Archived NIST Technical Series Publication

The attached publication has been archived (withdrawn), and is provided solely for historical purposes. It may have been superseded by another publication (indicated below).

Archived Publication

Series/Number:	NIST Special Publication 800-56B Rev. 1
Title:	Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography
Publication Date(s):	September 2014
Withdrawal Date:	March 21, 2019
Withdrawal Note:	SP 800-56B Rev. 1 is superseded in its entirety by the publication of SP 800-56B Rev. 2.

Superseding Publication(s)

The attached publication has been **superseded by** the following publication(s):

Series/Number:	NIST Special Publication 800-56B Rev. 2
Title:	Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography
Author(s):	Elaine Barker; Lily Chen; Allen Roginsky; Apostol Vassilev; Richard Davis; Scott Simon
Publication Date(s):	March 2019
URL/DOI:	https://doi.org/10.6028/NIST.SP.800-56Br2

Additional Information (if applicable)

Contact:	Computer Security Division (Information Technology Laboratory)
Latest revision of the attached publication:	
Related information:	https://csrc.nist.gov https://csrc.nist.gov/publications/detail/sp/800-56b/rev-1/archive/2014-10-01
Withdrawal announcement (link):	N/A

National Institute of Standards and Technology
U.S. Department of Commerce

Date updated: March 25, 2019

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

NIST Special Publication 800-56B
Revision 1

Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography

Elaine Barker
Lily Chen
Dustin Moody

http://dx.doi.org/10.6028/NIST.SP.800-56Br1

C O M P U T E R   S E C U R I T Y

National Institute of
Standards and Technology
U.S. Department of Commerce

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

NIST Special Publication 800-56B
Revision 1

Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography

Elaine Barker
Lily Chen
Dustin Moody
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-56Br1

September 2014

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Acting Under Secretary of Commerce for Standards and Technology and Acting Director

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Management Act of 2002 (FISMA), 44 U.S.C. § 3541 *et seq.*, Public Law 107-347. NIST is responsible for developing information security standards and guidelines, including minimum requirements for Federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate Federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), *Securing Agency Information Systems*, as analyzed in Circular A-130, Appendix IV: *Analysis of Key Sections*. Supplemental information is provided in Circular A-130, Appendix III, *Security of Federal Automated Information Resources*.

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

National Institute of Standards and Technology Special Publication 800-56B
Natl. Inst. Stand. Technol. Spec. Publ. 800-56B, 122 pages (September 2014)
http://dx.doi.org/10.6028/NIST.SP.800-56Br1
CODEN: NSPUE2

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-56Br1

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

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

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. All NIST Computer Security Division publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: 56B2014rev-comments@nist.gov

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in Federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

This Recommendation specifies key-establishment schemes using integer factorization cryptography, based on ANS X9.44, Key-establishment using Integer Factorization Cryptography [ANS X9.44], which was developed by the Accredited Standards Committee (ASC) X9, Inc.

Keywords

assurances; integer factorization cryptography; key agreement; key confirmation; key derivation; key-establishment; key management; key recovery; key-transport.

Acknowledgements

This publication was developed in collaboration with the National Security Agency (NSA). The Federal Information Security Management Act of 2002 mandates that NIST consult with the NSA on standards and guidelines for securing information systems. In particular, NIST wishes to acknowledge the substantial contributions made by Rich Davis of the NSA, who served on this publication's development team. He helped to develop the technical content of these guidelines and provided editorial review.

NIST also thanks the many contributions by the public and private sectors whose thoughtful and constructive comments improved the quality and usefulness of this publication. The authors also acknowledge the contributions by Andrew Regenscheid and Miles Smid made to the previous version of this Recommendation.

Conformance Testing

Conformance testing for implementations of this Recommendation will be conducted within the framework of the Cryptographic Algorithm Validation Program (CAVP) and the Cryptographic Module Validation Program (CMVP). The requirements of this Recommendation are indicated by the word "shall." Some of these requirements may be out-of-scope for CAVP or CMVP validation testing, and thus are the responsibility of entities using, implementing, installing or configuring applications that incorporate this Recommendation.

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

Table of Contents

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

This publication is available free of charge from http://dx.doi.org/10.6028/NIST.SP.800-56Br1

Figures

1 Introduction

Many U.S. Government Information Technology (IT) systems need to employ strong cryptographic schemes to protect the integrity and confidentiality of the data that they process. Algorithms such as the Advanced Encryption Standard (AES), as defined in Federal Information Processing Standard (FIPS) 197 [FIPS 197]; Triple DES, as specified in NIST Special Publication (SP) 800-67 [SP 800-67]; and HMAC, as defined in FIPS 198 [FIPS 198] make attractive choices for the provision of these services. These algorithms have been standardized to facilitate interoperability between systems. However, the use of these algorithms requires the establishment of secret keying material that is shared in advance. Trusted couriers may manually distribute this secret keying material, but as the number of entities using a system grows, the work involved in the distribution of the secret keying material grows rapidly. Therefore, it is essential to support the cryptographic algorithms used in modern U.S. Government applications with automated key-establishment schemes.

This Recommendation provides the specifications of key-establishment schemes that are appropriate for use by the U.S. Federal Government, based on a standard developed by the Accredited Standards Committee (ASC) X9, Inc.: ANS X9.44, *Key Establishment using Integer Factorization Cryptography* [ANS X9.44]. A key-establishment scheme can be characterized as either a key-agreement scheme or a key-transport scheme. This Recommendation provides asymmetric-based key-agreement and key-transport schemes that are based on the Rivest Shamir Adleman (RSA) algorithm. When there are differences between this Recommendation and the referenced ANS X9.44 [ANS X9.44] standard, this key-establishment schemes Recommendation **shall** have precedence for U.S. Government applications.

2 Scope and Purpose

This Recommendation is intended for use in conjunction with NIST Special Publication 800-57-Part 1, *Recommendation for Key Management* [SP 800-57]. This key-establishment schemes Recommendation, the Recommendation for Key Management [SP 800-57], and the FIPS 186 [FIPS 186] standard are intended to provide information for a vendor to implement secure key-establishment using asymmetric algorithms in FIPS 140 [FIPS 140] validated modules.

A scheme is a component of a protocol, which in turn may provide security properties not provided by the scheme when considered by itself. Note that protocols, per se, are not specified in this Recommendation.

3 Definitions, Symbols and Abbreviations

3.1 Definitions

Additional input	Information known by two parties that is cryptographically bound to the keying material being protected using the encryption operation.
Algorithm	A clearly specified mathematical process for computation; a set of rules that, if followed, will give a prescribed result.

Approved	Federal Information Processing Standards (FIPS)-**approved** or NIST-**recommended**. An algorithm or technique that meets at least one of the following: 1) is specified in a FIPS or NIST Recommendation, 2) is adopted in a FIPS or NIST Recommendation or 3) is specified in a list of NIST-**approved** security functions (e.g., specified as **approved** in the annexes of FIPS 140 [FIPS 140]).
Assumption	Used to indicate the conditions that are required to be true when an **approved** key-establishment scheme is executed in accordance with this Recommendation.
Assurance of possession of a private key	Confidence that an entity possesses a private key associated with a given public key.
Assurance of validity	Confidence that either a key or a key pair is arithmetically correct.
Binding	Assurance of the integrity of an asserted relationship between items of information that is provided by cryptographic means. Also see Trusted association.
Bit length	The length in bits of a bit string.
Bit string	An ordered sequence of 0's and 1's.
Byte	A bit string of length eight. A byte is represented by a hexadecimal string of length two. The rightmost hexadecimal character represents the rightmost four bits of the byte, and the leftmost hexadecimal character of the byte represents the leftmost four bits of the byte. For example, 9d represents the bit string 10011101.
Byte string	An ordered sequence of bytes.
Certification Authority (CA)	The entity in a Public Key Infrastructure (PKI) that is responsible for issuing public-key certificates and exacting compliance to a PKI policy.
Ciphertext	Data in its enciphered form.
Cryptographic key (Key)	A parameter used with a cryptographic algorithm that determines its operation.
Data integrity	A property whereby data has not been altered in an unauthorized manner since it was created, transmitted or stored. In this Recommendation, the statement that a cryptographic algorithm "provides data integrity" means that the algorithm is used to detect unauthorized alterations.

Decryption	The process of transforming ciphertext into plaintext using a cryptographic algorithm and key.
Destroy	An action applied to a key or a piece of (secret) data. In this Recommendation, after a key or a piece of data is destroyed, no information about its value can be recovered.
Encryption	The process of transforming plaintext into ciphertext using a cryptographic algorithm and key.
Entity	An individual (person), organization, device, or process. "Party" is a synonym.
Entity authentication	A process that establishes the origin of information, or determines an entity's identity to the extent permitted by the entity's identifier.
Fresh	Newly established keying material is considered to be fresh if the probability of being equal to any previously established keying material is acceptably small. The acceptably small probability may be application specific.
Greatest common divisor	The largest positive integer that divides each of two positive integers without a remainder.
Hash function	A function that maps a bit string of arbitrary length to a fixed-length bit string. **Approved** hash functions are expected to satisfy the following properties: **1.** One-way: It is computationally infeasible to find any input that maps to any pre-specified output, and **2.** Collision resistant: It is computationally infeasible to find any two distinct inputs that map to the same output.
Hash value	The fixed-length bit string produced by a hash function.
Identifier	A bit string that is associated with a person, device or organization. It may be an identifying name, or may be something more abstract (for example, a string consisting of an Internet Protocol (IP) address and timestamp).
Key agreement	A (pair-wise) key-establishment procedure where the resultant secret keying material is a function of information contributed by two participants, so that no party can predetermine the value of the secret keying material independently from the contributions of the other party. Contrast with key-transport.

Key-agreement transaction	A key-establishment event which results in secret keying material that is shared between the parties using a key-agreement scheme.
Key confirmation	A procedure to provide assurance to one party (the key-confirmation recipient) that another party (the key-confirmation provider) actually possesses the correct secret keying material and/or shared secret.
Key-confirmation provider	The party that provides assurance to the other party (the recipient) that the two parties have indeed established a shared secret or shared keying material.
Key-derivation function	A function used to derive keying material from a shared secret (or a key) and other information.
Key-derivation method	A method by which keying material is derived from a shared secret and other information. A key-derivation method may use a key-derivation function or a key-derivation procedure.
Key-derivation procedure	A multi-step process that uses an **approved** Message Authentication Code (MAC) algorithm to derive keying material from a shared secret and other information.
Key establishment	The procedure that results in keying material that is shared between the participating parties in a key-establishment transaction.
Key-establishment transaction	An instance of establishing secret keying material using a key-establishment scheme.
Key management	The activities involved in the handling of cryptographic keys and other related security parameters (e.g., initialization vectors (IVs) and passwords) during the entire life cycle of the keys, including their generation, storage, establishment, entry and output, and destruction.
Key pair	A public key and its corresponding private key; a key pair is used with a public-key algorithm.
Key transport	A key-establishment procedure whereby one party (the sender) selects a value for the secret keying material and then securely distributes that value to another party (the receiver). Contrast with key agreement.
Key-transport transaction	A key-establishment event which results in secret keying material that is shared between the parties using a key-transport scheme.
Key wrapping	A method of protecting keying material (along with associated integrity information) that provides both confidentiality and integrity protection when using a symmetric-key algorithm.

