Independent Submission
Internet Draft
Intended status: Informational
Expires: May 6, 2013

M. Kutylowski, P. Kubiak
Wroclaw University of Technology
M. Tabor, D. Wachnik
Trusted Information Consulting

November 2, 2012

Mediated RSA cryptography specification for additive private key splitting (mRSAA) draft-kutylowski-mrsa-algorithm-03

Status of this Memo

Copyright Notice

Abstract

This document describes recommendations for the implementation of public key cryptography based on the mediated RSA algorithm. The Mediated RSA algorithm bases on fragmentation of a private key. As a result the signature process consists from multiple stages. The verification process is the same as in the case of RSA algorithm [RFC3447].

Table of Contents

1. Introduction 3
2. Conventions used in this document 4
3. Key types 7
3.1. RSA public key 7
3.2. Mediated RSAA private key 7
3.2.1. User's mRSAA private key 8
3.2.2. Finalization service's mRSAA private key 8
4. Data conversion primitives 9
4.1. I2OSP 9
4.2. OS2IP 10
5. Cryptographic primitives 10
5.1. Key generation primitives 10
5.1.1. MRSAA_F_GP 11
5.1.2. MRSAA_U_GP 12
5.2. Encryption and decryption primitives 13
5.2.1. MRSAA_F_DP 13
MRSAA_F_DP(Kf, c) 13
5.2.2. MRSAA_U_DP 14
MRSAA_U_DP(Ku, mp, c) 14
5.2.3. MRSAA_EP 14
5.3. Signature and verification primitives 15
5.3.1. MRSAA_U_SP1 15
5.3.2. MRSAA_F_SP1 16
5.3.3. MRSAA_VP1 17
6. Overview of schemes 17
7. Key generation schemes 18
7.1. Key generation operation 19
7.1.2. Key generation on the finalization service's side . . . 19
8. Encryption schemes 20
8.1. MRSAAES-OAEP 21
8.1.1. Encryption operation 21
8.1.2. Decryption operation 21
8.2. MRSAAES-PKCS1-v1_5 23
8.2.1. Encryption operation 23
8.2.2. Decryption operation 24
9. Signature schemes with appendix 25

9.1. MRSAASSA-PSS 26
9.1.1. Signature generation operations 26
9.1.2. Signature verification operation 28
9.2. MRSAASSA-PKCS1-v1_5 28
9.2.1. Signature generation operations 28
MRSAA_F_PKCS1_v1_5_SIGN (Kf, Sp, EM) 29
9.2.2. Signature verification operation 30
10. Encoding method for signatures with appendix 30
11. Security Considerations 30
11.1. Identification of assets and actors 31
11.2. Key generation security 32
11.2.1. Key generation by a separate yet distributed service. 33
11.2.2. Key generation by the a separate, centralized service 34
11.2.4. Key generation on user's device 38
11.2.5. Key generation directly by the finalization service. . 39
11.3. Replacement of a finalization service 39
11.4. Signature creation process security 41
11.5. Decryption process security 45
11.6. Short summary of possible security techniques 45
12. IANA Considerations 46
13. Conclusions 46
14. References 47
14.1. Normative References 47
14.2. Informative References 47
15. Acknowledgments 49
Appendix A. Pseudorandom generator primitives . . 50
Appendix B. Standard RSA primitives 51
B.1. Encryption and decryption primitives 51
B.1.1. RSAEP((n,e),m) 51
B.1.2. RSADP(K,c) 51
B.2. Signature and verification primitives 52
B.2.1. RSASP1(K,m) 52
B.2.2. RSAVP1((n,e),s) 52
Appendix C. ASN.1 Syntax 53
C.1. MRSAA key representation 53
C.1.1. MRSAA public key syntax 53
C.1.2. MRSAA private key syntax 53
Appendix D. ASN.1 Module 55
Appendix E. Intellectual Property Considerations . 56

1. Introduction

This memo is the extension to the RFC 3447. This document describes aspects specific to the mediated RSA signature scheme such as:

- Key generation
- Signature creation

The recommendations are intended for general application within computer and communications systems, and as such include a fair amount of flexibility. It is expected that application standards based on these specifications may include additional constraints.

mRSA algorithm may exists in many versions, depending on the way how the private RSA exponent is split between parties. We might distinguish two main cases (cf. sect. 1 of [7]):

- mRSAA for additive splitting of the private exponent:

  d \equiv du + df (mod lcm(p-1,q-1)) for some integers du, df,

- and mRSAM if the private exponent is divided multiplicatively:

  d \equiv du' * df' (mod lcm(p-1,q-1)) for some integers du', df' coprime with lcm(p-1,q-1).

Since each of the above possibilities determines different communication content between the user and the finalization service during signature generation (this imposes constraints on interoperability of implementations of mRSA schemes), and since addition gives more flexibility in choice of a key generation procedure, this document covers only the additive scheme.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [1].

(n, e)	RSA public key
c	ciphertext representative, an integer between 0 and n-1
cp	ciphertext created using du key, an integer between 0 and n-1
C	ciphertext, an octet string
Cp	ciphertext created using du, an octet string

d	private exponent of a user, an integer such that df + du \equiv d (mod \lambda(n))
df	private component of a finalization service, a positive integer such that df + du \equiv d (mod lambda(n))
dP	p's exponent, a positive integer such that: e(dP)\equiv 1 (mod(p-1)) NOTE: The dP value is not used in the MRSAA algorithm. It has been placed for compliancy with the RSA specification.
dQ	q's exponent, a positive integer such that: e(dQ)\equiv 1 (mod(q-1)) NOTE: The dQ value is not used in the MRSAA algorithm. It has been placed for compliancy with the RSA specification.
e	public exponent
EM	encoded message, an octet string
emLen	intended length in octets of an encoded message
Fm	finalization service's master key
H	hash value, an output of Hash
Hash	hash function
hLen	output length in octets of hash function Hash
K	RSA private key, we assume that that meaningful fields of the K are at least (n,e,d,p,q)
Kf	RSA private key which consists of (n, df)
Ku	RSA private key which consists of (n, du)
k	length in octets of the modulus n
l	intended length of octet string
lcm(.,.)	least common multiple of two nonnegative integers

m	message representative, an integer between 0 and n-1
mp	message decrypted using df key, an integer between 0 and n-1
M	message, an octet string
MGF	mask generation function
n	RSA modulus, n=pq
P	encoding parameters, an octet string
p,q	prime factors of the modulus
qInv	CRT coefficient, a positive integer less than p such: q(qInv)\equiv 1 (mod p) NOTE: This value is not used in the MRSAA algorithm. It has been placed for compliancy with the RSA specification.
s	signature representative, an integer between 0 and n-1
Sp	signature representative, created using du key, an integer between 0 and n-1
S	signature, an octet string
Sp	signature created using du key, an octet string
Uid	user identifier, an octet string
x	a nonnegative integer
X	an octet string corresponding to x
\abs(x)	absolute value of argument x
\eea(a,b)	an extended Euclidean algorithm which gives a result that satisfies: result*a\equiv gcd(a,b) mod b, where gcd(a,b) is the greatest common divisor of arguments a and b.

\floor(x)	the greatest integer not greater than real number x
\lambda(n)	lcm(p-1, q-1), where n = pq
\log_2(x)	logarithm in base 2 from positive number x
\xor	bitwise exclusive-or of two octet strings
||	concatenation operator
||.||	octet length operator

3. Key types

This document employs two types of key: an RSA public and RSA private key. The RSA private key is divided into two fragments, one fragment is stored privately by the user the other is privately recalculated by the finalization service on finalization requests.

The public key is not fragmented and thus is independent from private key fragmentation schema.

3.1. RSA public key

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

n, the modulus, an odd, positive integer

e, the public exponent, an odd, positive integer

In a valid RSA public key, the modulus n is a product of two odd primes p and q, and the public exponent e is an integer between 3 and n-1 satisfying gcd(e, \lambda(n)) = 1, where \lambda(n) = lcm (p-1,q-1).

3.2. Mediated RSAA private key

This document defines mediated RSA key as a set of two keys:

- User's mRSAA private key
- Finalization service's mRSAA private key

These keys are created from standard RSA key (the base key) which in

particular includes:

o p, q, prime factors of the public modulus n, that is pq=n

o d, the private exponent, a nonnegative integer

In the valid RSA private key, the product pq equals the modulus n from the corresponding public key, and the private exponent d is a positive integer less than \lambda(n) satisfying:

ed \equiv 1 (mod \lambda(n)).

In mediated RSA variant the private exponent is split between two parties: between the user and the finalization service. In mRSAA the split is done to satisfy the following relation:

d \equiv df + du (mod \lambda(n)).

NOTE: In this document we require that df is longer than modulus n, hence above we have a congruence modulo \lambda(n) instead of equality of integers.

Although, it is possible to use a different key splitting method, this document covers only the method based on the addition (additive scheme).

3.2.1. User's mRSAA private key

User's key consists of two components:

o n, the modulus, a positive integer, the same one as in the public key

o du, user's private exponent, an integer

In a valid mRSAA user's private key the user's private component meets the following equation:

df + du \equiv d (mod \lambda(n))

The process of key generation is described in section 7.

Because some key generation techniques might yield du being a negative integer (cf. sect. 11.2.), we do not exclude du < 0 in the document

3.2.2. Finalization service's mRSAA private key

For the purposes of this document the finalization service's private key Kf consists of the pair (n,df) where the components have the following meaning:

- n, the modulus, a positive integer, the same one as in the public key
- df, the finalization service's private exponent, a positive integer

In a valid mRSAA finalization service's private key, the finalization service's private component meets the following equation:

df + du \equiv d (mod \lambda(n))

4. Data conversion primitives

In this document are used following data conversion primitives:

I2OSP: Integer-to-Octet-String primitive

OS2IP: Octet-String-to-Integer primitive

This document describes briefly syntax of data conversion primitives. More detailed description can be found in [RFC3447].

