHz	2048	268 ms	1475 ms
NEC μPD789828 [23]	40 MHz	1024	7 ms	100 ms
	40 MHz	2048	45 ms	750 ms

3.4 Key generation estimates

Each user of RSAES-OAEP must use a different RSA modulus. For this, two primes must be generated. To estimate the performance of the key generation procedure, the following rule of thumb may be used.

Prime generation of p and q as defined in RSAKG (see Section 1.5.1) involves an initial sieving phase to remove obvious non-primes. Approximately 90-95 % of all integers are rejected in this phase (92 % on the average if the 170 smallest primes are used and 94 % if the 2000 smallest primes are used). Unless the number of small primes is extremely large, the sieving phase is particularly efficient and has little impact on the time for prime generation.

The Miller-Rabin primality test is applied a specified number t of times to each candidate prime that passes the sieving stage. Each of these tests costs about the same as a full exponentiation modulo p. The probability of finding an M-bit prime can be estimated by $(M \ln 2)^{-1}$, which means that finding an M-bit prime (for a $2M$-bit RSA modulus) might require that we test roughly M integers. With an expected fraction of α of the integers eliminated in the sieving phase[1], we may expect, on the average, to find a prime after about $(1 - \alpha)M \ln 2 + (t - 1)$ Miller-Rabin tests; a random composite integer will most likely fail after just one test. To generate both primes, this amounts to around $2(1 - \alpha)M \ln 2 + 2(t - 1)$ times the expected time to decrypt using a $2M$-bit modulus with CRT (see estimates for decryption earlier in this section).

With $t = 8$ and $\alpha = 0.92$, the time needed to generate a 1024-bit key is roughly 70 times the expected time to decrypt using a 1024-bit modulus with CRT, while the time needed to generate a 2048-bit key is roughly 130 times the expected time to decrypt using a 2048-bit modulus with CRT. Yet, remember that these are just very rough estimates.

[1] $\alpha = 1 - \prod_{i=1}^{k} \frac{p_i - 1}{p_i}$, where $p_1, \ldots, p_k$ are the distinct small primes used in the sieving phase.

Bibliography

[1] *ANSI X9.44: Key Establishment Using Factoring-Based Public Key Cryptography for the Financial Services Industry*. Working draft, ANSI X9F1 Working Group, June 2000.

[2] J. Brandt and I. Damgard. On generation of probable primes by incremental search. In, E.F. Brickell, editor, *Advances in Cryptology -- Crypto'92*, pages 358--370. Springer-Verlag, 1993.

[3] 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. Springer Verlag, 1998.

[4] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In, R. Rueppel editor, *Advances in Cryptology - Eurocrypt'94*, Lecture Notes in Computer Science, volume 950, pages 92--111. Springer Verlag, 1994.

[5] D. Bleichenbacher, B. Kaliski and J. Staddon. Recent results on PKCS #1: RSA Encryption Standard. *RSA Laboratories' Bulletin No. 7*, 1998.

[6] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, volume 46 no 2, 1999.

[7] D. Boneh, R. DeMillo, and R. Lipton. On the importance of checking cryptographic protocols. In, W. Fumy, editor, *Advances in Cryptology - Eurocrypt'97*, Lecture Notes in Computer Science, volume 1233, pages 37--51. Springer Verlag, 1997.

[8] D. Boneh and G. Durfee. New results on cryptanalysis of low private exponent RSA. Preprint, 1998.

[9] D. Boneh, G. Durfee, and Y. Frankel. An attack on RSA given a fraction of the private key bits. In, K. Ohta and D. Pei, editors, *Advances in Cryptology - Asiacrypt'98*, Lecture Notes in Computer Science, volume 1514, pages 25--34. Springer Verlag, 1998.

[10] D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In, K. Nyberg, editor, *Advances in Cryptology - Eurocrypt'98*, Lecture Notes in Computer Science, volume 1403, pages 59--71. Springer Verlag, 1998.

[11] F. Boudot. Efficient proofs that a committed number lies in an interval. In, B. Preneel, editor, *Advances in Cryptology - Eurocrypt 2000*, Lecture Notes in Computer Science, volume 1807, pages 431--444, Springer Verlag, 2000.