Key-wrapping key	In this Recommendation, a key-wrapping key is a symmetric key established during a key-transport transaction and used with a key-wrapping algorithm to protect the keying material to be transported.
Keying material	Data that is represented as a binary string such that any non-overlapping segments of the string with the required lengths can be used as symmetric cryptographic keys. In this Recommendation, keying material is derived from a shared secret established during an execution of a key-agreement scheme, or transported by the sender in a key-transport scheme. As used in this Recommendation, secret keying material may include keys, secret initialization vectors, and other secret parameters.
Least common multiple	The smallest positive integer that is divisible by two positive integers without a remainder. For example, the least common multiple of 2 and 3 is 6.
Length in bits of a non-negative integer *x*	The length, in bits, of the shortest bit string containing the binary representation of *x*. For example, the length in bits of 5 (i.e., 101) is 3. When $x = 0$, its length in bits is 1.
Length in bytes of a non-negative integer, *x*	The length, in bytes, of the shortest byte string containing the binary representation of *x*. For example, the length in bytes of 5 is 1. When $x = 0$, its length in bytes is 1.
MAC tag	Data obtained from the output of a MAC algorithm that can be used by an entity to verify the integrity and the origination of the information used as input to the MAC algorithm.
Message Authentication Code (MAC) algorithm	A family of one-way cryptographic functions that is parameterized by a symmetric key. A given function in the family produces a MacTag on input data of arbitrary length. A MAC algorithm can be used to provide data-origin authentication, as well as data integrity. In this Recommendation, a MAC algorithm is used for key confirmation and key derivation.
Nonce	A time-varying value that has, at most, an acceptably small chance of repeating. For example, a nonce is a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.
Owner	For a key pair, the owner is the entity that is authorized to use the private key associated with a public key, whether that entity generated the key pair itself or a trusted party generated the key pair for the entity.

Party	An individual (person), organization, device, or process. “Entity” is a synonym for party.
Prime number	An integer that is greater than 1 and divisible only by 1 and itself.
Primitive	A low-level cryptographic algorithm used as a basic building block for higher-level cryptographic operations or schemes.
Private key	A cryptographic key that is kept secret and is used with a public-key cryptographic algorithm. A private key is associated with a public key.
Protocol	A special set of rules used by two or more communicating entities that describe the message order and data structures for information exchanged between the entities.
Provider	A party that provides (1) a public key (e.g., in a certificate); (2) assurance, such as an assurance of the validity of a candidate public key or assurance of possession of the private key associated with a public key; or (3) key confirmation. Contrast with recipient.
Public key	A cryptographic key that may be made public and is used with a public-key cryptographic algorithm. A public key is associated with a private key.
Public-key algorithm	A cryptographic algorithm that uses two related keys, a public key and a private key. The two keys have the property that determining the private key from the public key is computationally infeasible.
Public-key certificate	A data structure that contains an entity’s identifier(s), the entity's public key and possibly other information, along with a signature on that data set that is generated by a trusted party, i.e. a certificate authority, thereby binding the public key to the included identifier(s).
Public-key cryptography	A form of cryptography that uses two related keys, a public key and a private key; the two keys have the property that, given the public key, it is computationally infeasible to derive the private key. For key establishment, public-key cryptography allows different parties to communicate securely without having prior access to a secret key that is shared, by using one or more pairs (public key and private key) of cryptographic keys.
Public-key validation	The procedure whereby the recipient of a public key checks that the key conforms to the arithmetic requirements for such a key in order to thwart certain types of attacks.

Receiver	The party that receives secret keying material via a key-transport transaction. Contrast with sender.
Recipient	A party that receives (1) a public key (e.g., in a certificate); (2) assurance, such as an assurance of the validity of a candidate public key or assurance of possession of the private key associated with a public key; or (3) key confirmation. Contrast with provider.
Relatively prime	Two positive integers are relatively prime if their greatest common divisor is 1.
Scheme	A (cryptographic) scheme consists of a set of unambiguously specified transformations that are capable of providing a (cryptographic) service when properly implemented and maintained. A scheme is a higher-level construct than a primitive, and a lower-level construct than a protocol.
Secret keying material (that is shared)	As used in this Recommendation, the secret keying material that is either (1) derived by applying the key-derivation method to the shared secret and other shared information during a key-agreement transaction, or (2) is transported during a key-transport transaction.
Security properties	The security features (e.g., entity authentication, replay protection, or key confirmation) that a cryptographic scheme may, or may not, provide.
Security strength (also, "Bits of security")	A number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system.
Sender	The party that sends secret keying material to the receiver using a key-transport transaction. Contrast with receiver.
Shall	This term is used to indicate a requirement of a Federal Information Processing Standard (FIPS) or a requirement that needs to be fulfilled to claim conformance to this Recommendation. Note that **shall** may be coupled with **not** to become **shall not**.
Shared secret	A secret value that has been computed during a key-establishment scheme, is known by both participants, and is used as input to a key-derivation method to produce keying material.
Should	This term is used to indicate an important recommendation. Ignoring the recommendation could result in undesirable results. Note that **should** may be coupled with **not** to become **should not**.

Symmetric key	A single cryptographic key that is used with a secret (symmetric) key algorithm.
Symmetric-key algorithm	A cryptographic algorithm that uses one secret key that is shared between authorized parties.
Target security strength	The desired security strength for a cryptographic application. The target security strength is selected based upon the amount of security desired for the information protected by the keying material established using this Recommendation.
Trusted association	Assurance of the integrity of an asserted relationship between items of information that may be provided by cryptographic or non-cryptographic (e.g., physical) means. Also see Binding.
Trusted party	A party that is trusted by an entity to faithfully perform certain services for that entity. An entity may choose to act as a trusted party for itself.
Trusted third party	A third party, such as a CA, that is trusted by its clients to perform certain services. (By contrast, for example, the sender and receiver in a scheme are considered to be the first and second parties in a key-establishment transaction).

3.2 Symbols and Abbreviations

A	Additional input that is bound to keying material; a byte string.
$[a, b]$	The set of integers x such that $a \leq x \leq b$.
AES	Advanced Encryption Standard (as specified in FIPS 197 [FIPS 197]).
ANS	American National Standard.
ASC	The Accredited Standards Committee of the American National Standards Institute (ANSI).
ASN.1	Abstract Syntax Notation One.
BS2I	Byte String to Integer conversion routine.
Bytelen	Routine to determine the length in bytes of a string of bytes.
c	Ciphertext; an integer.
C, C_0, C_1	Ciphertext; each is a byte string.

CA	Certification Authority.
CRT	Chinese Remainder Theorem.
d	RSA private exponent; a positive integer.
Data	A variable-length string of zero or more (eight-bit) bytes.
DerivedKeyingMaterial	Derived keying material; a bit string.
dP	RSA private exponent for the prime factor *p* in the CRT format, i.e., *d* mod (*p*-1); an integer.
dQ	RSA private exponent for the prime factor *q* in the CRT format, i.e., *d* mod (*q*-1); an integer.
e	RSA public exponent; a positive integer.
eBits	Length in bits of the RSA exponent *e*.
GCD(*a*, *b*)	Greatest Common Divisor of two positive integers *a* and *b.* For example, GCD(12, 16) = 4.
H	An auxiliary function used in certain key derivation methods. H is either an **approved** hash function, *hash*, or an HMAC-*hash* based on an **approved** hash function, *hash*, with a salt value used as the HMAC key.
HMAC	Keyed-hash Message Authentication Code (as specified in FIPS 198 [FIPS 198]).
HMAC-*hash*	Keyed-hash Message Authentication Code (as specified in [FIPS 198]) with an **approved** hash function *hash*.
I2BS	Integer to Byte String conversion routine.
ID	The bit string denoting the identifier associated with an entity.
ID_P, ID_R, ID_U, ID_V	Identifier bit strings for parties P, R, U, and V, respectively.
IFC	Integer Factorization Cryptography.
k	Keying material; a positive integer.
K	Keying material; a byte string.
KBits	Length in bits of the keying material.

KAS	Key-Agreement Scheme.
KAS1-basic	The basic form of Key-Agreement Scheme 1.
KAS1-Party_V-confirmation	Key-Agreement Scheme 1 with confirmation by party V. Previously known as KAS1-responder-confirmation.
KAS2-basic	The basic form of Key-Agreement Scheme 2.
KAS2-bilateral-confirmation	Key-Agreement Scheme 2 with bilateral confirmation.
KAS2-Party_V-confirmation	Key-Agreement Scheme 2 with confirmation by party V. Previously known as KAS2-responder-confirmation.
KAS2-Party_U-confirmation	Key-Agreement Scheme 2 with confirmation by party U. Previously known as KAS2-initiator-confirmation.
KC	Key Confirmation.
KDF	Key-Derivation Function.
KDM	Key-Derivation Method.
KEM	Key-Encapsulation Mechanism.
KeyData	Keying material other than that which is used for the *MacKey* employed in key confirmation.
KTS	Key-transport Scheme (i.e., KTS-OAEP or KTS-KEM-KWS).
KTS-OAEP-basic	The basic form of the key-transport Scheme with Optimal Asymmetric Encryption Padding.
KTS-OAEP-Party_V-confirmation	Key-transport Scheme with Optimal Asymmetric Encryption Padding and key confirmation provided by party V. Previously known as KTS-OAEP-receiver-confirmation.
KTS-KEM-KWS-basic	The basic form of the key-transport Scheme with the Key Encapsulation Mechanism and Key-Wrapping Scheme.
KTS-KEM-KWS-Party_V-confirmation	Key-transport Scheme using the Key Encapsulation Mechanism and a Key-Wrapping Scheme, and with key confirmation provided by party V.
KWK	Key-Wrapping Key; a byte string.

kwkBits	Length in bits of the key-wrapping key.
KWS	(Symmetric) Key-Wrapping Scheme.
LCM(*a*, *b*)	Least Common Multiple of two positive integers *a* and *b*. For example, LCM(4, 6) = 12.
MAC	Message Authentication Code.
MacData	A byte string input to the *MacTag* computation.
$MacData_U$, (or $MacData_V$)	*MacData* associated with party U (or party V, respectively), and used to generate $MacTag_U$ (or $MacTag_V$, respectively). Each is a byte string.
MacKey	Key used to compute the MAC; a byte string.
MacKeyLen	Length in bytes of the *MacKey*.
MacTag	A byte string that allows an entity to verify the integrity of the information. *MacTag* is the output from the MAC algorithm (possibly after truncation). The literature sometimes refers to *MacTag* as a Message Authentication Code (MAC).
$MacTag_V$, ($MacTag_U$)	The *MacTag* generated by party V (or party U, respectively). Each is a byte string.
MacTagLen	The length of *MacTag* in bytes.
Mask	Mask; a byte string.
MGF	Mask Generation Function.
mgfSeed	String from which a mask is derived; a byte string.
n	RSA modulus. $n = pq$, where *p* and *q* are distinct odd primes.
(*n*, *d*)	RSA private key in the basic format.
(*n*, *e*)	RSA public key.
(*n, e, d, p, q, dP, dQ, qInv*)	RSA private key in the Chinese Remainder Theorem (CRT) format.
N_V	Nonce contributed by party V; a byte string.
nBits	Length in bits of the RSA modulus *n*.

nLen	Length in bytes of the RSA modulus *n*.
Null	The empty bit string.
OtherInfo	Other information for key derivation; a bit string.
p	First prime factor of the RSA modulus *n*.
(*p*, *q*, *d*)	RSA private key in the prime-factor format.
$PrivKey_U$, $PrivKey_V$	Private key of party U or V, respectively.
$PubKey_U$, $PubKey_V$	Public key of party U or V, respectively.
q	Second prime factor of the RSA modulus *n*.
qInv	Inverse of *q* modulo *p* in the CRT format, i.e., q^{-1} mod *p*; an integer.
RBG	Random Bit Generator.
RSA	Rivest-Shamir-Adleman algorithm
RSASVE	RSA Secret Value Encapsulation.
RSA-KEM-KWS	RSA Key Encapsulation Mechanism with a Key-Wrapping Scheme.
RSA-OAEP	RSA with Optimal Asymmetric Encryption Padding.
S	String of bytes.
s	Security strength in bits.
SHA	Secure Hash Algorithm.
$T_{MacTagBits}(X)$	A truncation function that outputs the most significant (i.e., leftmost) *MacTagBits* bits of the input string, *X*, when the bit length of *X* is greater than *MacTagBits*; otherwise, the function outputs *X*. For example, $T_2(1011)=10$, $T_3(1011)=101$, and $T_4(1011)=1011$.
TransportedKeyingMaterial	Transported keying material.
TTP	A Trusted Third Party.
U	One party in a key-establishment scheme.