For the purposes of this document, and consistent with ASN.1 syntax, an octet string is an ordered sequence of octets (eight-bit bytes). The sequence is indexed from first (conventionally, leftmost) to last rightmost). For purposes of conversion to and from integers, the first octet is considered the most significant in the following conversion primitives.

4.1. I2OSP

I2OSP converts a nonnegative integer to an octet string of a specified length.

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; or "integer too large"

4.2. OS2IP

OS2IP converts an octet string to a nonnegative integer.

OS2IP (X)

Input:

X octet string to be converted

Output:

x corresponding nonnegative integer

5. Cryptographic primitives

Cryptographic primitives are basic mathematical operations on which cryptographic schemes can be built. They are intended for implementation in hardware or as software modules, and are not intended to provide security apart from a scheme.

Five types of primitive are specified in this document, organized in sections: key generation; encryption and decryption; and signature and verification.

The specifications of the primitives assume that certain conditions are met by the inputs, in particular that public and private keys are valid.

5.1. Key generation primitives

The purpose of using key generation primitives is to create mRSAA key pair: a public key and pair of private keys: a finalization service's key and a user's key.

The finalization service's exponent df is derived from a master service key Fm and a user identifier Uid. The derivation is a

asymmetric signature key then generation process of exponent df might be as follows: Uid is signed with key Fm, and this signature is a seed for pseudorandom string generation (cf. deterministic seed for generation of pseudorandom bits in paper [14]). The resulting string df (output of the generator) should have bitlength defined as follows:

$$\floor(\log_2(n))+1 + Delta$$

where Delta is a fixed value taken from the set {80,...,128}. That is df should be longer by Delta bits from the length of the modulus n. The aim of taking a longer bitstring df is to equalize chances of each single value modulo \lambda(n) to be chosen as the remainder rem in the relation

$$df \equiv rem \mod \lambda(n)$$

Note that the finalization service does not know the value lambda(n). The greater Delta is the tighten equalization is achieved.

Note that in case of asymmetric method there is no need to make public the key complementary to Fm, that is the signature verification key. The complementary key should remain secret to prevent any cryptoanalytic attempts: one option is to use it internally in the system for the purpose of verification of implementation correctness (e.g., of subliminal channel freeness), another possibility is to completely erase the complementary key.

Generation of the pseudorandom bit string uses deterministic random bit generator which bases on block ciphers algorithms (CTR_DRBG). CTR_DRBG's primitives are specified in the paper [3]. For the purposes of this document, primitives specified in [3] have been briefly described in Appendix A.

The user's private key is created on a basis of the finalization service's private key (n,df) and the RSA base key K.

5.1.1. MRSAA_F_GP

MRSAA_F_GP generates pseudorandom key of requested length on a basis of a given bit string. This document describes MRSA_F_GP as a primitive which is based on the CRT_DRBG pseudorandom generator. An implementer may use other pseudorandom generator type conformant to [3].

MRSAA_F_GP(w, len)

w	the string of bits received from the consuming application
len	Requested bitlength of the output string

Output:

returned_bits	pseudorandom bit string with length specified in len

Assumptions:

CRT_DRBG generator conformant to [3] is used.

Steps:

1. Let H = Hash(w)
2. Let (V, Key, reseed_counter) = CTR_DRBG_Instantiate_algorithm(null, H)
3. Let (status, returned_bits, Key, V, reseed_counter) = CTR_DRBG_Generate_algoritm(reseed_counter, len, null)
4. If the status is SUCCESS output returned_bits and stop. In a different case return an error

5.1.2. MRSAA_U_GP

Creates user's private exponent from finalization service's private exponent and a base RSA key.

NOTE: For alternative procedures see sect.11.2.

MRSAA_U_GP(K,df)

Input:

K	Base RSA private key which contains values (n, d, p, q)
df	Private exponent of finalization service's key

Output:

du	Private exponent of user's key

Assumptions:

The K is a valid RSA private key corresponding to the public key (e,n) for which df has been calculated

Steps:

1. Let lambda = (p-1)*(q-1)/gcd(p-1,q-1).
2. Let du = (d - df) mod lambda

5.2. Encryption and decryption primitives

An encryption primitive produces a ciphertext from a message representative under a control of public key. The primitive MRSAA_F_DP is used to transform a ciphertext to a form which can be processed by the MRSAA_U_DP primitive. The primitive MRSAA_U_DP recovers a message from MRSAA_F_DP output.

The encryption/decryption scheme involves all three primitives described below.

Primitives RSAEP and RSADP are briefly described in the Appendix B.

MRSAA_EP is identical as a RSAEP primitive and thus they can be used interchangeably.

5.2.1. MRSAA_F_DP

MRSAA_F_DP transforms ciphertext to the form which can be decrypted by a MRSAA_U_DP primitive.

MRSAA_F_DP(Kf, c)

Input:

Kf	Finalization service's private key in the form of a pair of (n, df)
c	ciphertext representative, an integer between 0 and n-1

Output:

mp	message partially decrypted using df key, an integer between 0 and n-1

Assumptions:

The Kf key is a valid mRSAA key which corresponds to the user's identifier Uid.

Steps:

1. Let mp = RSADP(Kf, c)

5.2.2. MRSAA_U_DP

Retrieves message from partially deciphered message (result of an MRSAA_F_DP)

MRSAA_U_DP(Ku, mp, c)

Input:

Ku	user's private key in the form of a pair of (n, du)
mp	message partially decrypted using df key, an integer between 0 and n-1
c	ciphertext representative, an integer between 0 and n-1

Output:

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

Assumptions:

The Ku key is a valid mRSAA key (n, du)

Steps:

1. Let Ku' = (n,\abs(du))
2. Let temp = RSADP(Ku', c)
3. If (du < 0) temp = \eea(temp,n)
4. Let m = mp*temp mod n

5.2.3. MRSAA_EP

MRSAA_EP encrypts given message

MRSAA_EP((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; or "message representative out of range"

Assumptions:

The public key (n, e) is valid

Steps:

1. Let c = RSAEP((n,e) , m)

5.3. Signature and verification primitives

A verification primitive works in similar way like an RSAVP1 primitive. Signature process uses two primitives specified below: an MRSAA_U_SP1 and MRSAA_F_SP1 primitive. MRSAA_U_SP1 primitive creates partial signature, which cannot be verified before execution of the MRSAA_F_SP1 primitive.

Signature/verification scheme uses all three primitives specified below.

Verification primitive is identical as an RSAVP1 primitive and thus RSAVP1 can be used instead of MRSAA_VP1.

Primitives RSAVP1, RSASP1 are briefly described in Appendix B.

For the final signature creation step (MRSAA_F_SP1) we strongly recommend to perform signature verification before passing the

assume that signature verification includes verification of encoding correctness.

5.3.1. MRSAA_U_SP1

MRSAA_U_SP1 creates partial signature from given message.

MRSAA_U_SP1(Ku,m)

Input:

Ku	user's private key in the form of a pair of (n, du)
m	message representative, an integer between 0 and n-1

Output:

sp	representative of a partial signature, an integer between 0 and n-1, created using Ku key

Assumptions:

The Ku key is a valid mRSAA key which corresponds to the user's identifier Uid and public key (n,e)

Steps:

1. Let Ku' = (n,\abs(du))
2. Let sp = RSASP1(Ku', m)
3. If (du < 0) then sp = \eea(sp,n) mod n

5.3.2. MRSAA_F_SP1

MRSAA_F_SP1 creates valid RSA signature from given partial signature.

MRSAA_F_SP1(Kf, sp, m)

Input:

Kf	Finalization service's private key in the form of a pair (n, df)
sp	representative of a partial signature, an integer between 0 and n-1, created using Ku key
m	message representative, an integer between 0 and n-1

Output:

s	signature representative, an integer between 0 and n-1

Assumptions:

The Kf key is a valid mRSAA key which corresponds to the key Ku

Steps:

1. Let temp = RSADP1(Kf, m)
2. Let s = temp*sp mod n

5.3.3. MRSAA_VP1

MRSAA_VP1 retrieves a message from a given signature

MRSAA_VP1((n, e), s)

Input:

(n, e)	RSA public key
s	signature representative, an integer between 0 and n-1

Output:

c	message representative, an integer between 0 and n-1

Assumptions:

The public key (n, e) is valid

Steps:

1. Let m = RSAVP1((n, e) , s)

6. Overview of schemes

A scheme combines cryptographic primitives and other techniques to

achieve a particular security goal. Two types of scheme are specified in this document: encryption schemes and signature schemes with appendix.

The schemes specified in this document are limited in scope in that their operations consist only of steps to process data with an mRSAA public or private key. Additionally steps necessary for private key splitting have been described (see section 7.1.). Thus, in addition to the scheme operations, an application will typically include key management operations by which parties may select mRSAA public and private keys for a scheme operation. The specific additional operations and other details are outside the scope of this document.

As was the case for the cryptographic primitives (Section 5), the specifications of scheme operations assume that certain conditions are met by the inputs, in particular that mRSAA public and private keys are valid. The behavior of an implementation is thus unspecified when a key is invalid. The impact of such unspecified behavior depends on the application. In general possible means of addressing key validation include:

- explicit key validation by the application;
- key validation within the public-key infrastructure;
- assignment of liability for operations performed with an invalid key to the party which generated the key.