[12] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. In, J. Stern, editor, *Advances in Cryptology - Eurocrypt'99*, Lecture Notes in Computer Science, volume 1592, pages 107--122, Springer-Verlag, 1999.

[13] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10:233-260, 1997.

[14] D. Coppersmith, M. Franklin, J. Patarin, and M. Reiter. Low exponent RSA with related messages. In, U. Maurer, editor, *Advances in Cryptology - Eurocrypt'96*, Lecture Notes in Computer Science, volume 1070, pages 1--9. Springer Verlag, 1996.

[15] J.-S. Coron, M. Joye, D. Naccache and P. Paillier. New Attacks on PKCS #1 v1.5 Encryption. In, B. Preneel, editor, *Advances in Cryptology - Eurocrypt'00*, Lecture Notes in Computer Science, volume 1807. Springer Verlag, 2000.

[16] I. Damgard, P. Landrock, and C. Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of Computation*, 61: 177--194, (1993).

[17] H. Dobbertin, A. Bosselaers and B. Preneel. RIPEMD-160: a strengthened version of RIPEMD. In D. Gollmann, editor, *Fast Software Encryption, Third International Workshop, Lecture Notes in Computer Science 1039*, pages 71--72. Springer-Verlag, 1996. A corrected and updated version is available from www.esat.kuleuven.ac.be/~bosselae/ripemd160.html .

[18] *FIPS Publication 180-1: Secure Hash Standard*, National Institute of Standards and Technology (NIST), 1994.

[19] E. Fujisaki, T. Okamoto, D. Pointcheval and J. Stern. RSA-OAEP Is Still Alive! Preprint, November 2000. Available from http://eprint.iacr.org/ .

[20] H. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8 (6): 140--147, 1959.

[21] R. Gennaro, D. Micciancio, and T. Rabin. An efficient non-interactive statistical zero-knowledge proof system for quasi-safe prime products. In *Proceedings from the 5th ACM Conference on Computer and Communications Security*, pages 67--72, 1998.

[22] J. Hastad. Solving simultaneous modular equations of low degree. SIAM J. of Computing, 17:336-341, 1988.

[23] H. Handschuh and P. Paillier. Smart card crypto-coprocessors for public-key cryptography. *CryptoBytes*, Vol.4, No.1, pages 6--10. 1998.

[24] IBM Product Description. IBM 4758 PCI Cryptographic Coprocessor. Available at www-3.ibm.com/security/cryptocards/ .

[25] *IEEE Std 1363-2000: Standard Specifications for Public Key Cryptography*. IEEE, August 2000. The last draft is available via http://grouper.ieee.org/groups/1363/ .

[26] ISGLABS Product Description. NetSwift-1000 Timing Performance. Available at isglabs.rainbow.com/isglabs/cs_timing/cs_timing.htm .

[27] D. Johnson and S. Matyas. Asymmetric encryption: Evolution and enhancements. *CryptoBytes* 2 (1), 1996. Available via www.rsalabs.com .

[28] B. Kaliski. Timing Attacks on Cryptosystems. *RSA Laboratories Bulletin No. 2*, January 1996. Available via www.rsalabs.com .

[29] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer-Verlag, 1994.

[30] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In, N. Koblitz, editor, *Advances in Cryptology - Crypto'96*, Lecture Notes in Computer Science, volume 1109, pages 104--113. Springer Verlag, 1996.

[31] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Weiner, editor, *Advances in Cryptology - Crypto'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388--397, 1999. Springer Verlag.

[32] A. Lenstra and A. Shamir. Analysis and optimization of the TWINKLE factoring device. October 1999.

[33] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. November 24, 1999. Available at www.cryptosavvy.com/cryptosizes.pdf .

[34] M. Liskov and R. Silverman. A statistical zero-knowledge proof for secure RSA keys. Submitted to J. Cryptology.

[35] W. Mao. Verifiable partial sharing of the factors of an integer. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography '98 Proceedings*, volume 1556 of Lecture Notes in Computer Science, pages 95--109, Springer, New York, 1999.