V	Another party in a key-establishment scheme.
X	Byte string to be converted to or from an integer; the output of conversion from an ASCII string.
$X =? Y$	Check for equality of X and Y.
x	Non-negative integer to be converted to or from a byte string.
$x \bmod n$	The modular reduction of the (arbitrary) integer x by the positive integer n (the *modulus*). For the purposes of this Recommendation, $y = x \bmod n$ is the unique integer satisfying the following two conditions: 1) $0 \le y < n$, and 2) $x - y$ is divisible by n.
$x^{-1} \bmod n$	The multiplicative inverse of the integer x modulo the positive integer n. This quantity is defined if and only if x is relatively prime to n. For the purposes of this Recommendation, $y = x^{-1} \bmod n$ is the unique integer satisfying the following two conditions: 1) $0 \le y < n$, and 2) $1 = (xy) \bmod n$.
$\{X\}$	Indicates that the inclusion of X is optional.
$\{x, y\}$	A set containing the integers x and y.
$X \parallel Y$	Concatenation of two strings X and Y.
$\lceil x \rceil$	The ceiling of x; the smallest integer $\ge x$. For example, $\lceil 5 \rceil = 5$ and $\lceil 5.3 \rceil = 6$.
$\lvert x \rvert$	The absolute value of x.
Z	A shared secret that is used to derive secret keying material using a key-derivation method; a byte string.
z	The integer form of Z.
$\lambda(n)$	Lambda function of the RSA modulus n, i.e., the least positive integer i such that $1= a^i \bmod n$ for all a relatively prime to n. When $n=pq$, $\lambda(n) = \mathrm{LCM}(p-1, q-1)$.
$\oplus$	Exclusive-Or (XOR) operator, defined as bit-wise modulo 2 arithmetic with no carry.

4 Key-Establishment Schemes Overview

Secret cryptographic keying material may be electronically established between parties by using a key-establishment scheme, that is, by using either a key-agreement scheme or a key-transport scheme. Key-establishment schemes may use either symmetric-key techniques or asymmetric-key techniques or both. The key-establishment schemes described in this Recommendation use asymmetric-key techniques.

In this Recommendation, the **approved** key-establishment schemes are described in terms of the roles played by parties "U" and "V." These are specific labels that are used to distinguish between the two participants engaged in key establishment – irrespective of the actual labels that may be used by a protocol employing a particular **approved** key-establishment scheme.

During key agreement, the derived secret keying material is the result of contributions made by both parties. To be in conformance with this Recommendation, a protocol employing any of the **approved** pair-wise key-agreement schemes **shall** unambiguously assign the roles of U and V to the participants by clearly defining which participant performs the actions ascribed by this Recommendation to party U, and which performs the actions ascribed herein to party V.

During key transport, one party selects the secret keying material to be transported. The secret keying material is then encrypted/wrapped, and sent to the other party. The party that sends the secret keying material is called the sender, and the other party is called the receiver. In this Recommendation, key-wrapping is performed in the same transaction that establishes the key-wrapping key using an **approved** key-transport scheme, and party U is always assigned the role of the sender, while party V is the receiver.

The security of the Integer Factorization Cryptography (IFC) schemes in this Recommendation relies on the intractability of factoring integers that are products of two sufficiently large, distinct prime numbers. All IFC schemes in this Recommendation are based on RSA.

The security of an IFC scheme also depends on its implementation, and this document includes a number of practical recommendations for implementers. For example, good security practice dictates that implementations of procedures employed by primitives, operations, schemes, etc., include steps that destroy any potentially sensitive locally stored data that is created (and/or copied for use) during the execution of a particular procedure, and whose continued local storage is not required after the procedure has been exited. The destruction of such locally stored data ideally occurs prior to or during any exit from the procedure. This is intended to limit opportunities for unauthorized access to sensitive information that might compromise a key-establishment process.

Explicit instructions for the destruction of certain potentially sensitive values that are likely to be locally stored by procedures are included in the specifications found in this Recommendation. Examples of such values include local copies of any portions of secret or private keys that are employed or generated during the execution of a procedure, intermediate results produced during computations, and locally stored duplicates of values that are ultimately output by a procedure. However, it is not possible to anticipate the form of all possible implementations of the specified primitives, operations, schemes, etc., making it impossible to enumerate all potentially sensitive data that might be locally stored by a procedure employed in a particular implementation. Nevertheless, the destruction of any potentially sensitive locally stored data is an obligation of all implementations.

Error handling can also be an issue. Section 7 cautions implementers to handle error messages in a manner that avoids revealing even partial information about the decryption/decoding processes that may be performed during the execution of a particular procedure.

For compliance with this Recommendation, equivalent processes may be used. Two processes are equivalent if, whenever the same values are input to each process (either as input parameters or as values made available during the process), each process produces the same output as the other.

Some processes are used to provide assurance (for example, assurance of the arithmetic validity of a public key or assurance of possession of a private key associated with a public key). The party that provides the assurance is called the provider (of the assurance), and the other party is called the recipient (of the assurance).

A number of steps are performed to establish secret keying material as described in Sections 4.1, 4.2, and 4.3.

4.1 Key-Establishment Preparations

The owner of a private/public key pair is the entity that is authorized to use the private key of that key pair. Figure 1 depicts the steps that may be required of that entity when preparing for a key-establishment process (i.e., either key agreement or key transport).

The first step in the preparation is for the entity to obtain a key pair. Either the entity (i.e., the owner) generates the key pair as specified in Section 6.3, or a trusted third party (TTP) generates the key pair as specified in Section 6.3 and provides it to the owner. The owner obtains assurance of key-pair validity and, as part of the process, obtains assurance that it actually possesses the (correct) private key. **Approved** methods for obtaining assurance of key-pair validity by the owner are provided in Section 6.4.1.

An identifier is used to label the entity that owns a key pair used in a key-establishment transaction. This label may uniquely distinguish the entity from all others, in which case it could rightfully be considered an identity. However, the label may be something less specific – an organization, nickname, etc. – hence, the term *identifier* is used in this Recommendation, rather than the term *identity*. For example, an identifier could be "`Vegetable.gardener123`," rather than an identifier that names a particular person. A key pair's owner (or an agent trusted to act on the owner's behalf) is responsible for ensuring that the identifier associated with its public key is appropriate for the applications in which the public key will be used.

For each key pair employed during a key-establishment transaction, this Recommendation assumes that there is a trusted association between the owner's identifier(s) and the owner's public key. The association may be provided using cryptographic mechanisms or by physical means. The use of cryptographic mechanisms may require the use of a binding authority (i.e., a trusted authority) that binds the information in a manner that can be verified by others; an example of such a trusted authority is a registration authority working with a CA who creates a certificate containing both the public key and the identifier(s). The binding authority **shall** verify the owner's intent to associate the public key with the specific identifier(s) chosen for the owner; the means for accomplishing this is beyond the scope of this Recommendation. The binding authority **shall** obtain assurance of both the arithmetic validity of the owner's public key and the owner's possession of the private key corresponding to that public key. (**Approved** techniques that can be employed by the binding authority to obtain these assurances are described in Section 6.4.2.1 [method 1], Section 6.4.2.2, Section 6.4.2.3 and Section 6.4.2.3.2.)

As an alternative to reliance upon a binding authority, trusted associations between identifiers and public keys may be established by the direct exchange of this information between entities, using a mutually trusted method (e.g., a trusted courier or a face-to-face exchange). In this case, each entity receiving a public key and associated identifier(s) **shall** be responsible for obtaining the same assurances that would have been obtained on the entity's behalf by a binding authority (see the previous paragraph). Entities **shall** also be responsible for maintaining (by cryptographic or other means) the trusted associations between any identifiers and public keys received through such exchanges.

If an entity engaged in a key-establishment transaction owns a key pair that is employed during the transaction, then the identifier used to label that party **shall** be one that has a trusted association with the public key of that key pair. If an entity engaged in a key-establishment transaction does not employ a key pair during the transaction, but an identifier is still desired/required for that party, then a non-null identifier **shall** be selected/assigned in accordance with the requirements of the protocol relying upon the transaction.

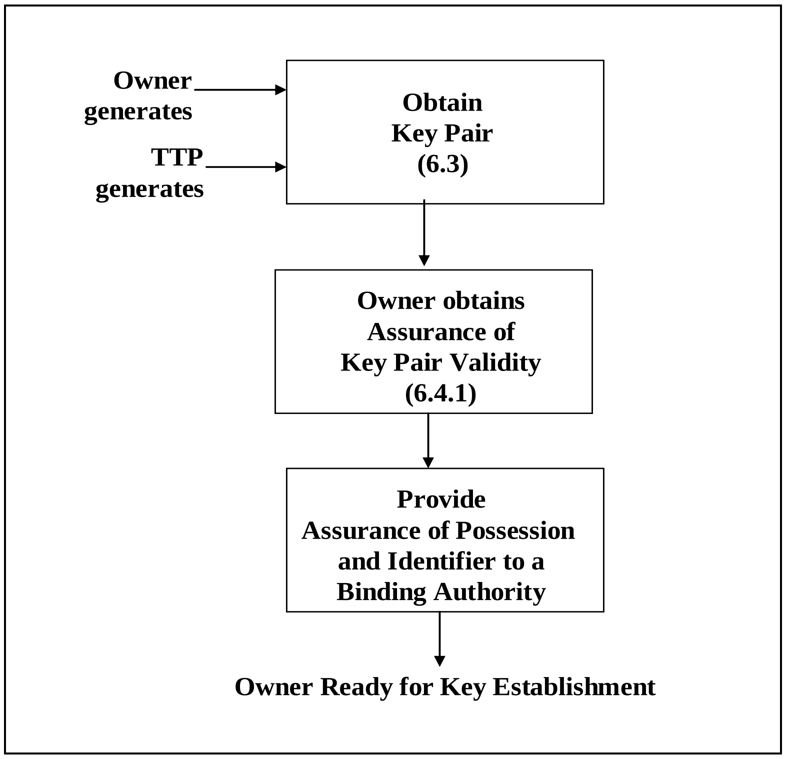
Owner generates
TTP generates
Obtain Key Pair (6.3)
Owner obtains Assurance of Key Pair Validity (6.4.1)
Provide Assurance of Possession and Identifier to a Binding Authority
Owner Ready for Key Establishment

Figure 1: Owner Key-establishment Preparations

After the above steps have been performed, the key-pair owner is ready to enter into a key-establishment process.