In case of mRSAA signatures the key validation could be performed by verifying the first finalized signature using the public key. To validate mRSAA decryption keys, decryption of some test message may be performed.

7. Key generation schemes

A key generation scheme combines a MRSAA_U_GP and MRSAA_F_GP with a RSASP1 operation to create MRSAA private key pair for user and finalization service.

The finalization service's key can be derived from a service key Fm and user identifier Uid. This kind of solution eliminates necessity of key archival on the finalization service's side. To derive private key from service's master key Fm some secure pseudorandom numbers generator is used. Fm is described below as an asymmetric signature key of a deterministic scheme. However, since Fm is used internally in the system, we do not exclude possibility of implementing other secure derivation methods of exponents df. As stated in section 5.1. we recommend the key complementary to asymmetric signature key Fm to

remain secret.

In key generation schemes described below we do not assume that all input variables are available at the same location, and that all output variables are generated at the same place.

7.1. Key generation operation

7.1.1. Generation of the private key assigned to identifier Uid of a user

MRSAA_U_GS-DRBG-GENERATE (K,df)

Input:

K	Base RSA private key which contains values (n, d, p, q)
df	Finalization services exponent calculated for the user of identifier Uid

Output:

Ku	RSA key (n, du) such that du + df \equiv d \lambda(n), and e*d \equiv 1 \lambda(n), where (n,e) is the public key for which exponent df has been calculated on finalization service side

Steps:

1. Apply an MRSAA_U_GP to generate user's MRSAA private exponent du:
   du = MRSAA_U_GP(K,df)
2. Return (n,du) as Ku

7.1.2. Key generation on the finalization service's side

MRSAA_F_GS-DRBG-GENERATE generates finalization service's private key. An implementer may use other deterministic signature algorithms (i.e. RSASSA-PKCS-v1_5-SIGN) or other methods to generate df exponent.

MRSAA_F_GS-DRBG-GENERATE (n, Uid, Fm)

Input:

n	public RSA modulus of the user and the identifier Uid
Uid	User identifier, an octet string
Fm	Finalization service's master key, a RSA key

Output:

Kf	Finalization service's key, RSA key (n, df) which together with key Ku shall satisfy the congruence: df + du \equiv d mod \lambda(n), where d is the private exponent corresponding to the public key (n,e) of the user of identifier Uid, that is d*e \equiv 1 mod \lambda(n)

Assumptions:

The finalization service uses a fixed value Delta, an integer from the set {80,...,128} (see remarks in sect. 5.1.).

In the case when Kf is generated anew for each signature finalization or/and for each decryption operation concerning the user of identifier Uid, that is why the process must be deterministic.

1. Steps:Apply the RSASP1 primitive to Fm key and Uid integer representative value to create signature W, with salt of length 0 (cf. [RFC3447], point 4 on page 37), or with salt being a fixed value (cf. [RFC3447], sect. 8.1 page 29):
   W = RSASSA-PSS-SIGN(Fm, Uid)
2. Convert W octet string to an integer representative w:
   w = OS2IP(W)
3. Apply an MRSAA_F_GP to generate finalization service's MRSAA private exponent df:
   df = MRSAA_F_GP(w, \floor(\log_2(n))+1 + Delta)
4. Return (n, df) as Kf

8. Encryption schemes

For the purpose of this document, an encryption scheme consists of an encryption operation and decryption operation, where the encryption operation produces a ciphertext from a message with a recipient's RSA public key, and the decryption operation recovers original message in two stages: transformation made by finalization service, and decryption performed by an recipient.

The encryption scheme can be employed in variety of applications, especially in systems with different level access to confidential

information. The scheme assures additional level of protection because of a usage of a finalization service.

The document extends existing RSA encryption and decryption scheme, particularly the decryption scheme, which differs from the RSA scheme.

The Encryption scheme combines encryption and decryption primitives with an encoding method for encryption. The encryption operations apply a message encoding operation to a message to produce an encoded message, which is then converted to an integer message representative. An encryption primitive is applied to the message representative to produce the ciphertext. Reversing this, the decryption operations apply a decryption primitive to the ciphertext to recover a message representative, which is then converted to an octet string encoded message. A message decoding operation is applied to the encoded message to recover the message and verify the correctness of the decryption.

8.1. MRSAAES-OAEP

MRSAA-OAEP combines RSAEP, MRSAA_U_DP, and MRSAA_F_DP primitives (Sections 5.2.2. 5.2.) with EME-OAEP encoding method specified in [RFC3447].

8.1.1. Encryption operation

There are no differences in encryption operation (RSAES-OAEP-ENCRYPT) according to [RFC3447].

8.1.2. Decryption operation

MRSAA-F-OAEP-DECRYPT(Kf, C)

Options:

Hash	Hash function (hLen denotes the length in octets of the hash function output)
MGF	Mask generation function

Input:

Kf	Finalization service's private key (k denotes the length in octets of the RSA modulus n)

C	Ciphertext to be decrypted, an octet string of length k, where k >= 2hLen + 2

Output:

Cp	Transformed ciphertext, which can be deciphered using Ku key

Assumptions:

The Kf key is generated with MRSAA_F_GS-DRBG-GENERATE and corresponds to the public key which has been used to produce the ciphertext C.

Steps:

1. Length checking
   a. If a the length of the ciphertext C is not k octets, output "decryption error" and stop.
   b. If k < 2hLen + 2, output "decryption error and stop"
2. RSA decryption
   a. Convert the ciphertext C to an integer ciphertext representative c (see Section 4.2.):
      c = OS2IP(C)
   b. Apply the MRSAA_F_DP decryption primitive (Section 5.2.1.)to the mRSAA private key Kf and ciphertext representative c to produce an integer representative cp:
      cp = MRSAA_F_DP(Kf,c)
   c. Convert the transformed ciphertext cp to an encoded transformed ciphertext Cp of length k octets:
      Cp = I2OSP(cp,k)
   d. Output the Cp

MRSAA-U-OAEP-DECRYPT(Ku, Cp, C, L)

Input:

Ku	User's private key (k denotes the length in octets of the RSA modulus n)
Cp	Transformed ciphertext, to be decrypted
C	Ciphertext, an octet string
L	Optional label whose association with the message is to be verified; the default value for L, if L is not provided, is the empty string

Output:

M message, an octet string of length mLen, where mLen <= k - 2hLen - 2

Steps:

1. RSA decryption
   a. Convert the ciphertext Cp to an integer ciphertext representative cp (see Section 4.2.):
      cp = OS2IP(Cp)
   b. Convert the ciphertext C to an integer ciphertext representative c (see Section 4.2.
      c = OS2IP(C)
   c. Apply the MRSAA_U_DP decryption primitive (Section 5.2.2.)to the RSA private key Ku and ciphertext representative cp to produce an integer representative m' of an encoded message:
      m' = MRSAA_U_DP(Ku, cp, c)
   d. Convert the encoded message representative m' to an encoded message M' of length k octets:
      M' = I2OSP(m',k)
2. EME-OAEP decoding message M from string M':

   Process is performed on M' as described in [RFC3447]

3. Output the message M

8.2. MRSAAES-PKCS1-v1_5

MRSAAES-PKCS1-v1_5 combines the RSAEP and MRSAADP primitives with the EME-PKCS1-v1_5 encoding method. RSAES-PKCS1-v1_5 can operate on messages length up to k-11 octets, although care should be taken to avoid certain attacks on low-exponent RSA due to Coppersmith, et al. when long messages are encrypted (see [4]). As general rule, the use of this scheme for encrypting an arbitrary message, as opposed to a randomly generated key, is not recommended. Moreover, one must still avoid encrypting the same random key with relatively short, different randomizers, to some number of recipients sharing the same small public exponent (see section 5 and appendix C of [15]).

8.2.1. Encryption operation

There are no differences between MRSAA and RSA encryption process and thus RSAES-PKC1-V1_5-ENCRYPT should be used. Detailed specification can be found in the [RFC3447]

8.2.2. Decryption operation

Input:

Kf	Finalization service's private key (k denotes the length in octets of the RSA modulus n)
C	Ciphertext to be decrypted, an octet string of length k, where k is the length in octets of the RSA modulus n

Output:

Cp	Transformed ciphertext, which can be deciphered using Ku key

Assumptions:

The Kf key is generated with MRSAA_F_GS-DRBG-GENERATE and corresponds to the public key which has been used to produce the ciphertext C.