[36] A. Menezes, P. van Oorschot and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[37] *PKCS #1: RSA Cryptography Standard, Version 2.0*. RSA Laboratories, 1998. Available via www.rsalabs.com .

[38] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18 (21): 905--907, 1982.

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

[40] Factorization of RSA-155. RSA Laboratories. Available via www.rsalabs.com .

[41] V. Shoup. OAEP Reconsidered. Preprint, November 2000. Available from http://eprint.iacr.org/ .

[42] R. Silverman. An analysis of Shamir's factoring device. RSA Laboratories, Bulletin #12, May 3, 1999. Available via www.rsalabs.com .

[43] R. Silverman. A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths. *RSA Laboratories Bulletin No. 13*, April 2000. Available via www.rsalabs.com .

[44] Robert D. Silverman. RSA public key validation. To appear.

[45] H. te Riele. Security of E-commerce threatened by 512-bit number factorization. Press release available at `www.cwi.nl/~kik/persb-UK.html` .

[46] W. Dai. Crypto++ 3.1 Benchmarks. Available via `www.eskimo.com/~weidai/benchmarks.html` .

[47] M. Wiener. Cryptanalysis of short RSA secret exponents. *IEEE Transactions on Information Theory*, 36:553-558, 1990.

A
Multi-prime RSA

In this section, multi-prime RSA is treated; "multi-prime" means that the modulus is the product of three or more primes. The benefit of multi-prime RSA is lower computational cost for the decryption and signature primitives, provided that the CRT (Chinese Remainder Theorem) is used. Better performance can be achieved on single processor platforms, but to a greater extent on multiprocessor platforms, where the modular exponentiations involved can be done in parallel. As of today, for a level of security comparable to that of ordinary two-prime RSA, it is recommended that the number of prime factors in the modulus be pursuant to the following table.

Bit length of modulus	1024	1536	2048
Maximal number of primes	3	4	5

For a thorough treatment of the security of multi-prime RSA, see [43].

To support multi-prime RSA, one has to update some of the definitions in this document. We refer to Amendment 1 of PKCS #1 v2.0 [37] for a detailed description and confine ourselves to discussing the most important modificiations of the decryption primitive RSADP (Subsection 1.2.4).

Let k be the number of prime factors of the modulus, and let $p = r_1, q = r_2, r_3, \ldots, r_k$ be the primes. Let d_i be the inverse of e modulo $r_i - 1$ for $i \geq 3$:

$$e \cdot d_i \equiv 1 \pmod{r_i - 1}.$$

Furthermore, let t_i be the multiplicative inverse of $r_1 \cdot \ldots \cdot r_{i-1}$ modulo r_i for $i \geq 3$. Following Amendment 1 to PKCS #1 v2.0, we redefine the second representation as consisting of the quintuple (p, q, dP, dQ,$qInv$) and the sequence of triplets (r_i, d_i, t_i), $i = 3, \ldots, k$. The following steps need to be added to RSADP.

2.6 Let $m_i = c^{d_i} \bmod r_i$, $i = 3, \ldots k$.

2.7 Let $R = r_1$ and for $i = 3$ to k do

2.7.1 Let $R = R \cdot r_{i-1}$.

2.7.2 Let $h = (m_i - m) \cdot t_i \bmod r_i$.

2.7.3 Let $m = m + R \cdot h$.

Remark. The algorithm used here is based on Garner's algorithm (see [20] or Algorithm 14.71 in [36]). Yet, one may follow Garner's algorithm more closely, rewriting steps 2.2-2.7 as a single loop -- just reverse the order of p and q. For consistency with existing standards, however, it is recommended that the first two primes p and q are treated separately from the additional primes.

B

Mathematical background

The purpose of this section is to give a brief summary of the mathematics needed to understand the RSA algorithm and why it works.

Recall that the public key is a pair (n, e) of integers, where n is the modulus and e is the public exponent; the modulus n is the product of two distinct primes p and q. The private key contains information about the private exponent d, which satisfies

$$(x^e)^d \equiv x \pmod{n} \tag{B.1}$$

for any integer x. The private exponent d has the property that

$$e \cdot d \equiv \begin{cases} 1 & \pmod{p-1} \\ 1 & \pmod{q-1} \end{cases}, \tag{B.2}$$

and our goal is to show that this is sufficient. For proofs of Fermat's Theorem and the Chinese Remainder Theorem, see [29] or any undergraduate textbook in number theory.