4.2 Key-Agreement Process

Figure 2 depicts the steps implemented by an entity when establishing secret keying material with another entity using one of the key-agreement schemes described in this Recommendation. (Some discrepancies in ordering may occur in practice, depending on the communication protocol in which the key-agreement process is performed.) Note that some of the actions shown may not be a part of every scheme. For example, key confirmation is not provided in the basic key-agreement schemes (see Sections 8.2.2 and 8.3.2). The specifications of this Recommendation indicate when a particular action is required.

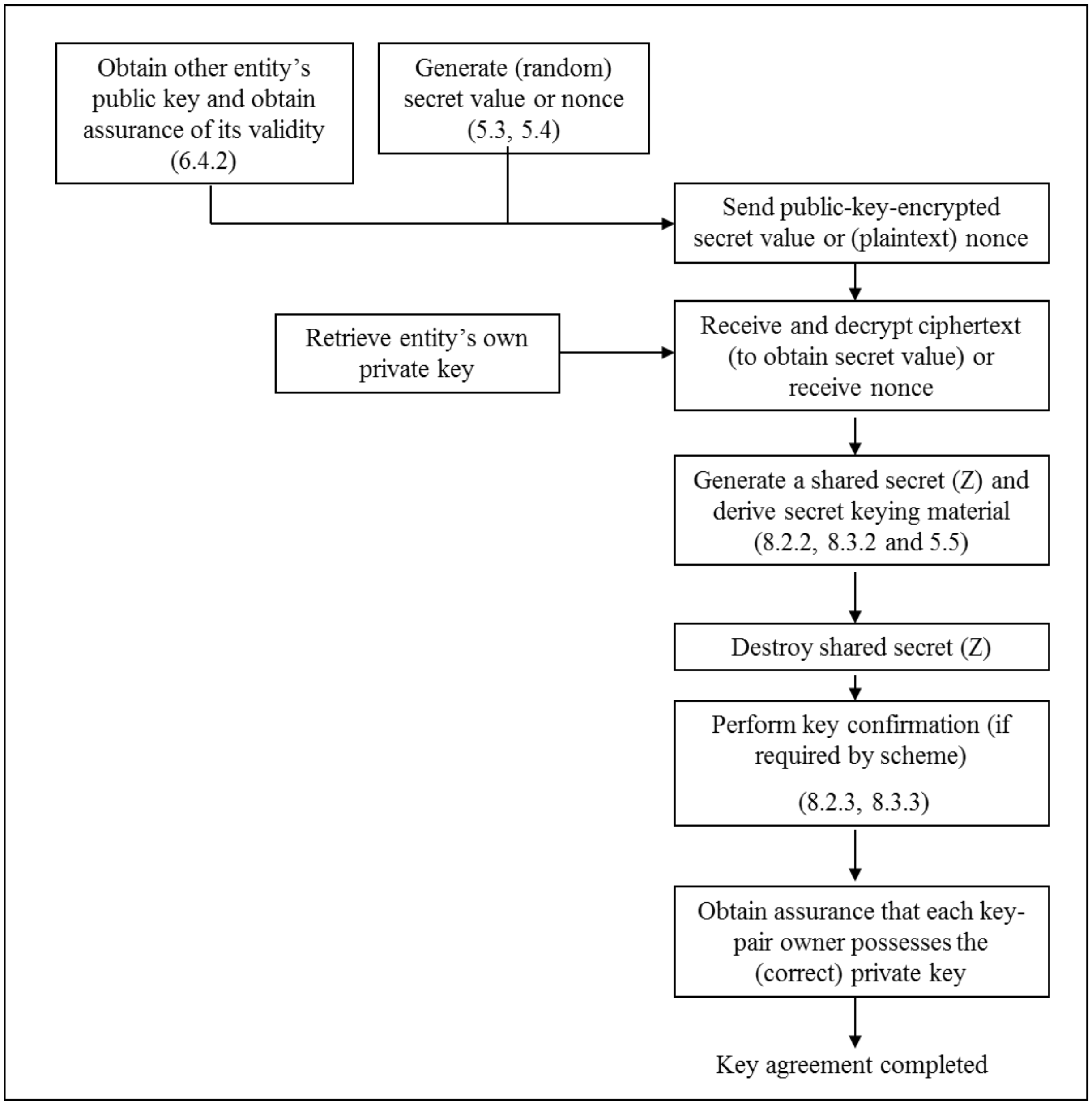
Obtain other entity's public key and obtain assurance of its validity (6.4.2)
Generate (random) secret value or nonce (5.3, 5.4)
Send public-key-encrypted secret value or (plaintext) nonce
Retrieve entity's own private key
Receive and decrypt ciphertext (to obtain secret value) or receive nonce
Generate a shared secret (Z) and derive secret keying material (8.2.2, 8.3.2 and 5.5)
Destroy shared secret (Z)
Perform key confirmation (if required by scheme) (8.2.3, 8.3.3)
Obtain assurance that each key-pair owner possesses the (correct) private key
Key agreement completed

Figure 2: Key-Agreement Process

Each participant obtains the identifier associated with the other entity, and verifies that the identifier of the other entity corresponds to the entity with whom the participant wishes to establish secret keying material.

Each entity that requires the other entity's public key for use in the key-agreement scheme obtains a public key that has a trusted association with the other party's identifier, and obtains assurance of the validity of the public key. **Approved** methods for obtaining assurance of the validity of another entity's public key are provided in Section 6.4.2.

Each entity generates either a (random) secret value or a nonce, as required by the particular key-agreement scheme. If the scheme requires an entity to generate a secret value, that secret value is generated as specified in Section 5.3 and encrypted using the other entity's public key. The resulting ciphertext is then provided to the other entity. If the key-agreement scheme requires that an entity provide a nonce, that nonce is generated as specified in Section 5.4 and provided (in plaintext form) to the other party. (See Sections 8.2 and 8.3 for details).

Each participant in the key-agreement process uses the appropriate public and/or private keys to establish a shared secret (*Z*) as specified in Section 8.2.2 or 8.3.2. Each participant then derives secret keying material from the shared secret (and other information), as specified in Section 5.5.

If the key-agreement scheme includes key confirmation provided by one or both of the participants, then key confirmation is performed as specified in Section 8.2.3 or 8.3.3. When performed in accordance with those sections, successful key confirmation may also provide assurance that a key-pair owner possesses the (correct) private key (see Section 6.4.2.3.2).

The owner of any key pair used during the key-agreement transaction is required to have assurance that the owner is in possession of the correct private key. Likewise, the recipient of another entity's public key is required to have assurance that its owner is in possession of the corresponding private key. Assurance of private-key possession is obtained prior to using the derived keying material for purposes beyond those of the key-agreement transaction itself. This assurance may be provided/obtained either through key confirmation, or by some other **approved** means (see Sections 6.4.1 and 6.4.2).

4.3 Key-Transport Process

Figure 3 depicts the steps implemented by two entities when using one of the key-transport schemes described in this Recommendation to establish secret keying material.

The entity who will act as the sender obtains the identifier associated with the entity that will act as the receiver, and verifies that the receiver's identifier corresponds to an entity to whom the sender wishes to send secret keying material.

Prior to performing key transport, the sender obtains the receiver's public key and obtains assurance of its validity. **Approved** methods for obtaining assurance of the validity of another entity's public key are provided in Section 6.4.2. The sender is also required to have assurance that the receiver is in possession of the private key corresponding to the receiver's public key prior to key transport, unless that assurance is obtained via key confirmation included as part of the scheme. (See Sections 9.2 and 9.3 for details).

The sender selects the secret keying material (and, perhaps, additional input) to be transported to the other entity. Then, using the intended receiver's public key, the sender either encrypts that material directly (as specified in Section 9.2.3) or employs a combination of secret value encapsulation and key wrapping (as specified in Section 9.3.3). The resulting ciphertext is transported to the receiver.

Prior to participating in a key-establishment transaction, the receiver is required to have assurance of the validity of its own key pair. This assurance may be renewed whenever desired. Upon (or before) receipt of the transported ciphertext, the receiver retrieves the private key from its own key pair. Using its private key, the receiver takes the necessary steps (as specified in Section 9.2.3 or 9.3.3) to decrypt the ciphertext and obtain the transported plaintext keying material.

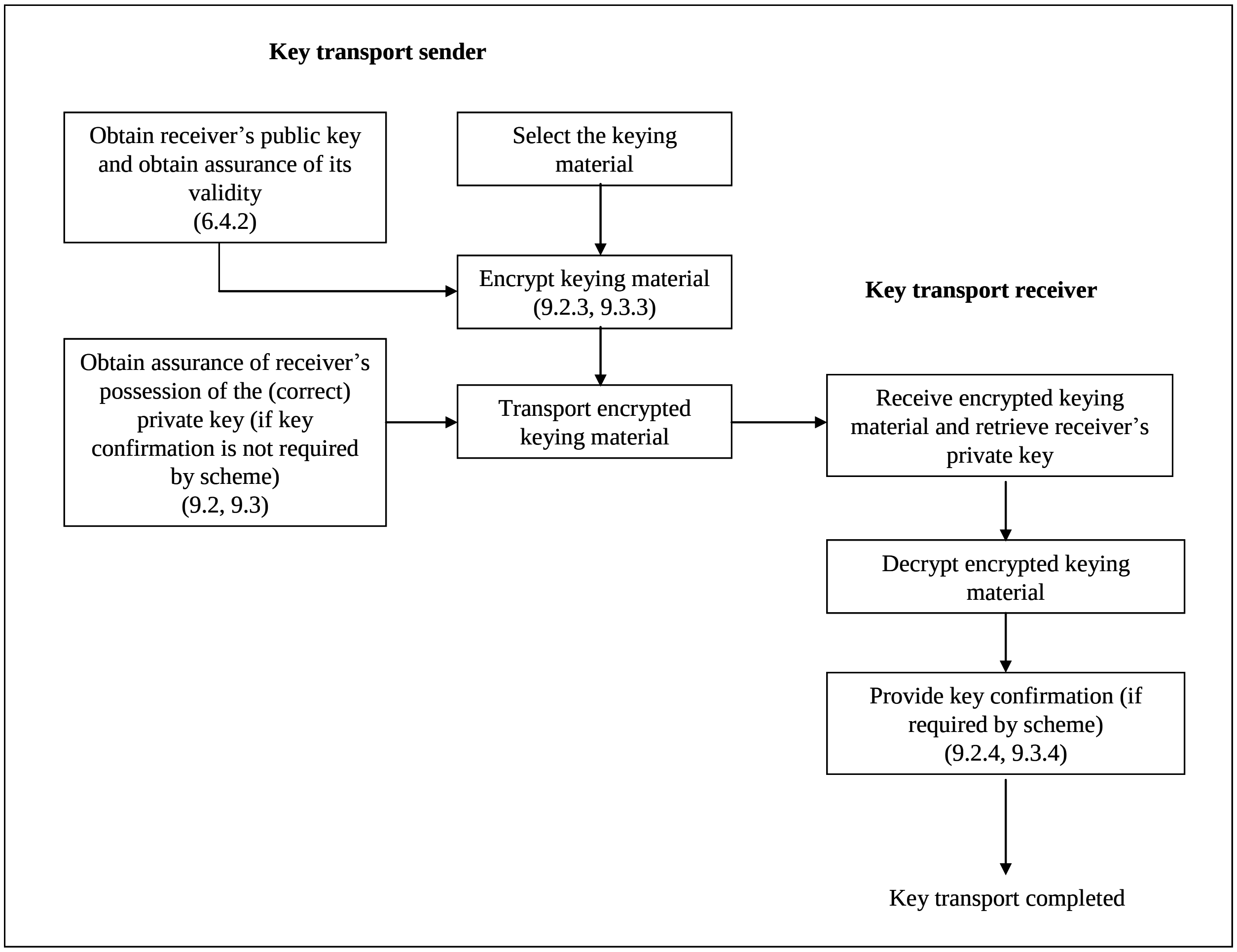
Key transport sender
Obtain receiver's public key and obtain assurance of its validity (6.4.2)
Select the keying material
Encrypt keying material (9.2.3, 9.3.3)
Obtain assurance of receiver's possession of the (correct) private key (if key confirmation is not required by scheme) (9.2, 9.3)
Transport encrypted keying material
Key transport receiver
Receive encrypted keying material and retrieve receiver's private key
Decrypt encrypted keying material
Provide key confirmation (if required by scheme) (9.2.4, 9.3.4)
Key transport completed