Steps:

1. Length checking: If the length of the ciphertext C is not k octets (or if k < 11), output "decryption error" and stop.
2. RSA decryption
   a. Convert the ciphertext C to an integer ciphertext representative c (see Section 4.2.):
      c = OS2IP(C)
   b. Apply the MRSAA_F_DP decryption primitive (section 5.2.1.)to the mRSAA private key Kf and ciphertext representative c to produce an integer representative cp:
      cp = MRSAA_F_DP(Kf,c)
   c. Convert the transformed ciphertext cp to an encoded transformed ciphertext Cp of length k octets:
      Cp = I2OSP(cp,k)
3. Output Cp

MRSAAES-U-PKCS-1-V1_5-DECRYPT (Ku, CP, C)

Input:

Ku	User's private key (k denotes the length in octets of the RSA modulus n)
Cp	Transformed ciphertext, to be decrypted

C	Original ciphertext, octet string
M	message, an octet string of length at most k - 11

Steps:

1. Length checking: If the length of the ciphertext C is not k octets (or if k < 11), output "decryption error" and stop.
2. RSA decryption
   a. Convert the ciphertext C to an integer ciphertext representative c (see Section 4.2.):
      c = OS2IP(C)
   b. Convert the transformed ciphertext Cp to an integer ciphertext representative cp (see Section 4.2.):
      cp = OS2IP(Cp)
   c. Apply the MRSAA_U_DP decryption primitive (section 5.2.2.)to the RSA private key Ku and transformed ciphertext representative cp and original ciphertext c to produce an integer representative of an encoded message m':
      m' = MRSAA_U_DP(Ku,cp, c)
      If MRSAA_U_DP outputs "ciphertext representative out of range" (meaning that cp>=n), output "decryption error" and stop.
   d. Convert the encoded message representative m' to an encoded message M' of length k octets:
      M' = I2OSP(m',k)
3. EME-PKCS1-v1_5 decoding message M from string M'
   Performed on string M' as specified in [RFC3447].
4. Output M

9. Signature schemes with appendix

For the purposes of this document signature schemes with appendix consists of three main operations: signature verification and two complementary operations of signature generation. Signature verification produces a message from signature value using RSA public key, and signature operations creates a signature value from a message value and from the intermediate product, i.e., from a partial signature. The partial signature is created using a user's private key, and final signature value is created using finalization service's private key corresponding to the public key of the user.

Two signature schemes are specified in this document: MRSAASA-PSS and MRSAASA-PKCS1-v1_5. Both of them base on an corresponding RSA scheme specified in [RFC3447].

9.1. MRSAASSA-PSS

MRSAASSA-PSS combines MRSAA_U_SP1, MRSAA_F_SP1 and RSAVP1 which is The MRSAA_U_SP1 primitive is identical to the RSASP1 primitive, and thus it is sufficient to implement only RSASP1.

The length of messages on which MRSAASSA-PSS can operate is either unrestricted or constrained by a very large number, depending on the hash function underlying the EMSA-PSS encoding method.

To make signature verification by the finalization service possible (see remark in Section 5.3.) we strongly recommend adding at least mHash to the list of arguments passed to the finalization service. Note that according to [RFC3447] (page 37, point 3) the EMSA-PSS encoding is secure even if an adversary can freely choose mHash. Thus to utilize strength of the EMSA-PSS encoding verification must at least comprise verification of encoding correctness.

9.1.1. Signature generation operations

MRSAASSA_U_PSS_SIGN (Ku, M)

Input:

Ku	Signer's RSA private key
M	Message to be signed, an octet string

Output:

Sp	Partial signature, an octet string of length k, where k is the length in octets of the RSA modulus n
EM	Message encoded using EMSA-PSS-ENCODE primitive (see [RFC3447] paragraph 9.1.1)

Errors: "message too long;" "encoding error"

Steps:

1. Let EM = EMSA-PSS-ENCODE(M, modBits -1)
2. Convert the encoded message into integer message representative m: m=OS2IP(EM)
3. Apply the RSASP1 signature primitive to the key Ku and Message M: sp = RSASP1(Ku, m)
4. Convert the signature representative sp into partial signature value Sp with the length of k octets. Where k is the length in octets of the RSA modulus n

Sp = I2OSP(sp, k)
5. Output Sp, EM

MRSASSSA_F_PSS_SIGN (Kf, Sp, EM)

Input:

Kf	Finalization service's RSA private key
Sp	Partial signature to be transformed into valid signature
EM	Message encoded using EMSA-PSS-ENCODE primitive (see [RFC3447] paragraph 9.1.1)

Output:

S	signature, an octet string of length k, where k is the length in octets of the RSA modulus n

Errors: "message too long;"

Assumptions:

The Kf key is generated with MRSAA_F_GS-DRBG-GENERATE and corresponds to the public key which will be used to verify the signature S.

Steps:

1. RSA signature

   a. Convert the partial signature Sp to an integer signature representative sp:
      sp = OS2IP(Sp)
   b. Convert the pss-encoded message EM to an integer representative m:
      m = OS2IP(EM)
   c. Apply the MRSAA_F_SP1 signature primitive (see Section 5.3.2.) to the RSA private key Kf and the partial signature representative sp to produce a signature representative s:
      s = MRSAA_F_SP1(Kf, sp, m)
   d. Convert the signature representative s to a signature S of length k octets (see section 4.1.):
      S = I2OSP(s, k)

2. Output the signature S

9.1.2. Signature verification operation

There are no differences between MRSASSSA-PSS-VERIFY and RSASSA-PSS-VERIFY operation and thus RSASSA-PSS-VERIFY should be used. The RSASSA-PSS-VERIFY operation has been specified in [RFC3447] section 8.1.2.

9.2. MRSAASSA-PKCS1-v1_5

MRSAASSA-PKCS-v1_5 combines MRSAA_U_SP1, MRSAA_F_SP1 and MRSAA_VP1 with EMSA-PKCS1-v1_5 encoding method. Since MRSAA_U_SP1 is identical to MRSAASP and MRSAA_VP1 is identical to RSAVP1, RSA version of primitives is used in specification.

However, as paper [16] indicates, fixed pattern padding method is relatively weak (note that for a fixed hash function PKCS-v1_5 is a fixed pattern encoding). The attack from [16] allows producing valid signatures under a message of attacker's choice (it might be a hash of some meaningful message), if the following conditions are satisfied simultaneously:

- hash of the message is longer than one-third of the length of the modulus,
- a victim has signed three encoding results without verifying that the values present in the hash-block of encoding are really hashes of some messages. The three values are related with the final signature forged by the adversary.

Although the attack seems to be theoretical (the victim must sign three artificial messages to make the adversary able to produce a single meaningful forgery), it exhibits weakness of this encoding method. To mitigate this kind of attacks we recommend to use hashes that are shorter than one-third of the RSA modulus, and to make the finalization service capable of checking that the value present in the hash-block is really a hash of some message.

Verification process in the scheme is identical to the RSASSA-PKCS1-v1_5 verification process. The signing process consists of two stages: intermediate signature generation (presignature), which is done in similar way to the RSA scheme, and signature finalization, where MRSAA_F_SP1 primitive is used in combination with EMSA-PKCS1-v1_5 encoding.

9.2.1. Signature generation operations

MRSAASSA_U_PKCS1_v1_5_SIGN (Ku, M)

Input:

Ku	Signer's RSA private key
M	Message to be signed, an octet string

Output:

Sp	Partial signature, an octet string of length k, where k is the length in octets of the RSA modulus n
EM	Pkcs1-encoded message (see [RFC3447] section 9.2)

Errors: "message too long"; "RSA modulus too short"

Assumptions:

The Kf key is generated with MRSAA_F_GS-DRBG-GENERATE and corresponds to the public key which will be used to verify the signature S.

Steps:

1. Let EM = EMSA-PKCS1-V1_5-ENCODE (M, k)
2. Convert EM value into message representative m:
   m = OS2IP(EM)
3. Apply RSASP1 signature primitive to the key Ku and message m to obtain partial signature value Sp:
   Sp = RSASP1(Ku,m)
4. Output Sp, EM

MRSAA_F_PKCS1_v1_5_SIGN (Kf, Sp, EM)

Input:

Kf	Finalization service's RSA private key
Sp	Partial signature to be transformed into valid signature
EM	Pkcs1-encoded message (see [RFC3447] section 9.2)

Output:

S	signature, an octet string of length k, where k is the length in octets of the RSA modulus n

Errors: "message too long;"

Steps:

1. RSA signature
   a. Convert the partial signature Sp to an integer signature

      sp = OS2IP(Sp)
   b. Convert the encoded message EM to an integer representative m:
      m = OS2IP(EM)
   c. Apply the MRSAA_F_SP1 signature primitive (see section 5.3.2.) to the RSA private key Kf, the partial signature representative sp and message representative m to produce a signature representative s:
      s = MRSAA_F_SP1(Kf, sp, m)
   d. Convert the signature representative s to a signature S of length k octets (see section 4.1.):
      S = I2OSP (s, k)
2. Output the signature S

9.2.2. Signature verification operation

There are no differences between MRSAASSA-PKCS1-v1_5-VERIFY and RSASSA-PKCS1-v1_5-VERIFY operation and thus RSA version should be used. RSASSA-PKCS1-v1_5-VERIFY operation has been specified in [RFC3447] section 8.2.2.

10. Encoding method for signatures with appendix

The encoding methods specified in [RFC3447] in section 9. should be used.

11. Security Considerations

Formal proof of security of two-party RSA signatures in the random oracle model is given in the paper [7]. The proof applies both to multiplicative and to additive private exponent splitting.

However, the paper [7] assumes that the key material is securely generated, distributed and stored. It does not investigate for example side channel or fault injection attacks into parts of the system involved in generation, distribution of keys and replacement of old key material.

Some issues of designing, implementing and maintaining a secure system that is demonstrably trustworthy are depicted for example in papers [8], [9].

Obviously, building such a system is not a trivial task, and many factors have to be taken into consideration.

The following sections contain an analysis of those factors and give recommendations, what kind of controls could be taken into consideration when building such a system.

The subsequent sections identify the assets which should be protected and describe security considerations which should be taken into account. Security considerations are identified for particular processes such as: key generation, signature generation etc.

11.1. Identification of assets and actors

The main advantage of the MRSAA scheme over the RSA scheme is the private exponent fragmentation. This allows the trusted third party to confirm user's right to perform MRSAA operation. To achieve this functionality neither user nor finalization service knows a base RSA key. This can be achieved using distributed key generation techniques (see for example [10]), or by outsourcing key generation to a trusted third party. Alternatively, to facilitate implementation the base key may exist on one component for a short period of lifecycle of the keys in a secure environment and must be destroyed immediately afterwards.