An important result is as follows.

Theorem 1 (Fermat's Theorem) *Let r be a prime and let x be an integer. Then*

$$x^{j(r-1)+1} \equiv x \pmod{r}$$

for any nonnegative integer j.

This proposition immediately implies that

$$(x^e)^d \equiv \begin{cases} x & \pmod{p} \\ x & \pmod{q} \end{cases} \tag{B.3}$$

for any x if e and d satisfy (B.2). Namely, (B.2) means that $e \cdot d = j(p-1)+1$ for some integer j and similar for q.

The next step is to demonstrate that (B.3) implies (B.1). This is a consequence of the following result.

Theorem 2 (Chinese Remainder Theorem) *Let $r_1, r_2, \ldots, r_k$ be positive integers such that $GCD(r_i, r_j) = 1$ whenever $i \neq j$. For any integers $a_1, a_2, \ldots, a_k$, the system of equations*

$$y \equiv a_i \pmod{r_i}$$

has a unique solution y such that $0 \leq y < r_1 r_2 \ldots r_k$.

RSADP (Section 1.2.4) with the second representation of the private key is based on the Chinese Remainder Theorem (CRT): With notations as in Section 1.2.4, Fermat's Theorem implies that

$$\begin{cases} m_1 & \equiv & c^d \pmod{p} \\ m_2 & \equiv & c^d \pmod{q}. \end{cases}$$

By CRT, there is exactly one solution m to the system of equations

$$\begin{cases} m & \equiv & m_1 \pmod{p} \\ m & \equiv & m_2 \pmod{q} \end{cases}$$

satisfying $0 \leq m < n$. This solution is the one given in step 2.5 in Section 1.2.4:

$$m = m_2 + q\left((m_1 - m_2)t \mod p\right),$$

where $t = qInv$ satisfies

$$q \cdot t \equiv 1 \pmod{p}.$$

Namely,

$$m_2 + q\left((m_1 - m_2)t \mod p\right) \equiv \begin{cases} m_2 + q(m_1 - m_2)t \equiv m_2 + m_1 - m_2 \equiv m_1 \pmod{p}; \\ \\ m_2 \pmod{q}. \end{cases}$$

This discussion is easily extended to the multi-prime setting (Appendix A).

C

Test vectors

In this section, we give an example of the process of encrypting and decrypting a message with RSAES-OAEP. The message is an octet string of length 16, while the size of the modulus in the public key is 1024 bits. The second representation of the private key is used, which means that CRT is applied in the decryption process.

The underlying hash function in the EME-OAEP encoding operation is SHA-1; the mask generation function is MGF with SHA-1 as specified in Section 1.3.3 in part B.1 of this document.

Integers are represented by strings of octets with the leftmost octet being the most significant octet. For example, 9202000 = `8c 69 50`.

RSA key information

n, the modulus:

```
bb f8 2f 09 06 82 ce 9c 23 38 ac 2b 9d a8 71 f7 36 8d 07 ee d4 10 43 a4
40 d6 b6 f0 74 54 f5 1f b8 df ba af 03 5c 02 ab 61 ea 48 ce eb 6f cd 48
76 ed 52 0d 60 e1 ec 46 19 71 9d 8a 5b 8b 80 7f af b8 e0 a3 df c7 37 72
3e e6 b4 b7 d9 3a 25 84 ee 6a 64 9d 06 09 53 74 88 34 b2 45 45 98 39 4e
e0 aa b1 2d 7b 61 a5 1f 52 7a 9a 41 f6 c1 68 7f e2 53 72 98 ca 2a 8f 59
46 f8 e5 fd 09 1d bd cb
```

e, the public exponent:

```
(0x)11
```

p, the first prime factor of n:

```
ee cf ae 81 b1 b9 b3 c9 08 81 0b 10 a1 b5 60 01 99 eb 9f 44 ae f4 fd a4
93 b8 1a 9e 3d 84 f6 32 12 4e f0 23 6e 5d 1e 3b 7e 28 fa e7 aa 04 0a 2d
5b 25 21 76 45 9d 1f 39 75 41 ba 2a 58 fb 65 99
```