Figure 3: Key-transport Process

If the key-transport scheme includes key confirmation, then key confirmation is provided by the receiver to the sender as specified in Section 9.2.4 or 9.3.4. Through the use of key confirmation, the sender can obtain assurance that the receiver has correctly recovered the keying material from the ciphertext. Successful key confirmation may also provide assurance that the receiver was in possession of the correct private key (see Section 6.4.2.3.2).

5 Cryptographic Elements

This section describes the basic cryptographic elements that support the development of key-establishment schemes specified in this Recommendation.

5.1 Cryptographic Hash Functions

In this Recommendation, cryptographic hash functions may be used in mask generation during RSA-OAEP encryption/decryption, in key derivation, and/or in MAC-tag computation during key confirmation. An **approved** hash function **shall** be used when a hash function is required (see [FIPS 180] and [FIPS 202]).

5.2 Message Authentication Code (MAC) Algorithms

A Message Authentication Code (MAC) algorithm defines a family of one-way (MAC) functions that is parameterized by a symmetric key. The input to a MAC function includes a symmetric key, called *MacKey*, and a binary data string called *MacData*. That is, a MAC function is represented as MAC(*MacKey*, *MacData*). In this Recommendation, a MAC function is used in key confirmation and may be used for key derivation.

Approved MAC algorithms are specified in [FIPS 198] (i.e., HMAC) and [SP 800-38B] (i.e., CMAC). HMAC requires the use of an **approved** hash function; CMAC requires the use of an **approved** block cipher algorithm.

Key derivation may be performed as either a single- or multiple-step process. When used for single-step key derivation, either an **approved** hash function, or HMAC with an **approved** hash function **shall** be selected as the function H, in accordance with Section 5.5.1. The appropriate use of HMAC and/or CMAC in an extraction-then-expansion key-derivation procedure (a multiple-step process) is specified in [SP 800-56C]. Additional **approved** application-specific uses of MAC algorithms for key-derivation purposes are provided in [SP 800-135].

When used for key confirmation, the key-confirmation provider is required to compute a MAC tag on received or derived data using the agreed-upon MAC function. A symmetric key derived from a shared secret (during a key-agreement transaction) or extracted from transported keying material (during a key-transport transaction) is used as *MacKey*. The resulting MAC tag is sent to the key-confirmation recipient, who can obtain assurance (via MAC-tag verification) that the shared secret and derived keying material were correctly computed (in the case of key agreement) or that the transported keying material was successfully received (in the case of key transport). MAC-tag computation and verification are defined in Sections 5.2.1 and 5.2.2.

5.2.1 *MacTag* Computation for Key Confirmation

The computation of the MAC tag is represented as follows:

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

To compute a MAC tag:

1. An **approved**, agreed-upon MAC algorithm (see [FIPS 198] or [SP 800-38B]) is used with a *MacKey* to compute a MAC on the *MacData*, where *MacKey* is a symmetric key, and

MacData represents the data on which the MAC tag is computed. The minimum length of *MacKey* is specified in Section 5.6.3.

MacKey is obtained from the *DerivedKeyingMaterial* (when a key-agreement scheme employs key confirmation) or obtained from the *TransportedKeyingMaterial* (when a key-transport scheme employs key confirmation), as specified in Section 5.6.1.1.

The resulting MAC consists of *MacBitLen* bits, which is the full output length of the selected MAC algorithm.

2. The *MacBitLen* bits are input to a truncation function $T_{MacTagBits}$ to obtain the most significant (i.e., leftmost) *MacTagBits* bits, where *MacTagBits* represents the intended length of the *MacTag*, which is required to be less than or equal to *MacBitLen*. (When *MacTagBits* equals *MacBitLen*, $T_{MacTagBits}$ acts as the identity function.) The minimum value for *MacTagBits* is specified in Section 5.6.3.

Note: A routine implementing a MacTag computation for key confirmation **shall** destroy any local copies of *MacKey* and *MacData*, any locally stored portions of *MacTag*, and any other locally stored values used or produced during the execution of the routine; their destruction **shall** occur prior to or during any exit from the routine – whether exiting early because of an error or exiting normally, with the output of *MacTag*.

5.2.2 *MacTag* Verification for Key Confirmation

To verify the MAC tag received during key confirmation, a new MAC tag, *MacTag′*, is computed as specified in Section 5.2.1 using the values of *MacKey*, *MacTagBits*, and *MacData* possessed by the key-confirmation recipient. *MacTag′* is compared with the received MAC tag (i.e., *MacTag*). If their values are equal, then it may be inferred that the same *MacKey*, *MacTagBits*, and *MacData* values were used in the computation of *MacTag* and *MacTag′*. That is, the key-confirmation provider has obtained the same keying material as the key-confirmation recipient.

5.3 Random Bit Generators

Whenever this Recommendation requires the use of a randomly generated value (for example, for obtaining keys or nonces), the values **shall** be generated using an **approved** random bit generator (RBG), as specified in [SP 800-90], that provides an appropriate security strength.

When an **approved** RBG is used to generate a secret value as part of a key-establishment scheme specified in this Recommendation (i.e., Z in a scheme from the KAS1 or KTS-KEM-KWS families, or Z_U and Z_V in a scheme from the KAS2 family), that RBG **shall** be instantiated to support a security strength that is equal to or greater than the security strength associated with the RSA modulus length as specified in [SP 800-57, Part 1]. See [SP 800-90] for details.

5.4 Nonces

A nonce is a time-varying value that has (at most) a negligible chance of repeating. This Recommendation requires party V to supply a nonce, N_V, during the execution of key-agreement schemes in the KAS1 family (see Section 8.2). This nonce is included in the input to the key-derivation process, and (when key confirmation is employed) is also used in the computation of the MAC tag sent from party V to party U.

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

1. A random bit string that is generated anew for each nonce, using an **approved** random bit generator. A nonce containing a component of this type is called a *random nonce*.
2. A timestamp of sufficient resolution (detail) so that it is different each time it is used.
3. A monotonically increasing sequence number, or
4. A combination of a timestamp and a monotonically increasing sequence number, such that the sequence number is reset when and only when the timestamp changes. (For example, a timestamp may show the date but not the time of day, so a sequence number is appended that will not repeat during a particular day.)

For the KAS1 schemes, the required nonce N_V **should** be a random nonce, including a component that consists of a random bit string that is generated anew for each transaction using an **approved** random bit generator (RBG). For conformance with this Recommendation, the security strength (in bits) supported by the instantiation of this RBG **shall** be at least 112 bits for use with a 2048-bit RSA modulus, and **shall** be at least 128 bits for use with a 3072-bit RSA modulus. For use with a 2048-bit RSA modulus, the length in bits of the random bit string incorporated into the random nonce **shall** be at least 112 bits, but **should** be at least 224 bits. For use with a 3072-bit RSA modulus, the length in bits of the random bit string incorporated into the random nonce **shall** be at least 128 bits, but **should** be at least 256 bits. For details concerning the security strength supported by an instantiation of a random bit generator, see [SP 800-90].

As part of the proper implementation of this Recommendation, system users and/or agents trusted to act on their behalf **should** determine that the components selected for inclusion in required nonces meet the security requirements of those users or agents. The application tasked with performing key establishment on behalf of a party **should** determine whether or not to proceed with a key-establishment transaction, based upon the perceived adequacy of the method(s) used to form the required nonces. Such knowledge may be explicitly provided to the application in some manner, or may be implicitly provided by the operation of the application itself.

5.5 Key-Derivation Methods

This section introduces **approved** key-derivation methods for use in key establishment as specified in this Recommendation. An **approved** key-derivation method **shall** be used to derive keying material from the shared secret Z during the execution of a key-establishment scheme from the KAS1, KAS2, or KTS-KEM-KWS family of schemes.

Key-derivation methods that conform to this Recommendation include the use of an **approved** single-step key-derivation function (KDF), as well as the use of an **approved** two-step (extraction-then-expansion) key-derivation procedure (for more details, see Sections 5.5.1 and 5.5.2, respectively). Certain **approved** application-specific key-derivation methods may be used as well (see Section 5.5.3). Other key-derivation methods may be temporarily allowed for backward compatibility; these other allowable methods – and any restrictions on their use – will be specified in [FIPS 140 IG].

When employed during the execution of a key-establishment scheme as specified in this Recommendation, the agreed-upon key-derivation method uses input that includes a freshly created shared secret *Z* along with other information. The derived keying material **shall** be computed in its entirety before outputting any portion of it, and (all copies of) *Z* **shall** be destroyed immediately following its use. The output produced by the key-derivation method **shall** only be used as secret keying material – such as a symmetric key used for key wrapping, data encryption, or message integrity; a secret initialization vector; or, perhaps, a master (key-derivation) key that will be used to generate additional keying material (possibly using a different process) (see [SP 800-108]). Non-secret keying material (such as a non-secret initialization vector) **shall not** be generated from input that includes the shared secret *Z*.

5.5.1 The Single-step Key-Derivation Function

This section specifies an **approved** key-derivation function (KDF) that is executed in a single step, rather than the two-step procedure discussed in Section 5.5.2. The input to the KDF includes the shared secret *Z* (represented as a byte string).

This single-step KDF uses an auxiliary function H, which can be either 1) an **approved** hash function (see Section 5.1), denoted by *hash*, or 2) an HMAC based on an **approved** hash function (as specified in [FIPS 198]), denoted by HMAC-*hash*. When the single-step key-derivation function is employed by a key-establishment scheme specified in this Recommendation, any **approved** hash function, *hash*, can be used to define the auxiliary function, H, whether H = *hash* or H = HMAC-*hash* (see Section 5.5.1.1).