After the introduction of the key generation trusted third party, the MRSAA key generation algorithm actors list is as follows:

- User's device - uses user's MRSAA key
- Finalization service - uses finalization service's MRSAA key
- Key generation service - generates base RSA key

In the MRSA algorithm the following assets can be identified:

- Fm - master key used in finalization service's exponent (df) derivation
- df - finalization service's MRSAA private exponent derived from Fm and Uid
- du - user's MRSAA private exponent

- d - base RSA private exponent
- Uid - user identifier used in derivation of df

For the MRSAA algorithm the following threats can be identified:

- Fm key's privacy violation - disclosure of the Fm key allows an attacker to generate finalization private exponent df on the basis of the Uid's. The attacker is able to create his own
- df privacy violation - disclosure of the df exponent allows an attacker to act as an finalizations service in a relation to a specific user that holds a complementary Ku key.
- du privacy violation - a disclosure of the du exponent allows an attacker to sign in behalf of the key owner. To produce such a signature, the attacker has to use finalization service to complete a signature. In the case of an encryption scheme the attacker has to obtain a partially deciphered message from finalization service.
- d exponent's privacy violation - a disclosure of the base RSA key allows an attacker to make all operations, either signature and decryption, without participation of the user and finalization service
- Uid's privacy violation - revealing Uid does not affect the MRSAA algorithm security.

11.2. Key generation security

Generation of keys referring to a particular user includes three main steps:

- Generation of user's base RSA key;
- Generation of finalization service's key Kf (the key is complementary to the user's key Ku); the procedure is usually executed anew for each signature finalization and for each partial decryption operation referring to the user;
- Generation of the user's private key Ku

MRSAA keys are distributed among parties. The security of the MRSAA algorithm bases on fact that neither user nor finalization service knows user's base RSA key (K).

Security of the generation process is critical because on this stage the base key K with private exponent d is generated. In this stage the exponent du, and hence the exponent df needed for this task, are calculated as well.

We can distinguish at least five approaches to the key generation process:

- Key generation by a separate yet distributed key generation service
- Generation of the RSA key directly by the user and the finalization service with a two-party protocol
- Generation of the RSA keys on user's device
- Key generation directly by the finalization service

11.2.1. Key generation by a separate yet distributed service.

The key generation model described in addresses the threat of disclosing the base RSA key and finalization exponent df by a single protocol participant. Distributed key generation protocols enjoys the following property: compromise of a single party does not lead to disclosure of the private key. This is achieved by representing some intermediate values of the protocol and the final private key as a set of shares, and each party knows only a single share of each shared value and of the final key. Only having collected some subset of shares of a value an adversary is able to reconstruct the value. The same refers to the private key. The cardinality of the subset must be greater than some threshold. Usually the threshold equals floor((k-1)/2), where k is the number of parties generating the key, in the two-party schemes the threshold is obviously set to one.

However, distributed protocols suffer from some drawback: total communication volume grows rapidly with the number of participants k, thus the protocols do not scale well and are useful only for relatively small values of k. Distributed protocols assume that point-to-point communication lines between parties are secured (i.e., are encrypted and authenticated).

To refer to distributed RSA-key generation, many such protocols exist in the literature. Some of them consider two-party setting [22], [23], [24], whereas other use techniques requiring at least three participants of the protocol - see e.g., [25], [26], [10].

Suppose that RSA key-pair (n,e), d was generated by k parties: P_1, P_2, ..., P_k, and the private exponent d (an integer d such that d*e \equiv 1 \lambda(n)) is represented as sum of integers:

$$d = d_1 + d_2 + ... + d_k,$$

where party P_i knows only the component d_i. Let df be split by the finalization service to the following sum of integers:

$$df = df_1 + df_2 + ... + df_k$$

Next, df_i is sent over a secure channel to party P_i. Then P_i calculates integer value d_i - df_i and sends the result over a secure channel to the device of user u. Finally, user u obtains

$$du = (d_1 - df_1) + ... + (d_k - df_k)$$

which equals d - df.

The distributed key generation is exposed to the eavesdropping. For example, an adversary who is able to get access to all of the df_i fragments is able to recover a df exponent. Access to all fragments of the d exponent gives a possibility to recreate the original d exponent.

Therefore, to secure distributed key generation mechanism the following security measures may be taken into account:

- Authenticating all parties involved in communication
- Encrypting communication channels between parties with session keys that are secret and unique for each communicating pair.

11.2.2. Key generation by the a separate, centralized service

In this case a single party, i.e., the key generation service, has a control over the base key K. Moreover, it is necessary to involve this party into generation of the user's private exponent du.

On the other hand generation of the finalization service's private exponent df should be performed by a finalization service in order to not disclose finalization service's master key Fm to other parties. Note that key Fm (and thus exponent df) is independent of the values of generated RSA (public (n,e), private d) key-pairs. Exponent df depends only on the length of the RSA modulus.

To secure key generation process the following requirements should be fulfilled:

- Fm key should be protected by a finalization service, the key must not be disclosed to the other parties
- df exponent should be generated in a secure environment by a finalization service; the key should be transferred to a key generation service in encrypted form
- du exponent should be generated in a secure manner by a key generation service; the exponent du should be transferred to the user in a encrypted form;
- K key should be generated in a secure environment by a key generation service; the key should not be transferred to other parties; private part d of the key should be destroyed

A centralized approach to the key generation problem gives an opportunity to use many security techniques to protect privacy and availability of assets. To protect privacy the following controls may be used:

- Hardware security modules
- Strong authentication and authorization between components
- Smart cards for protection of user's key du
- Encryption of messages passed between the key generation module and the user's device

To achieve high availability of the key generation environment the duplication of the Fm key on multiple devices may be considered as long as full control over usage of Fm is maintained. However, duplication of Fm rises a risk of the key being compromised.

In the described model a single party knows the complete RSA key K ((n,e), d). However, one might prevent the party from learning how the private exponent is divided between the finalization service and the user's device. In such case the following method may be used.

Let the finalization services express the positive integer df as a sum of two pseudorandom integers df_1+df_2, where both values have roughly the same length. Then df_1 might be encrypted by finalization service with the public key to (a device of) the user. We suppose that at the time of signature key generation there is some public key assigned to the device of the user (e.g., it might be the key used

for the device authentication), and that this public key might also be used for encryption purposes. Then the finalization service signs the pair (ciphertext of df_1, Uid) and both the signed pair and df_2 are sent to the key generation service. The key generation service calculates a multiple of \lambda(n) such that the sum of the multiple and d is greater than df_2. Next the service subtracts df_2 from the sum result, and passes this subtraction result (we denote it by df_2') to the user's device. Also the pair (ciphertext of df_1, Uid) and key generation service's signature under it is passed to the user's device. We assume that all messages are sent through a secure and authenticated channel. The user's device checks Uid, the signature of the key generation service, and decrypts df_1. Finally the device calculates the integer du=df_2' - df_1.

Below we shall justify some requirements concerning the channel between user's device and the finalization service. Note that pre-signature m^{du} mod n, together with the finalized signature m^d mod n both represent the DLP (discrete logarithm problem) in the multiplicative group of ring Zn:

Knowing m^d mod n one might easily obtain m. Consequently, to learn du from m^{du} mod n it suffices to solve the DLP problem in the ring Zn. Knowing factorization of n the (staff of the) key generation service might transform this problem to an equivalent pair of DLP problems: one modulo p, the other modulo q. However, comparing known attacks on factorization problem and on the DLP defined in prime fields it is obvious that these two resulting DLP problems are much easier to solve than factoring n by an external observer (n is about two times longer than each of its factors p, q). Therefore, to prevent the (staff of the) key generation service from learning du by attacking the DLP problem we advise not only authenticate, but also to encrypt the channel between the user's device and the finalization service. If the channel between the device and the finalization service is not authenticated then, having exponent du, the adversary might submit pre-signatures that shall be correctly finalized and assigned to the owner of the public key (e,n).

Both in the distributed and the centralized version of the RSA key generation service one should take into account possibility of kleptographic attacks [27]. To prevent them one might verify a fraction of randomly chosen keys being generated regarding randomness used.

Some methods of randomness verification are presented in the papers [8], [9]. Another option is to use a deterministic signature scheme as a tool to generate seeds for a pseudo-random number generator [14]. Such seeds are unpredictable, but easily verifiable in respect

to correctness.

Another issue concerns the finalization service. The issue is possibility of subliminal leakage made by finalization service's manufacturer by utilizing the relation d \equiv df + du mod lambda(n). Note that d must be odd and \lambda(n) is even, hence manipulating parity bit of df the finalization service chooses parity bit of du. If the channel between the user's device and finalization service is not encrypted, then parity of du might be easily determined by testing values of Jacobi symbol calculated by user's device on a few pre-signatures. Suppose that the finalization service is given master key Fm and the aim of the attack prepared by finalization service's manufacturer is to leak a ciphertext of Fm say AES CFB-mode ciphertext (AES encryption key might be chosen in advance by the producer). Let the infected device divide the AES ciphertext into say 48-bit blocks and let each block be loaded as an initial state of some LFSR (Linear Feedback Shift Register) secretly implemented inside finalization's service HSM. Suppose the upper limit L on the number of ciphertext's blocks is correctly estimated by service's malicious producer. Let hash of Uid indicates both the index of LFSR and the index of the bit on that LFSR's output, and let parity bit of du be value of the bit indicated by hash of Uid. pseudorandom sequence of period 2^{48}-1, hence indicated bits are sparsely distributed in LFSR's period and we expect that there would be no bit-repetition. Accordingly, outside the system parity bits of exponents du seem to be not correlated. On the other hand, stealing some number of users' devices the producer collects different bits in the sequence produced by each LFSR, that is the producer collects linear equations with unknowns being the initial state's bits. Having collected 48 independent equations for each LFSR the HSM producer calculates values of initial state of each LFSR. In such a way all blocks of the ciphertext are collected.