q, the second prime factor of n:

```
c9 7f b1 f0 27 f4 53 f6 34 12 33 ea aa d1 d9 35 3f 6c 42 d0 88 66 b1 d0
5a 0f 20 35 02 8b 9d 86 98 40 b4 16 66 b4 2e 92 ea 0d a3 b4 32 04 b5 cf
ce 33 52 52 4d 04 16 a5 a4 41 e7 00 af 46 15 03
```

dP, p's exponent:

```
54 49 4c a6 3e ba 03 37 e4 e2 40 23 fc d6 9a 5a eb 07 dd dc 01 83 a4 d0
ac 9b 54 b0 51 f2 b1 3e d9 49 09 75 ea b7 74 14 ff 59 c1 f7 69 2e 9a 2e
20 2b 38 fc 91 0a 47 41 74 ad c9 3c 1f 67 c9 81
```

dQ, *q*'s exponent:

```
47 1e 02 90 ff 0a f0 75 03 51 b7 f8 78 86 4c a9 61 ad bd 3a 8a 7e 99 1c
5c 05 56 a9 4c 31 46 a7 f9 80 3f 8f 6f 8a e3 42 e9 31 fd 8a e4 7a 22 0d
1b 99 a4 95 84 98 07 fe 39 f9 24 5a 98 36 da 3d
```

qInv, the CRT coefficient:

```
b0 6c 4f da bb 63 01 19 8d 26 5b db ae 94 23 b3 80 f2 71 f7 34 53 88 50
93 07 7f cd 39 e2 11 9f c9 86 32 15 4f 58 83 b1 67 a9 67 bf 40 2b 4e 9e
2e 0f 96 56 e6 98 ea 36 66 ed fb 25 79 80 39 f7
```

Encryption

M, the message to be encrypted:

```
d4 36 e9 95 69 fd 32 a7 c8 a0 5b bc 90 d3 2c 49
```

P, encoding parameters:

```
NULL
```

$pHash = Hash(P)$:

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

$DB = pHash \| PS \| 01 \| M$:

```
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 01 d4 36 e9 95 69
fd 32 a7 c8 a0 5b bc 90 d3 2c 49
```

seed, a random octet string:

```
aa fd 12 f6 59 ca e6 34 89 b4 79 e5 07 6d de c2 f0 6c b5 8f
```

$dbMask = MGF(seed, 107)$:

```
06 e1 de b2 36 9a a5 a5 c7 07 d8 2c 8e 4e 93 24 8a c7 83 de e0 b2 c0 46
26 f5 af f9 3e dc fb 25 c9 c2 b3 ff 8a e1 0e 83 9a 2d db 4c dc fe 4f f4
77 28 b4 a1 b7 c1 36 2b aa d2 9a b4 8d 28 69 d5 02 41 21 43 58 11 59 1b
e3 92 f9 82 fb 3e 87 d0 95 ae b4 04 48 db 97 2f 3a c1 4e af f4 9c 8c 3b
7c fc 95 1a 51 ec d1 dd e6 12 64
```

$maskedDB = DB \oplus dbMask$:

```
dc d8 7d 5c 68 f1 ee a8 f5 52 67 c3 1b 2e 8b b4 25 1f 84 d7 e0 b2 c0 46
26 f5 af f9 3e dc fb 25 c9 c2 b3 ff 8a e1 0e 83 9a 2d db 4c dc fe 4f f4
77 28 b4 a1 b7 c1 36 2b aa d2 9a b4 8d 28 69 d5 02 41 21 43 58 11 59 1b
e3 92 f9 82 fb 3e 87 d0 95 ae b4 04 48 db 97 2f 3a c1 4f 7b c2 75 19 52
81 ce 32 d2 f1 b7 6d 4d 35 3e 2d
```

$seedMask = MGF(maskedDB, 20)$:

```
41 87 0b 5a b0 29 e6 57 d9 57 50 b5 4c 28 3c 08 72 5d be a9
```

maskedSeed $=$ *seed* $\oplus$ *seedMask*:

```
eb 7a 19 ac e9 e3 00 63 50 e3 29 50 4b 45 e2 ca 82 31 0b 26
```