5.5.1.1 The Single-step KDF Specification

This section specifies an **approved** single-step key-derivation function (KDF) whose input includes the shared secret *Z* (represented as a byte string) and other information.

The KDF is specified as follows:

Function call: kdf (*Z, OtherInput*),

where *OtherInput* consists of *KBits* and *OtherInfo*.

Auxiliary Function H (two options):

Option 1: H(*x*) = *hash*(*x*), where *hash* is an **approved** hash function (see Section 5.1); the input, *x*, is a bit string.

Option 2: H(*x*) = HMAC-*hash*(*salt*, *x*), where HMAC-*hash* is an instantiation of the HMAC function (as defined in [FIPS 198]) employing an **approved** hash function, *hash* (see Section 5.1). An implementation-dependent byte string, *salt*, serves as the HMAC key, and *x* (the input to H) is a bit string that serves as the HMAC "message" – as specified in [FIPS 198].

Implementation-Dependent Parameters:

1. *hBits*: an integer that indicates the length (in bits) of the output block of the hash function, *hash*, employed by the auxiliary function, H, that is used to derive blocks of secret keying material.
2. *max_H_inputBits*: an integer that indicates the maximum-permitted length (in bits) of the bit string, *x*, used as input to the auxiliary function, H.

3. *salt*: a (secret or non-secret) byte string that is only required when an HMAC-based auxiliary function is implemented (see Option 2 above). The *salt* could be, for example, a value computed from nonces exchanged as part of a key-establishment protocol that employs one or more of the key-agreement schemes specified in this Recommendation, a value already shared by the protocol participants, or a value that is pre-determined by the protocol. The length of the *salt* can be any agreed-upon length. However, if there is no means of selecting the *salt*, then it **shall** be an all-zero byte string whose bit length equals the length of the input block for the hash function, *hash*.

Input:

1. *Z*: a byte string that represents the shared secret.
2. *KBits*: An integer that indicates the length (in bits) of the secret keying material to be derived; *KBits* **shall** be less than or equal to $hBits \times (2^{32}-1)$.
3. *OtherInfo*: A bit string of context-specific data (see Section 5.5.1.2 for details).

Process:

1. $reps = \lceil KBits / hBits \rceil$.
2. If $reps > (2^{32}-1)$, then output an error indicator, and exit this process without performing the remaining actions.
3. Initialize a 32-bit, big-endian bit string *counter* as 00000001_{16} (i.e. 0x00000001).
4. If *counter* || *Z* || *OtherInfo* is more than *max_H_inputBits* bits long,
   then output an error indicator, and exit this process without performing the remaining actions.
5. For *i* = 1 to *reps* by 1, do the following:

   5.1 Compute *K*(*i*) = H(*counter* || *Z* || *OtherInfo*).
   5.2 Increment *counter* (modulo 2^{32}), treating it as an unsigned 32-bit integer.
6. Let *K_Last* be set to *K*(*reps*) if (*KBits* / *hBits*) is an integer; otherwise, let *K_Last* be set to the (*KBits* mod *hBits*) leftmost bits of *K*(*reps*).
7. Output *DerivedKeyingMaterial* = *K*(*1*) || *K*(2) || … || *K*(*reps-1*) || *K_Last*.

Output:

The bit string *DerivedKeyingMaterial* (of length *KBits* bits), or an error indicator.

Errors:

1. The intended bit length of the derived keying material (as specified by *KBits*) is too long.
2. The bit string *Z* || *OtherInfo* (when prepended with a 32-bit counter) is too long.

Notes: When an **approved** key-establishment scheme employs this single-step KDF and OtherInfo is used, the participants **shall** know which entity is acting as party U and which entity is acting as party V to ensure (among other things) that they will derive the same keying material. (See Sections 8 and 9 for descriptions of the specific actions required of parties U and V during the execution of each of the **approved** key-establishment schemes that use a key-derivation method.) The roles of parties U and V **shall** be assigned to the key-establishment participants by the protocol employing the key-establishment scheme.

In step 5.1 above, the entire output of the hash function *hash* **shall** be used, whether H(*x*) = *hash*(*x*) or H(*x*) = HMAC-*hash*(*salt*, *x*). Therefore, the bit length of each output block of H is *hBits* bits. Some of the **approved** hash functions (see Section 5.1) are defined with an internal truncation operation (e.g., SHA-384). In these cases, the "entire output" of *hash* is the output value (e.g., for SHA-384, the entire output is defined to be the 384 bits resulting from the internal truncation, so *hBits* = 384, in this case). Any truncation performed by the KDF (external to *hash*) is done in step 6 above.

5.5.1.2 *OtherInfo*

The bit string *OtherInfo* **should** be used to ensure that the derived keying material is adequately "bound" to the context of the key-establishment transaction. Although other methods may be used to bind keying material to the transaction context, this Recommendation makes no statement as to the adequacy of these other methods. Failure to adequately bind the derived keying material to the transaction context could adversely affect the types of assurance that can be provided by certain key-agreement schemes.

Context-specific information that may be appropriate for inclusion in *OtherInfo*:

- Public information about parties U and V, such as their identifiers.
- The public keys contributed by each party to the key-establishment transaction. (One could, for example, include a certificate that contains the public key.)
- Other public and/or private information shared between parties U and V before or during the transaction, such as nonces or secret data already shared by parties U and V.
- An indication of the protocol or application employing the key-derivation method.
- Protocol-related information, such as a label or session identifier.
- The desired length of the derived keying material.
- An indication of the key-establishment scheme and/or key-derivation method used.
- An indication of various parameter or primitive choices (e.g., hash functions, MAC tag lengths, etc.).
- An indication of how the derived keying material should be parsed, including an indication of which algorithm(s) will use the (parsed) keying material.

For rationale in support of including entity identifiers, scheme identifiers, and/or other information in *OtherInfo*, see Appendix B of [SP 800-56A]

If *OtherInfo* is used, the meaning of each information item and each item's position within the *OtherInfo* bit string **shall** be specified. In addition, each item of information included in *OtherInfo* **shall** be unambiguously represented. For example, *OtherInfo* could take the form of a fixed-length bit string, or, if greater flexibility is needed, *OtherInfo* could be represented in a *Datalen* || *Data* format, where *Data* is a variable-length string of zero or more (eight-bit) bytes, and *Datalen* is a fixed-length, big-endian counter that indicates the length (in bytes) of *Data*. These requirements can be satisfied, for example, by using ASN.1 DER encoding as specified in 5.5.1.2.2 for *OtherInfo*.

Recommended formats for *OtherInfo* are specified in Sections 5.5.1.2.1 and 5.5.1.2.2. One of these two formats **should** be used by the single-step KDF specified in Section 5.5.1.1 when the auxiliary function employed is H = *hash*. When *OtherInfo* is included during the key-derivation process, and the recommended formats are used, the included items of information **shall** be divided into (three, four, or five) subfields as defined below.

AlgorithmID: A required non-null subfield that indicates how the derived keying material will be parsed and for which algorithm(s) the derived secret keying material will be used. For example, *AlgorithmID* might indicate that bits 1 to 112 are to be used as a 112-bit HMAC key and that bits 113 to 240 are to be used as a 128-bit AES key.

PartyUInfo: A required non-null subfield containing public information about party U. At a minimum, *PartyUInfo* **shall** include ID_U, an identifier for party U, as a distinct item of information. This subfield could also include information about the public key (if any) contributed to the key-establishment transaction by party U. Although the schemes specified in the Recommendation do not require the contribution of a nonce by party U, any nonce provided by party U **should** be included in this subfield.

PartyVInfo: A required non-null subfield containing public information about party V. At a minimum, *PartyVInfo* **shall** include ID_V, an identifier for party V, as a distinct item of information. This subfield could also include information about the public key(s) contributed to the key-establishment transaction by party V. When this KDF is used in a KAS1 scheme (see Section 8.2), the nonce, $N_{V,}$ supplied by party V **shall** be included in this field.

SuppPubInfo: An optional subfield that contains additional, mutually known public information (e.g., *KBits*, an identifier for the particular key-establishment scheme that was used to determine *Z*, an indication of the protocol or application employing that scheme, a session identifier, etc. This is particularly useful if these aspects of the key-establishment transaction can vary). While an implementation may be capable of including this subfield, the subfield may be null for a given transaction.

SuppPrivInfo: An optional subfield that contains additional, mutually known private information (e.g., a secret symmetric key that has been communicated through a separate channel). While an implementation may be capable of including this subfield, the subfield may be null for a given transaction.

5.5.1.2.1 The Concatenation Format for *OtherInfo*

This section specifies the concatenation format for *OtherInfo*. This format has been designed to provide a simple means of binding the derived keying material to the context of the key-establishment transaction, independent of other actions taken by the relying application. Note: When the single-step KDF specified in Section 5.5.1.1 is used with H = *hash* as the auxiliary function and this concatenation format for *OtherInfo*, the resulting key-derivation method is the Concatenation Key-Derivation Function specified in the original version of SP 800-56A.

For this format, *OtherInfo* is a bit string equal to the following concatenation:

$$AlgorithmID \,\|\, PartyUInfo \,\|\, PartyVInfo \,\{\|\, SuppPubInfo\,\}\{\|\, SuppPrivInfo\,\},$$

where the five subfields are bit strings comprised of items of information as described in Section 5.5.1.2.

Each of the three required subfields *AlgorithmID*, *PartyUInfo*, and *PartyVInfo* **shall** be the concatenation of a pre-determined sequence of substrings in which each substring represents a distinct item of information. Each such substring **shall** have one of these two formats: either it is a fixed-length bit string, or it has the form *Datalen* || *Data* – where *Data* is a variable-length string of zero or more (eight-bit) bytes, and *Datalen* is a fixed-length, big-endian counter that indicates the length (in bytes) of *Data*. (In this variable-length format, a null string of data **shall** be represented by a zero value for *Datalen*, indicating the absence of following data.) A protocol using this format for *OtherInfo* **shall** specify the number, ordering and meaning of the information-bearing substrings that are included in each of the subfields *AlgorithmID*, *PartyUInfo*, and *PartyVInfo*, and **shall** also specify which of the two formats (fixed-length or variable-length) is used by each such substring to represent its distinct item of information. The protocol **shall** specify the lengths for all fixed-length quantities, including the *Datalen* counters.

Each of the optional subfields *SuppPrivInfo* and *SuppPubInfo* (when allowed by the protocol employing the one-step KDF) **shall** be the concatenation of a pre-determined sequence of substrings representing additional items of information that may be used during key derivation upon mutual agreement of parties U and V. Each substring representing an item of information **shall** be of the form *Datalen* || *Data*, where *Data* is a variable-length string of zero or more (eight-bit) bytes, and *Datalen* is a fixed-length, big-endian counter that indicates the length (in bytes) of *Data*; the use of this form for the information allows U and V to omit a particular information item without confusion about the meaning of the other information that is provided in the *SuppPrivInfo* or *SuppPubInfo* subfield. The substrings representing items of information that parties U and V choose not to contribute are set equal to *Null*, and are represented in this variable-length format by setting *Datalen* equal to zero. If a protocol allows the use of the *OtherInfo* subfield *SuppPrivInfo* and/or the subfield *SuppPubInfo*, then the protocol **shall** specify the number, ordering and meaning of additional items of information that may be used in the allowed subfield(s) and **shall** specify the fixed-length of the *Datalen* counters.