If output of user's device is encrypted, a powerful adversary might try to bypass encryption (cf. instruction nulling with laser beam [28] in case of smart cards). We do not exclude possibility of mounting such a sophisticated attack against key Fm. Therefore, to prevent such an attack we recommend to fix the parity bit of values df generated in the system, or to generate exponent df in a verifiable manner, for example from seeds being signatures.

11.2.3. Key generation executed directly by the user's device and the finalization service with a two-party protocol

This is a special case of distributed RSA key generation limited to two protocol participants [24],[22]. If executed properly (i.e.,

parties are curious but honest), it prevents each single participant from learning the private key. Such a solution eliminates the need of having a separate key generation service. On the other hand, a distributed protocol imposes heavy communication and computational burden on user's device and (cumulatively) on the central server. Moreover, if a more robust version of the protocol is considered (i.e., one that prevents parties from cheating - cf. e.g., sect. 6 of [23]) the imposed burden is even greater. Designing robust and efficient solution well-tailored to capabilities of constrained parties seems still to be a challenge.

11.2.4. Key generation on user's device

In this model functionalities related to the key generation service are implemented by a user device. It should be noticed that to successfully generate user's exponent du, the user's device need to know the df exponent. The df exponent can be transmitted during key generation process or stored securely on the device production phase.

In this model the user's exponent du and the base key K are not revealed to the finalization service. The disadvantage of this model is that at some stage of the protocol the user's device knows the df exponent and the base key K, which gives it a possibility to create exposes the user to attacks performed by a malicious device producer.

To increase security level of this model the following requirements should be taken into consideration:

- Securely store encryption of exponent df key on user's device before the key K is generated by the device. Decryption key Kdf for this ciphertext should be sent to the device after revealing the public RSA key by the device. In this way even a malicious device cannot adjust the key generated to the given exponent df. Note that one round of communication after generation of the RSA key is always required in this model in order to obtain a certificate. Note also that the decryption key Kdf does not have to be sent over a secure channel provided that the ciphertext of df is placed on the user's device in a secure environment.

- Completely erase df and K from user's device immediately after the key Ku is generated.

It should be noticed that all mentioned controls are based on the trust to manufacturer of the user's device. In particular the user must trust that no kleptographic channel is implemented on the device (cf. [11]).

Note however that if the RSA key is generated directly on a user's

device, then particular attention should be given to quality of randomness. This refers also to signature generation process for non-deterministic signature schemes (cf. RSA-PSS with random salt).

11.2.5. Key generation directly by the finalization service.

This model does not employ a third party generating keys. All steps related to the key generation are performed by the finalization service. In such case a finalization service has all necessary information needed to forge user's signatures.

All threats specific key generation by a separate centralized service (Section 11.2.1. , but since finalization service should be publicly available for signatures finalization, the risk of interception of the finalization service by an adversary becomes significantly bigger. Moreover, because the key generation procedure is cumulated into the server that permanently is contacted by users' devices and might be heavily loaded with new connections, the process controlling quality of keys generated might be somewhat cumbersome. Further, theoretically, one might develop a malicious implementation, which might generate false signatures of some user on a request of an adversary (we suppose that the malicious implementation knows how to recover user's complete RSA key because this implementation has generated the key).

Therefore, the key generation performed directly by the finalization service could be considered only in environments with low security requirements.

11.3. Replacement of a finalization service

Probability of a successful attack on the finalization service's key depends on strength of cryptographic algorithm used, strength of the used key, implementation, security policies executed during system lifetime and other issues. Even in implementations that are supposed to be secure there is a danger of extracting the Fm key by utilizing some new attack, which was not taken into account when the system was designed (cf. e.g., [29],[30],[31]). To minimize consequences of such situation and to additionally facilitate load balancing it is possible to use multiple finalizations services with different Fm keys.

If a few finalization services operate at the same time, with corresponding master keys Fm_1, Fm_2, ..,Fm_t, then if one of the keys (say Fm_2) becomes compromised it is possible to replace it with a new one, and then gradually change users' public keys (immediate change of all public keys in a large system might be infeasible).

Suppose that user's device holds exponents du_1, du_2, ..., du_t referring to the same public key (n,e) or holds some subset of these exponents. Each du_i corresponds to df_i produced from key Fm_i, and the following condition is satisfied:

du_i + df_i = d_i, where d_i is some integer such that d_i*e \equiv 1 mod \lambda(n), that is, d_i \equiv d mod \lambda(n).

If say Fm_2 becomes compromised, exponent du_2 should be erased from the user's device. To introduce a new finalization server with a new master key Fm_2' the following, exemplary procedure might be applied:

Generate df_j (for some j\neq 2) on the server holding Fm_j and split df_j pseudorandomly to the sum of two integers of roughly the same length: df_j = df_j1+df_j2. Next the server holding Fm_j sends df_j2 to the server holding Fm_2'. The latter server generates df_2' from user's Uid and key Fm_2' and subtracts the value generated from the value received: df_j2-df_2'. The resulting value is encrypted with the public encryption key corresponding to Uid (see remarks in Sect. 11.2.2. on preventing the key generation service from learning df). Next the pair (the ciphertext, Uid) is signed by the server holding Fm_2', and this pair and the signature is sent to the server holding Fm_j. The latter server sends df_j1 and the data received from the server holding Fm_2' to the user's device over a secure channel.

The user's device checks the signature, decrypts the ciphertext and adds du_j to the sum of values: df_j1+(df_j2-df_2') = df_j-df_2'.

As a result value d_j - df_2' \equiv d - df_2' mod \lambda(n) is obtained. User's device defines du_2' as d_j - df_2'. When du_2' is calculated all intermediate data should be immediately erased from the user's device.

The procedure above might be initialized at the time of finalization of some signature - this should reduce communication overhead from user's point of view.

Apart from key Fm the finalization server should also hold some authentication key pair and its short-term certificate. Thus in case of detection that Fm could be compromised such an additional security mechanisms should prevent signature finalization outside the legitimate finalization service.

There is some limitation of the technique described above. If Fm_j becomes compromised later, then security of df_2' depends on inaccessibility of df_j - df_2' outside the user's device. If such inaccessibility might be questionable, then we recommend to gradually replace users' public keys.

11.4. Signature creation process security

The signing process consists of two parts:

- Creating a partial signature
- Finalizing a signature

The partial signature creation is performed by a user. The final signature is created by a finalization service. The user key Ku is stored by the user. The key Kf is derived by a finalization service on a basis of the user specific identifier Uid and finalization service's master key Fm.

To create a final signature there is a need to establish a communication channel between the user and the finalization service.

It should be noticed that in the case of the interception of a user's private exponent du it is easy to make the exponent useless. The adversary has to access to the finalization service to create a valid signature. The access to the finalization service can be blocked for a specific exponent. The mediated signature scheme has another advantage: in the case of dispute, the finalization service is able to present the evidence of the signature creation. The finalization service's logs might even have a public form of undeniable timestamps [17] made automatically by the finalization Note that the not-mediated RSA variant usually does not satisfy this property: user's certificate revocation does not prevent the adversary from making signatures with the stolen key. It is possible to achieve comparable level of functionality with timestamped signature with a mandatory signature policy. However, it should be noticed that signature verification applications must be adjusted accordingly.

Since in MRSAA scheme user's device does not know the factorization of the modulus, the Bellcore attack [18] aimed to injecting faults modulo one of the factors of n, does not apply. Even more general attacks [19], [20] on implementation not utilizing knowledge about modulus factorization (i.e., CRT) do not apply directly because nobody knows public exponent eu such that

$$eu * du \equiv 1 \bmod \lambda(n).$$

Even user's device should not know such eu (note that knowledge of both eu and du, provided eu*du < n^2, is polynomially equivalent to the knowledge of factors of n - cf. [21]). Moreover, there is a very efficient probabilistic algorithm that, given eu and du, factors n. The algorithm does not impose constraints on eu*du.

However, in MRSAA scheme the device of the user is not the only component involved in signature generation. Suppose that the channel between user's device and the finalization service is not encrypted, and that the finalization service does not verify the final signature. Let m, sp' be integers representing some encoded message and its faulty partial signature correspondingly. Let sp' be faulty because of a single-bit fault that occurred during this partial signature generation (cf. assumptions in [19], [20], concerning faults generated). Since m and sp' are sent to the finalization service, and

$$m^{df} * sp' \bmod n$$

is returned to the device of the user, it is easy for the adversary to obtain $m^{df} \bmod n$ from the data eavesdropped. If exponentiation method used on the side of the device is the method attacked in [19] or the method attacked in [20], then having m and $m^{df} \bmod n$ the adversary might modify the original attack from [19] or [20] to obtain exponent du.

Hence to prevent an adversary from learning du by utilizing some fault injection attack we recommend to verify the final signature by the finalization service. Verification should include verification of encoding correctness.