EM $=$ *maskedSeed*$\|$*maskedDB*:

```
eb 7a 19 ac e9 e3 00 63 50 e3 29 50 4b 45 e2 ca 82 31 0b 26 dc d8 7d 5c
68 f1 ee a8 f5 52 67 c3 1b 2e 8b b4 25 1f 84 d7 e0 b2 c0 46 26 f5 af f9
3e dc fb 25 c9 c2 b3 ff 8a e1 0e 83 9a 2d db 4c dc fe 4f f4 77 28 b4 a1
b7 c1 36 2b aa d2 9a b4 8d 28 69 d5 02 41 21 43 58 11 59 1b e3 92 f9 82
fb 3e 87 d0 95 ae b4 04 48 db 97 2f 3a c1 4f 7b c2 75 19 52 81 ce 32 d2
f1 b7 6d 4d 35 3e 2d
```

C, the RSA encryption of EM:

```
12 53 e0 4d c0 a5 39 7b b4 4a 7a b8 7e 9b f2 a0 39 a3 3d 1e 99 6f c8 2a
94 cc d3 00 74 c9 5d f7 63 72 20 17 06 9e 52 68 da 5d 1c 0b 4f 87 2c f6
53 c1 1d f8 23 14 a6 79 68 df ea e2 8d ef 04 bb 6d 84 b1 c3 1d 65 4a 19
70 e5 78 3b d6 eb 96 a0 24 c2 ca 2f 4a 90 fe 9f 2e f5 c9 c1 40 e5 bb 48
da 95 36 ad 87 00 c8 4f c9 13 0a de a7 4e 55 8d 51 a7 4d df 85 d8 b5 0d
e9 68 38 d6 06 3e 09 55
```

Decryption

$c \bmod p$ (c is the integer value of C):

```
de 63 d4 72 35 66 fa a7 59 bf e4 08 82 1d d5 25 72 ec 92 85 4d df 87 a2
b6 64 d4 4d aa 37 ca 34 6a 05 20 3d 82 ff 2d e8 e3 6c ec 1d 34 f9 8e b6
05 e2 a7 d2 6d e7 af 36 9c e4 ec ae 14 e3 56 33
```

$c \bmod q$:

```
a2 d9 24 de d9 c3 6d 62 3e d9 a6 5b 5d 86 2c fb ec 8b 19 9c 64 27 9c 54
14 e6 41 19 6e f1 c9 3c 50 7a 9b 52 13 88 1a ad 05 b4 cc fa 02 8a c1 ec
61 42 09 74 bf 16 25 83 6b 0b 7d 05 fb b7 53 36
```

$m_1 = c^{dP} \bmod p = (c \bmod p)^{dP} \bmod p$:

```
89 6c a2 6c d7 e4 87 1c 7f c9 68 a8 ed ea 11 e2 71 82 4f 0e 03 65 52 17
94 f1 e9 e9 43 b4 a4 4b 57 c9 e3 95 a1 46 74 78 f5 26 49 6b 4b b9 1f 1c
ba ea 90 0f fc 60 2c f0 c6 63 6e ba 84 fc 9f f7
```

$m_2 = c^{dQ} \bmod q = (c \bmod q)^{dQ} \bmod q$:

```
4e bb 22 75 85 f0 c1 31 2d ca 19 e0 b5 41 db 14 99 fb f1 4e 27 0e 69 8e
23 9a 8c 27 a9 6c da 9a 74 09 74 de 93 7b 5c 9c 93 ea d9 46 2c 65 75 02
1a 23 d4 64 99 dc 9f 6b 35 89 75 59 60 8f 19 be
```

$h = (m_1 - m_2)qInv \bmod p$:

```
01 2b 2b 24 15 0e 76 e1 59 bd 8d db 42 76 e0 7b fa c1 88 e0 8d 60 47 cf
0e fb 8a e2 ae bd f2 51 c4 0e bc 23 dc fd 4a 34 42 43 94 ad a9 2c fc be
1b 2e ff bb 60 fd fb 03 35 9a 95 36 8d 98 09 25
```

$m = m_2 + qh$ is equal to the integer value of the encoded message EM above. The intermediate values of the decoding operation are similar to those of the encoding operation.