5.5.1.2.2 The ASN.1 Format for *OtherInfo*

The ASN.1 format for *OtherInfo* provides an alternative means of binding the derived keying material to the context of the key-establishment transaction, independent of other actions taken by the relying application. Note: When the single-step KDF specified in Section 5.5.1.1 is used with H = *hash* as the auxiliary function and with this ASN.1 format for *OtherInfo*, the resulting key-derivation method is the ASN.1 Key-Derivation Function specified in the original version of SP 800-56A.

For the ASN.1 format, *OtherInfo* is a bit string resulting from the ASN.1 Distinguished Encoding Rules (DER) encoding (see [ISO/IEC 8825]) of a data structure comprised of a sequence of three required subfields *AlgorithmID*, *PartyUInfo*, and *PartyVInfo*, and, optionally, a subfield *SuppPubInfo* and/or a subfield *SuppPrivInfo* – as described in Section 5.5.1.2. A protocol using this format for *OtherInfo* **shall** specify the type, ordering and number of distinct items of information included in each of the (three, four, or five) subfields employed.

5.5.1.2.3 Other Formats for *OtherInfo*

Formats other than those provided in Sections 5.5.1.2.1 and 5.5.1.2.2 (e.g., those providing the items of information in a different arrangement) may be used for *OtherInfo*, but the context-specific information described in the preceding sections **should** be included (see the discussion in Section 5.5.1.1). This Recommendation makes no statement as to the adequacy of other formats.

5.5.2 The Extraction-then-Expansion Key-Derivation Procedure

This Recommendation permits the use of an **approved** extraction-then-expansion key-derivation procedure as an alternative to the single-step key-derivation function specified in Section 5.5.1. When the extraction-then-expansion key-derivation procedure is employed in an **approved** key-agreement scheme, the secret keying material is derived in two steps using the shared secret Z (represented as a byte string) as input (along with a salt and additional data, such as that included in the *OtherInfo* used by the KDFs specified above; also see Appendix B for guidance). The first step is called (randomness) extraction, and the second step is called (key) expansion. The details of the **approved** extraction-then-expansion key-derivation procedure are specified in [SP 800-56C].

5.5.3 Application-Specific Key-Derivation Methods

Additional **approved** application-specific key-derivation methods are enumerated in [SP 800-135]. A routine that implements a key-derivation procedure specified in [SP 800-135] **shall** destroy any local copies of sensitive input values, as well as any other locally stored values used or produced during its execution (including any local copies of portions of the derived keying material). Their destruction **shall** occur prior to or during any exit from the routine – whether exiting early because of an error or exiting normally, with the output of keying material.

5.6 Key Confirmation

The term key confirmation (KC) refers to actions taken to provide assurance to one party (the key-confirmation recipient) that another party (the key-confirmation provider) is in possession of a (supposedly) shared secret and/or to confirm that the other party has the correct version of keying material that was derived or transported during a key-establishment transaction (correct, that is, from the perspective of the key-confirmation recipient.) Such actions are said to provide unilateral key confirmation when they provide this assurance to only one of the participants in the key-establishment transaction; the actions are said to provide bilateral key confirmation when this assurance is provided to both participants (i.e., when unilateral key confirmation is provided in both directions).

Oftentimes, key confirmation is obtained (at least implicitly) by some means that are external to the key-establishment scheme employed during a transaction (e.g., by using a symmetric key that was established during the transaction to decrypt an encrypted message sent later by the key-confirmation provider), but this is not always the case. In some circumstances, it may be appropriate to incorporate the exchange of explicit key-confirmation information as an integral part of the key-establishment scheme itself. The inclusion of key confirmation may enhance the security services that can be offered by a key-establishment scheme. For example, the key-establishment schemes incorporating key confirmation that are specified in this Recommendation could be used to provide the KC recipient with assurance that the KC provider is in possession of the private key corresponding to the provider's public key-establishment key, from which the recipient may infer that the provider is the owner of that key pair.

For key confirmation to comply with this Recommendation, key confirmation **shall** be incorporated into an **approved** key-establishment scheme as specified in the sections that follow. If any other methods are used to provide key confirmation, this Recommendation makes no statement as to their adequacy.

5.6.1 Unilateral Key Confirmation for Key-Establishment Schemes

As specified in this Recommendation, unilateral key confirmation occurs when one participant in the execution of a key-establishment scheme (the key-confirmation "provider") demonstrates to the satisfaction of the other participant (the key-confirmation "recipient") that both the KC provider and the KC recipient have possession of the same secret *MacKey*.

MacKey **shall** be a symmetric key that is unique to a specific execution of a key-establishment scheme and (from the perspective of the KC provider) **shall** be unpredictable prior to that key-establishment transaction. In the case of a key-agreement scheme, *MacKey* is derived using the shared secret *Z* created during the execution of that scheme (see Section 5.5 for the details of key derivation). In the case of a key-transport scheme, *MacKey* is included as part of the transported keying material. Section 5.6.1.1 specifies how *MacKey* is to be extracted from the derived or transported keying material.

MacKey and certain context-specific *MacData* (as specified in Sections 5.6.1.1) are used by the KC provider as input to an **approved** MAC algorithm to obtain a MAC tag that is sent to the KC recipient. The recipient performs an independent computation of the MAC tag. If the MAC tag value computed by the KC recipient matches the MAC tag value received from the KC provider, then key confirmation is successful. (See Section 5.2 for MAC-tag generation and verification, and Section 5.6.3 for a discussion of MAC-tag security.)

In the case of a key-agreement scheme (see Sections 8.2.3 and 8.3.3), successful key confirmation provides assurance to the KC recipient that the same *Z* value has been used by both parties to correctly derive the keying material (which includes *MacKey*). In the case of a key-transport scheme (see Sections 9.2.4 and 9.3.4), successful key confirmation provides assurance to the KC recipient (who sent the keying material) that the transported keying material (which includes *MacKey*) has been correctly decrypted or unwrapped by the party to whom it was sent.

A close examination of the KC process shows that each of the pair-wise key-establishment schemes specified in this Recommendation that incorporate key confirmation can be used to provide the KC recipient with assurance that the KC provider is currently in possession of the (correct) private key – the one corresponding to the KC provider's public key-establishment key. The use of transaction-specific values for both *MacKey* and *MacData* prevents (for all practical purposes) the replay of any previously computed value of *MacTag*. The receipt of a correctly computed MAC tag provides assurance to the KC recipient that the KC provider has used the correct private key during the current transaction – to successfully recover the secret data that are a prerequisite to learning the value of *MacKey*.

5.6.1.1 Adding Unilateral Key Confirmation to a Key-Establishment Scheme

To include unilateral key confirmation, the following steps **shall** be incorporated into the scheme. (Additional details will be provided for each scheme in the appropriate subsections of Sections 8 and 9.) In the discussion that follows, the key-confirmation provider, P, may be either party U or party V, as long as the KC provider, P, contributes a key pair to the key-establishment transaction. The key-confirmation recipient, R, is the other party.

1. The provider, P, computes

   $MacData_P = message\_string_P \parallel ID_P \parallel ID_R \parallel EphemData_P \parallel EphemData_R \{\parallel Text_P\}$

   where

   - *message_string*$_P$ is a six-byte character string, with a value of "KC_1_U" when party U is providing the MAC tag, or "KC_1_V" when party V is providing the MAC tag. (Note that these values will be changed for bilateral key confirmation, as specified in Section 5.6.2.)
   - ID_P is the identifier used to label the key-confirmation provider.
   - ID_R is the identifier used to label the key-confirmation recipient.
   - $EphemData_P$ and $EphemData_R$ are (ephemeral) values contributed by the KC provider and recipient, respectively. These values are specified in the sections describing the schemes that include key confirmation.
   - $Text_P$ is an optional bit string that may be used during key confirmation and that is known by both parties.

   The content of each of the components that are concatenated to form $MacData_P$ **shall** be precisely defined and unambiguously represented. A particular component's content may be represented, for example, as a fixed-length bit string or in the form *Datalen* || *Data*, where *Data* is a variable-length string of zero or more (eight-bit) bytes, and *Datalen* is a fixed-length, big-endian counter that indicates the length (in bytes) of *Data*. These requirements could also be satisfied by using a specific ASN.1 DER encoding of each component. It is imperative that the provider and recipient have agreed upon the content and format that will be used for each component of $MacData_P$.

   MacData **shall** include a non-null identifier, ID_P, for the key-confirmation provider. Depending upon the circumstances, the key-confirmation recipient's identifier, ID_R, may be replaced by a null string. The rules for selecting ID_P and ID_R are as follows:

   As specified in this Recommendation, the key-confirmation provider must own a key pair that is employed by the basic key-establishment scheme (KAS1-basic, KAS2-basic, KTS-OAEP-basic, or KTS-KEM-KWS-basic) that determines the *MacKey* value used in the key-confirmation computations performed during the transaction. The identifier, ID_p, included in $MacData_P$ **shall** be one that has a trusted association with the public key of that key pair.

   If the key-confirmation recipient also owns a key pair that is employed by the basic key-establishment scheme used during the transaction, then the identifier, ID_R, included in $MacData_P$ **shall** be one that has a trusted association with the public key of that key pair.

   If the key-confirmation recipient does not own a key pair employed for key-establishment purposes, and no identifier has been used to label that party during the execution of the basic key-establishment scheme employed by the transaction, then ID_R may be replaced by a null string. However, if an identifier is desired/required for that party for key confirmation purposes, then a non-null value for ID_R, **shall** be selected/assigned in accordance with the requirements of the protocol relying upon the transaction.

2. Whenever a particular identifier has been used to label the key-confirmation recipient or key-confirmation provider in the execution of the basic key-establishment scheme used during the transaction, that same identifier **shall** be used as ID_P or ID_R, respectively, in the $MacData_P$ used during key confirmation. For example, if party U is the key-confirmation recipient, and ID_U has been used to label party U in the *OtherInfo* employed by the key-derivation method of a key-agreement scheme used during the transaction, then the $MacData_P$ used during key confirmation **shall** have $ID_R = ID_U$.

   In the case of a key-agreement scheme: After computing the shared secret Z and applying the key-derivation function to obtain the derived keying material, *DerivedKeyingMaterial* (see Section 5.5), the KC provider uses agreed-upon bit lengths to parse *DerivedKeyingMaterial* into two parts, *MacKey* and *KeyData*:

   $$MacKey \,||\, KeyData = DerivedKeyingMaterial.$$

   In the case of a key-transport scheme: The KC provider parses the *TransportedKeyingMaterial* into *MacKey* and *KeyData*:

   $$MacKey \,||\, KeyData = TransportedKeyingMaterial.$$

3. Using an agreed-upon bit length *MacTagBits*, the KC provider computes $MacTag_P$ (see Sections 5.2.1 and 5.6.3):

   $$MacTag_P = T_{MacTagBits}[\text{MAC}\,(MacKey, MacData_P)],$$

   and sends it to the KC recipient.

4. The KC recipient forms $MacData_P$, determines *MacKey*, computes $MacTag_P$ in the same manner as the KC provider, and then compares its computed $MacTag_P$ to the value received from the provider. If the received value is equal to the computed value, then the recipient is assured that the provider has used the same value for *MacKey* and that the provider shares the recipient's value of $MacTag_P$.