The signature creation in MRSAA scheme gives also a possibility to perform the following attack: suppose that the finalization service (e.g., Uid is not included in the certificate of the key (n,e) nor Uid includes hash of (n,e)). Suppose that public keys are not stored by the finalization service's device (e.g., a HSM). Let the adversary might input all necessary data to the finalization service's device and let the device does not verify the finalized signature nor check message encoding correctness. Instead of modulus n the adversary will use modulus n' such that n'> n, let factorization of n' is known to the adversary, and for each prime factor p of n' number p-1 is smooth. In such a case having $m^{df} \bmod n'$ the adversary is able to compute df mod $\lambda(n')$ using Pohling-Hellman method modulo each prime factor of n'. Details of the attack are as follows:

Let the adversary input to the finalization service some tuple (m,sp,n'), where m, sp are some integers.

The finalization service outputs $sp*m^{df} \bmod n'$.

Knowing sp the adversary calculates $m^{df} \bmod n'$ and then

$$df \bmod \lambda(n').$$

If \lambda(n') is too short to recover df completely, the adversary might repeat the procedure for another, appropriately chosen n''. As a result

$$df \bmod \operatorname{lcm}(\lambda(n'),\lambda(n''))$$

is found by the adversary.

The attack above might be modified in such a way that df leaks even if the finalization service verifies the signature (but does not check encoding correctness) - in this case we assume that if signature verification fails finalization service does not return any value modulo n'. The modification proceeds as follows:

modulus n' will again be chosen in a way that the multiplicative group of Z_{n'} will have smooth order and that some other conditions are satisfied, namely: let prime p be divisor of n' and let p' be a prime factor of p-1 such that df is not divisible by p'. Let (p')^t be the greatest power of p' dividing p-1, and let for each prime factor q of n' either (p')^t be the greatest power of p' dividing q-1 or p' do not divide q-1.

Let sp, m belong to the multiplicative group Z_{n'}^{*} of the ring Z_{n'} and are chosen in such a way that:

sp is the neutral element in each cyclic subgroup of the order (p')^t, and in other subgroups of Z_{n'}^{*} it might be any element.

m is the neutral element of each cyclic subgroup of the order not divisible by p', and it is a generator of each subgroup of the order (p')^t. Such m modulo Z_{n'} might be easily found using generators of the multiplicative groups Z_{p}^{*}, Z_{q}^{*} and then utilizing the Chinese Remainder Theorem.

Then the adversary will test some set of exponents e being co-prime with p', such that the set of tested exponents is not greater than the complete set of elements of the multiplicative group Z_{(p')^t}^{*}. Let each tested exponent e fulfill the following condition: for each prime factor q of n' and for each prime factor q' of q-1, if q' is different from p', then e is divisible by the greatest power of q' dividing q-1.

During each test the adversary inputs tuple (m, sp, (n', e)) to the finalization service. If no value modulo n' is returned by the finalization service, then the adversary tries the next exponent e giving not tested yet element of Z_{(p')^t}^{*}. Values m, sp do not have to be changed for the next trial. If df is indeed not divisible by p', then for some e the condition

$$(sp * m^{df})^e \equiv m \bmod n'$$

is satisfied, and the finalization service returns sp*m^{df} mod n'. The adversary knows that for this e the congruence

$$df * e \equiv 1 \bmod (p')^t$$

holds, hence df mod (p')^t is found. If no value is returned and all residues from Z_{(p')^t}^{*} were tested as exponents e, then

$$df \equiv 0 \bmod p'.$$

Executing the attack for different primes p' the adversary finally finds complete exponent df. We have assumed that m does not represent correct message encoding, because finding m, n' fulfilling the conditions above and m representing correct encoding might be computationally hard for appropriate encoding.

If (m, sp, (n',e)) must be delivered over user's device, then security of df depends on security of the authentication key.

Therefore, to provide some security margin we recommend checking during finalization all the following conditions:

- Does m represent correct encoding?
- Do (n,e) and Uid correspond to each other?
- Does Uid correspond the authentication key of the user's device?
- Is the finalized value a correct signature?

Additionally, to protect a signature process the following requirements should be fulfilled:

- du exponent should be stored securely by the user; the exponent shouldn't be revealed to other parties;
- df exponent should be derived from the key Fm in a secure environment by the finalization service; the df exponent should be destroyed immediately after the completion of the signature process;
- the channel between the user's device and the finalization service should be encrypted.

11.5. Decryption process security

The decryption process in MRSAA algorithm consists of the following stages:

- transformation of the ciphertext by the finalization service,
- decryption of the ciphertext by the user.

To perform a decryption it is necessary to transfer a transformed ciphertext to the user's device. The device must check correctness of encoding of the encrypted value.

To protect a decryption process the following requirements should be fulfilled:

- df exponent should be derived in a secure environment by the finalization service; the exponent should be destroyed immediately after the completion of the transformation process;
- (n, e) and df must correspond to the same Uid;
- du exponent should be stored securely by the user; the exponent must not be relayed to other parties.

To provide some security margin we recommend encrypting the channel between the user's device and the finalization service.

11.6. Short summary of possible security techniques

To sum up this section we can briefly enumerate exemplary countermeasures to the threats listed in subsection 11.1. (in the previous subsections we have analyzed need of some of the countermeasures in more detail):

- log service, which works in an append-only way, and which is audited by a third party,
- mutual authentication protocol such like chip authentication and terminal authentication protocols (cf. e.g., [3])resulting in secure and authenticated connection between the finalization service and user's device (in particular, such a connection must be inaccessible for operators of the RSA key generation server).
- internal (nested) signature carried by salt in EMSA-PSS encoding,
- external timestamp and signature applied automatically by the finalization service,

- evolution of unique secrets shared between users and the finalization service as a simple clone-detection mechanism,

- hiding exponent df from the RSA key generation service which can be realized by the finalization service in the following way: the finalization service expresses df as a sum of two integers df_1 + df_2 and then encrypts df_1 to the user, and next sends df_2 instead of df to the key generation service; getting du from the key generation service the user must only subtract df_1 from du to obtain the correct value.

Most of the techniques listed above are described in [13]. The general goal is system decomposition by utilizing existing or introducing additional security layers (cf. chip and terminal authentication procedures), and by assigning to separate parties the task of generating key material used in separate layers.

Moreover, a prudent approach is to require that the parties generating key material have only "local" knowledge on the system that is to require that they are provided with minimum data necessary to accomplish their task. Accordingly, we advise to separate information flow by utilizing encryption.

12. IANA Considerations

13. Conclusions

The main difference between a MRSAA and a RSA algorithm is related to the signing key. In the MRSAA algorithm the key is divided into two fragments. This fact gives a possibility to transfer a final step of signature creation to a finalization service [5].

Incorporating a third party into the signature process allows confirming the signature at the creation time. In such a case the signature which can be successfully verified using a public key can be considered as a confirmed one. The confirmation process can consist of verification of multiple signature attributes (see [6]), particularly a signer's certificate validity check (cf. [12]).

The signature with certificate validity verified on the signature creation stage can be considered as a trusted one without additional verification of certificate validity.

In conclusion, MRSAA protects relying party from rogue signer.

14. References

14.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] E. Barkley, J. Kelsey, "NIST SP800-90 Recommendation for Random Number Generation using Deterministic Random Bit Generators", NIST, March 2007

[3] Bundesamt fur Sicherheit in der Informationstechnik: Technical Guideline TR-03110, Advanced Security Mechanisms for Machine Readable Travel Documents Version 2.05

14.2. Informative References

[4] D. Coppersmith, M. Franklin, J. Patarin and M. Reiter. Low-Exponent RSA with Related Messages. In Advances in Cryptology-Eurocrypt '96, pp. 1-9, Springer-Verlag, 1996

[RFC3447] J. Jonsson, B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1

[5] D. Boneh X. Ding, G. Tsudik and Ch. M. Wong, "A method for Fast Revocation of Public Key Certificates and Security Capabilities", USENIX Association, 2001

[6] M. Tabor, "Electronic signature - easy as card transactions", Polskie Karty 2011 almanach. Medien Service Slawomir Cieslinski,2010

[7] M. Bellare, R. Sandhu, "The Security of Practical Two-Party RSA Signature Schemes", Cryptology ePrint Archive, Report 2001/060

[8] E. Konstantinou, V. Liagkou, P. G. Spirakis, Y. C. Stamatiou, M. Yung, "Trust Engineering: " From Requirements to System Design and Maintenance - A Working National Lottery System Experience", ISC, 2005

[9] E. Konstantinou, V. Liagkou, P. G. Spirakis, Y. C. Stamatiou, M. Yung, "Electronic National Lotteries.", Financial Cryptography, 2004

[10] I. Damgard, G. L. Mikkelsen, "Efficient, Robust and Constant-Round Distributed RSA Key Generation", TCC, 2010

[11] A. Young, M. Young, "A Space Efficient Backdoor in RSA and Its Applications", Selected Areas in Cryptography, 2005

[12] D. Boneh, X. Ding, G. Tsudik, "Fine-grained control of security capabilities", ACM Trans. Internet Techn.,4(1) 2004

[13] P. Blaskiewicz, P. Kubiak, M Kutylowski, "Digital signatures for e-government - a long-term security architecture", China Communications 7(6),December 2010

[14] D. Chaum: Secret-Ballot Receipts: True Voter-Verifiable Elections. IEEE Security & Privacy 2(1): 38-47 (2004)

[15] A. Bauer, J.-S. Coron, D. Naccache, M. Tibouchi, D. Vergnaud, "On the Broadcast and Validity-Checking Security of pkcs#1 v1.5 Encryption", ACNS, 2010

[16] A. K. Lenstra, I. Shparlinski, "Selective Forgery of RSA Signatures with Fixed-Pattern Padding", Public Key Cryptography, 2002

[17] S. Haber, W. S. Stornetta, "How to Time-Stamp a Digital Document", J. Cryptology, 1991