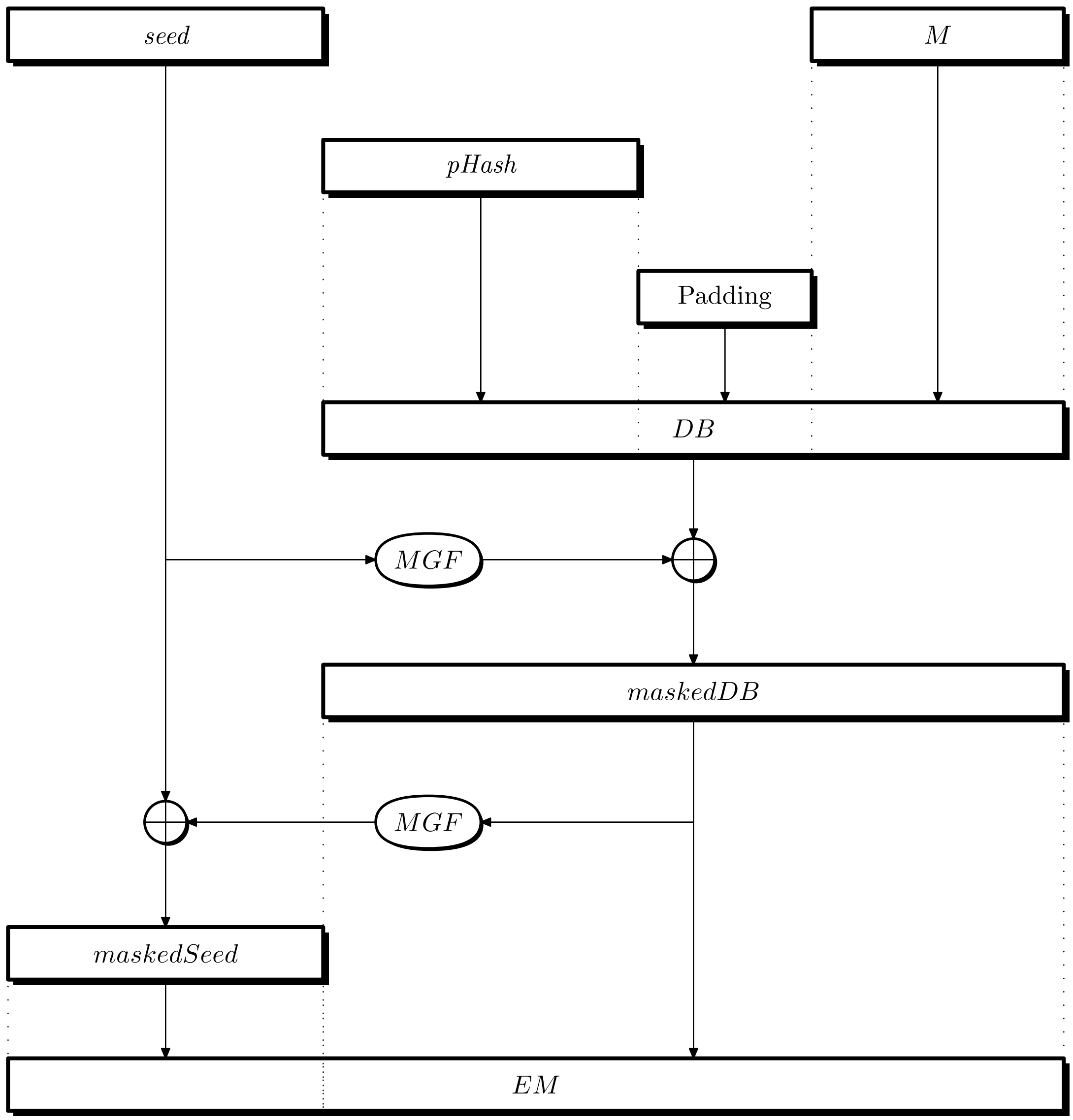

Figure C.1: *EME-OAEP encoding operation. pHash is the hash of the encoding parameters.*