[18] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)", EUROCRYPT, 1997

[19] A. Pellegrini, V. Bertacco, T. M. Austin, "Fault-based attack of RSA authentication", DATE 2010, 2010

[20] D. Boneh, R. A. DeMillo, R. J. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations", J. Cryptology 14, 2001

[21] J. Coron, A. May, "Deterministic Polynomial-Time Equivalence of Computing the RSA Secret Key and Factoring", J. Cryptology 20, 2007

[22] C. Cocks, "Split Knowledge Generation of RSA Parameters", IMA Int. Conf., 1997

[23] M. Joye, R. Pinch, "Cheating in split-knowledge RSA parameter generation", Workshop on Coding and Cryptography, January 1999

[24] N. Gilboa, "Two Party RSA Key Generation", CRYPTO, 1999

[25] J. Algesheimer, J. Camenisch, V. Shoup, "Efficient Computation

Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products", CRYPTO, 2002

[26] E. Ong, J. Kubiatowicz, "Optimizing Robustness While Generating Shared Secret Safe Primes", Public Key Cryptography, 2005

[27] A. Young, M. Yung, "A Space Efficient Backdoor in RSA and Its Applications", Selected Areas in Cryptography, 2005

[28] G. Barbu, H. Thiebeauld, V. Guerin, "Attacks on Java Card 3.0 Combining Fault and Logical Attacks", CARDIS, 2010

[29] O. Aciicmez, C. K. Koc, J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis",Cryptology ePrint Archive: Report 2006/351

[30] D. Bleichenbacher: Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. CRYPTO 1998: 1-12

[31] E. Biham, Y. Carmeli, A. Shamir: Bug Attacks. CRYPTO 2008: 221-240

15. Acknowledgments

This document was prepared using 2-Word-v2.0.template.dot.

Appendix A. Pseudorandom generator primitives

Primitives described below are described in details in [2]. This annex describes only inputs and outputs for these primitives.

A.1. CTR_DRBG_Instantiate_algorithm(entropy_input, personalization_string)

Input:

entropy_input	the string of bits obtained from the source of entropy input
personalization_string	the personalization string received from the consuming application

Output:

initial_working_state	The initial values for V, Key, and reseed_counter

A.2. CTR_DRBG_Generate_algorithm(working_state, requested_number_of_bits, additional_input)

Input:

working_state	The current values for V, Key and reseed_counter
requested_number_of_bits	The number of pseudorandom bits to be returned to the generate function
additional_input	The additional input string received from the consuming application. If additional_input is not supported by an implementation, then step 2 becomes: additional_input = 0^seedlen

Output:

status	The status returned from the function. The status will indicate SUCCESS, or indicate that a reseed is required before the requested pseudorandom bits can be generated.
returned_bits	The pseudorandom bits returned to the generate function
working_state	The new values for V, Key and reseed_counter

Appendix B. Standard RSA primitives

Appendix contains a list of primitives with short description. Detailed specification of primitives can be found in [RFC3447] document.

B.1. Encryption and decryption primitives

RSAEP transforms cleartext into ciphertext. RSADP performs a reverse operation.

B.1.1. 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 or message "message representative out of range"

B.1.2. RSADP(K,c)

Input:

K	RSA private key, where K has one of the following forms - A pair (n,d) - 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 or message "ciphertext representative out of range"

In the present document RSADP was used for key K=Kf and key K=Ku', thus the first form of a private key representation has been utilized.

B.2. Signature and verification primitives

RSASP1 produces signature value from message. RSAVP1 retrieves a message value from signature value.

B.2.1. RSASP1(K,m)

Input:

K	RSA private key, where K has one of the following forms - A pair (n,d) - A quintuple(p,q,dP,dQ,qInv)
c	message representative, an integer between 0 and n-1

Output:

m	signature representative, an integer between 0, and n-1 or message "message representative out of range"

In the present document RSASP1 was used for key K=Ku and key K=Fm. In the first case the first form of a private key representation has been utilized.

B.2.2. RSAVP1((n,e),s)

Input:

(n,e)	RSA public key
s	signature representative, an integer between 0 and n-1

Output:

m	message representative, an integer between 0, and n-1 or message "invalid"

Appendix C. ASN.1 Syntax

C.1. MRSAA key representation

This section defines ASN.1 object identifiers for MRSA public and private keys. It does not define new types, but shows how to use RSA data types with MRSA scheme.

C.1.1. MRSAA public key syntax

The RSA public key syntax is identical to a RSA public key syntax. The RSAPublicKey type should be used to represent MRSAA public key.

C.1.2. MRSAA private key syntax

MRSAA private key should be represented with ASN.1 type RSAPrivateKey according to [RFC3447] section A.1.2:

```
RSAPrivateKey ::= SEQUENCE {
          version           Version,
          modulus           INTEGER,  -- n
          publicExponent    INTEGER,  -- e
          privateExponent   INTEGER,  -- d
          prime1            INTEGER,  -- p
          prime2            INTEGER,  -- q
          exponent1         INTEGER,  -- d mod (p-1)
          exponent2         INTEGER,  -- d mod (q-1)
          coefficient       INTEGER,  -- (inverse of q) mod p
          otherPrimeInfos   OtherPrimeInfos OPTIONAL
      }
```

MRSAA private key, either Ku or Kf, should not contain information which give a possibility to recover d private exponent which corresponds to the public exponent. This implies that the key cannot contain original p and q, d mod (p-1), d mod (q-1), qInv values.

For the purposes of storing finalization service's key Kf or user's key Ku, RSAPrivateKey structure's fields have the following meaning:

- version is the version number, for compatibility with future revisions of this document. It shall be 0 for this version of the document, unless multi-prime is used, in which case it shall be 1. In the case of MRSAA version number 2 should be used

```
Version ::= INTEGER { two-prime(0), multi(1) }
(CONSTRAINED BY
{-- version must be multi if otherPrimeInfos present --})
```

- modulus is the RSA modulus n

- privateExponent is the MRSAA private exponent (df or du respectively)

Other fields should be set to value 0

C.1.3. Scheme identification

The section defines object identifiers for the MRSAA encryption schemes. For signature schemes RSA object identifiers should be used as defined in [RFC3447] section A.2.

Here are the type identifier definitions for the MRSAA OIDs:

mrsa OBJECT-IDENTIFIER ::= { iso(1) identified-organization(3) dod(6) internet(1) private(4) enterprise(1) 37722 applications(1) 1} }

mrsaEncryption OBJECT-IDENTIFIER ::= { applications(1) mrsa(1) algorithms(1) 1}

Id-MRSAES-OAEP OBJECT-IDENTIFIER ::= { applications(1) mrsa(1) algorithms(1) 2}

Appendix D. ASN.1 Module

```
MRSAA  {
 iso(1) identified-organization(3) dod(6) internet(1) private(4)
enterprise(1) 37722 applications(1) 1
}


DEFINITIONS EXPLICIT TAGS ::= BEGIN


EXPORTS ALL;
IMPORTS  ALGORITHM-IDENTIFIER, PKCS1Algorithms
   FROM PKCS-1 {
      iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) 1
   } ;


 MRSAAAlgorithms  ALGORITHM-IDENTIFIER ::= {
 { OID mrsaaEncryption  PARAMETERS NULL } |
 { OID id-MRSAAES-OAEP  PARAMETERS RSA-OAEP-params } |
 PKCS1Algorithms,
 ...
 }


mrsaa OBJECT IDENTIFIER  ::= {
   iso(1) identified-organization(3) dod(6) internet(1) private(4)
enterprise(1) 37722 applications(1) 1
}

--
-- When rsaEncryption is used in an AlgorithmIdentifier the
-- parameters MUST be present and MUST be NULL.
--
mrsaaEncryption OBJECT IDENTIFIER ::= { mrsaa algorithms(1) 1}

--
-- When id-MRSAAES-OAEP is used in an AlgorithmIdentifier the
-- parameters MUST be present and MUST be RSAES-OAEP-params.
--
id-MRSAAES-OAEP OBJECT IDENTIFIER ::= { mrsa algorithms(1) 2}


END
```

Appendix E. Intellectual Property Considerations

The RSA public-key cryptosystem is described in U.S. Patent 4,405,829, which expired on September 20, 2000. RSA Security Inc. For this moment there are no active patents related to the RSA algorithm.

The Mediated RSA cryptographic method and system is described in U.S.patent 20,040,252,830, which will expire on June 2024. The patent describes a mediated RSA cryptosystem with the key fragmented using additive scheme. Additionally the messages passed between a finalization service (Trusted Authority) and message receiver are blinded using random number. However the schema described in this document differs from the one described in the patent, some similarities such like user exponent du can be found.

A U.S. patent 20 050 002 528 describes mediated signature system with identifier based key generation. The concept described in the patent is similar to the one described in this document, but the key generation method is different than presented in this document. Additionally, during a message transmission a random blinding is incorporated.

Authors' Addresses

Miroslaw Kutylowski
Wroclaw University of Technology
Wybrzeze Wyspianskiego 27
PL-50-370 Wroclaw

Email: miroslaw.kutylowski@pwr.wroc.pl

Przemyslaw Kubiak
Wroclaw University of Technology
Wybrzeze Wyspianskiego 27
PL-50-370 Wroclaw
Email: przemyslaw.kubiak@pwr.wroc.pl

Michal Tabor
Trusted Information Consulting
Domaniewska 41A
PL-02-672 Warszawa

Email: michal.tabor@ticons.pl

Daniel Wachnik
Trusted Information Consulting

Domaniewska 41A
PL-02-672 Warszwawa

Email: daniel.wachnik@ticons.pl