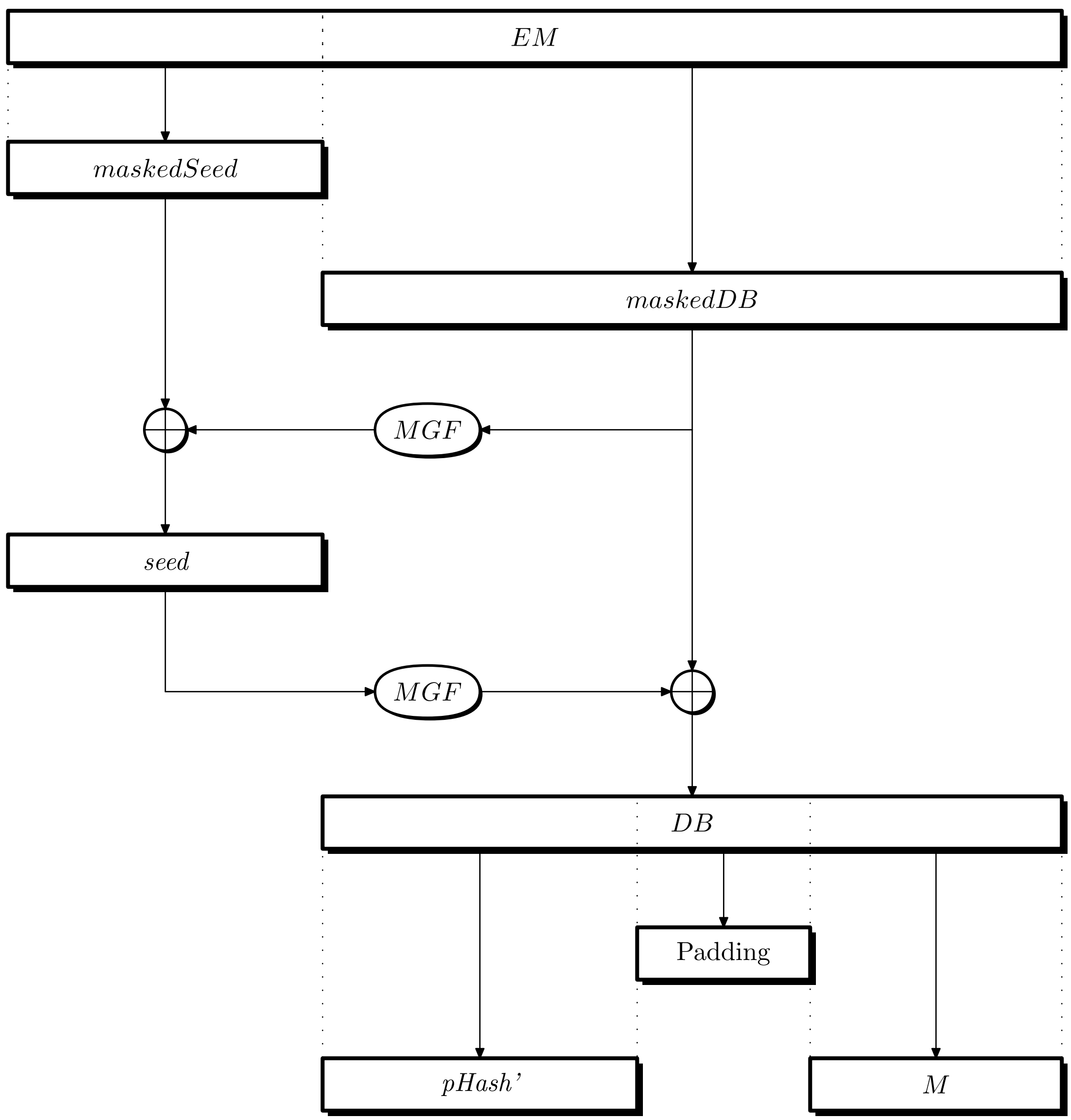

Figure C.2: *EME-OAEP decoding operation. If pHash' equals the hash of the encoding parameters, then decryption is considered successful.*