Each participant **shall** destroy all copies of the *MacKey* that he employed for key-confirmation purposes during a particular pair-wise key-establishment transaction, once *MacKey* is no longer needed to provide or obtain key confirmation as part of that transaction.

If $MacTag_P$ cannot be verified by the KC recipient during a particular key-establishment transaction, then key confirmation has failed, and both participants **shall** destroy all of their copies of *MacKey* and *KeyData*. In particular, *MacKey* and *KeyData* **shall not** be revealed by either participant to any other party (not even to the other participant), and the keying material **shall not** be used for any further purpose. In the case of a key-confirmation failure, the key-establishment transaction **shall** be terminated.

Note: The key-confirmation routines employed by the KC provider and KC recipient **shall** destroy all local copies of *MacKey*, *KeyData*, *MacData*, and any other locally stored values used or produced during their execution. Their destruction **shall** occur prior to or during any exit from those routines – whether exiting normally or exiting early, because of an error.

Unilateral key confirmation, as specified in this Recommendation, can be incorporated into any key-establishment scheme in which the key-confirmation provider is required to own a key-establishment key pair that is used in the key-establishment process. Unilateral key confirmation may be added in either direction to a KAS2 scheme (see Sections 8.3.3.2 and 8.3.3.3); it may also be added to a KAS1, KTS-OAEP, or KTS-KEM-KWS scheme, but only with party V (the party contributing the key pair) acting as the key-confirmation provider, and party U acting as the key-confirmation recipient (see Sections 8.2.3.1, 9.2.4.2, and 9.3.4.2).

5.6.2 Bilateral Key Confirmation for KAS2 Schemes

Bilateral key confirmation, as specified in this Recommendation, can be incorporated into a KAS2 key-agreement scheme, since each party is required to own a key-establishment key pair that is used in the key-agreement process. Bilateral key confirmation is accomplished by performing unilateral key confirmation in both directions (with party U providing $MacTag_U$ to KC recipient V, and party V providing $MacTag_V$ to KC recipient U) during the same scheme.

To include bilateral key confirmation, two instances of unilateral key confirmation (as specified in Section 5.6.1.1, subject to the modifications listed below) **shall** be incorporated into the KAS2 scheme, once with party U as the key-confirmation provider (i.e., P = U and R = V), and once with party V as the key-confirmation provider (i.e., P = V and R = U). Additional details will be provided in Section 8.3.3.4.

In addition to setting P = U and R = V in one instance of the unilateral key-confirmation procedure described in Section 5.6.1.1 and setting P = V and R = U in a second instance, the following changes/clarifications apply when using the procedure for bilateral key confirmation:

1. When computing $MacTag_U$, the value of $message\_string_U$ that forms the initial segment of $MacData_U$ is the six-byte character string "KC_2_U".

2. When computing $MacTag_V$, the value of $message\_string_V$ that forms the initial segment of $MacData_V$ is the six-byte character string "KC_2_V".

3. If used at all, the value of the (optional) byte string $Text_U$ used to form the final segment of $MacData_U$ can be different than the value of the (optional) byte string $Text_V$ used to form the final segment of $MacData_V$, provided that both parties are aware of the value(s) used.

4. The identifiers used to label the parties U and V when forming $MacData_U$ **shall** be the same as the identifiers used to label the parties U and V when forming $MacData_V$, although ID_U and ID_V will play different roles in the two strings. If $ID_P = ID_U$ and $ID_R = ID_V$ are used in $MacData_U$, then $ID_P = ID_V$ and $ID_R = ID_U$ are used in $MacData_V$.

5.6.3 Minimum Requirements for *MacKey* and *MacTag*

For compliance with this Recommendation, a MAC tag used for key confirmation **shall** be generated using an **approved** MAC algorithm, which can be HMAC [FIPS 198] with an

approved hash function or AES CMAC [SP 800-38B] with a key length of 128, 192, or 256 bits (see Section 5.2). *MacKey* **shall** be at least 112 bits in length when a 2048-bit RSA modulus is used during the transaction, and at least 128 bits in length when a 3072-bit modulus is used. The bit length of *MacTag* **shall** be at least 64 bits for key-confirmation in this Recommendation. For other applications, please refer to the relevant specifications for the minimum length required for message authentication codes.

In cases where key confirmation is incorporated in a key-transport scheme as specified in this Recommendation, the method used to generate the MAC key that is included in the transported keying material **shall** comply with the methods specified in [SP 800-133] for generating symmetric keys. The MAC key and MAC algorithm **shall** be capable of supporting at least a 112-bit security strength when a 2048-bit RSA modulus is used during the transaction, and of supporting at least a 128-bit security strength when a 3072-bit modulus is used.

6 RSA Key Pairs

6.1 General Requirements

The following are requirements on RSA key pairs (see the Recommendation for Key Management [SP 800-57]):

1. Each key pair **shall** be created using an **approved** key-generation method as specified in Section 6.3.

2. The private keys and prime factors of the modulus **shall** be protected from unauthorized access, disclosure, and modification.

3. Public keys **shall** be protected from unauthorized modification. This is often accomplished by using public-key certificates that have been signed by a Certification Authority (CA).

4. A recipient of a public key **shall** be assured of the integrity and correct association of (a) the public key and (b) the identifier of the entity that owns the key pair (that is, the party with whom the recipient intends to establish secret keying material). This assurance is often provided by verifying a public-key certificate that was signed by a trusted third party (for example, a CA), but may be provided by direct distribution of the public key and identifier from the owner, provided that the recipient trusts the owner and distribution process to do this.

5. One key pair **shall not** be used for different cryptographic purposes (for example, a digital-signature key pair **shall not** be used for key establishment or vice versa), with the following possible exception: when requesting the certificate for a public key-establishment key, the private key-establishment key associated with the public key may be used to sign the certificate request (see SP 800-57, Part 1 on Key Usage for further information). A key pair may be used in more than one key-establishment scheme. However, a key pair used for schemes specified in this Recommendation **should not** be used for any schemes not specified herein.

6. The owner of a key pair **shall** have assurance of the key pair's validity (see Section 6.4.1.1); that is, the owner **shall** have assurance of the correct generation of the key pair

(see Section 6.3), consistent with the criteria of Section 6.2; assurance of private and public-key validity; and assurance of pair-wise consistency.

7. A recipient of a public key **shall** have assurance of the validity of the public key (see Section 6.4.2.1). This assurance may be provided, for example, through the use of a public-key certificate if the CA obtains sufficient assurance of public-key validity as part of its certification process.
8. A recipient of a public key **shall** have assurance of the owner's possession of the associated private key (see Section 6.4.2.3). This assurance may be provided, for example, through the use of a public key certificate if the CA obtains sufficient assurance of possession as part of its certification process.

6.2 Criteria for RSA Key Pairs for Key Establishment

6.2.1 Definition of a Key Pair

A valid RSA key pair, in its basic form, **shall** consist of an RSA public key (n, e) and an RSA private key (n, d), where:

1. n, the public modulus, **shall** be the product of exactly two distinct, odd positive prime factors, p and q, that are kept secret. Let $nBits$ be the length of n in bits.

2. The public exponent e **shall** be an odd integer that is selected prior to the generation of p and q such that:

$$65\,537 \leq e < 2^{256}$$

3. The prime factors p and q **shall** be generated using one of the methods specified in Appendix B.3 of [FIPS 186] such that:

   a. $(\sqrt{2})(2^{nBits/2-1}) \leq p \leq (2^{nBits/2} - 1)$.
   b. $(\sqrt{2})(2^{nBits/2-1}) \leq q \leq (2^{nBits/2} - 1)$.
   c. $|p - q| > 2^{nBits/2-100}$.
   d. GCD(e, LCM(p-1, q-1)) = 1.
4. The private exponent d **shall** be selected such that:

   a. $2^{nBits/2} < d <$ LCM((p-1), (q-1)), and

   b. $d = e^{-1}$ mod (LCM((p-1), (q-1))).

Note that these criteria are also specified in [FIPS 186].

6.2.2 Formats

The RSA private key may be expressed in several formats. The basic format of the RSA private key consists of the modulus n and a private-key exponent d that depends on n and the public-key exponent e; this format is used throughout this Recommendation. The other two formats may be used in implementations, but may require appropriate modifications for correct implementation. To facilitate implementation testing, the format for the private key **shall** be one of the following:

1. The basic format: (n, d).

2. The prime-factor format: (p, q, d).
3. The Chinese Remainder Theorem (CRT) format: (n, e, d, p, q, dP, dQ, $qInv$), where $dP = d \bmod (p - 1)$, $dQ = d \bmod (q - 1)$, and $qInv = q^{-1} \bmod p$.

Key-pair generators and key-pair validation methods are given for each of these formats in Sections 6.3 and 6.4, respectively.

6.3 RSA Key-Pair Generators

The key pairs employed by the key-establishment schemes specified in this Recommendation **shall** be generated using the techniques specified in Appendix B.3 of [FIPS 186], employing the requisite methods for prime-number generation, primality testing, etc., that are specified in Appendix C of that document.

An **approved** RSA key-pair generator **shall** be used to produce a random RSA key pair with a modulus length of either 2048 or 3072 bits, possibly using other inputs during the process. Key-pair generators require the use of an **approved** random bit generator (RBG). See Section 5.3.

The **approved** RSA key-pair generators provided in Sections 6.3.1 and 6.3.2 are differentiated by the method for determining the public-key exponent e that is used as part of an RSA public key (i.e., (n, e)); Section 6.3.1 addresses the use of a fixed value for the exponent, whereas Section 6.3.2 uses a randomly generated value.

6.3.1 RSAKPG1 Family: RSA Key-Pair Generation with a Fixed Public Exponent

The RSAKPG1 family of key-pair generation methods consists of three RSA key-pair generators where the public exponent has a fixed value (see Section 6.2).

Three representations are addressed:

1. *rsakpg1-basic* generates the private key in the basic format (n, d),
2. *rsakpg1-prime-factor* generates the private key in the prime-factor format (p, q, d), and
3. *rsakpg1-crt* generates the private key in the Chinese Remainder Theorem format (n, e, d, p, q, dP, dQ, $qInv$).

An implementation may perform a key-pair validation before outputting the key pair from the generator. The key-pair validation methods for this family are specified in Section 6.4.1.2.

6.3.1.1 *rsakpg1-basic*

rsakpg1-basic is the generator in the RSAKPG1 family where the private key is in the basic format (n, d).

Function call: *rsakpg1-basic*(s, $nBits$, e)

Input:

1. s: the target security strength;

2. *nBits*: the intended length in bits of the RSA modulus; and
3. *e*: a fixed public exponent – an odd integer, such that $65\,537 \le e < 2^{256}$.

Process:

1. Check the values:
   a. If s is not the integer 112 or 128, output an indication that the security strength is incorrect, and exit without further processing.
   b. If s =112 and $nBits \neq 2048$, or if s =128 and $nBits \neq 3072$, output an indication that the modulus length is incorrect, and exit without further processing.
   c. If e is not an odd integer such that $65\,537 \le e < 2^{256}$, output an indication that the exponent is out of range, and exit without further processing.
2. Generate the prime factors p and q, as specified in [FIPS 186].
3. Determine the private exponent d:

   $$d = e^{-1} \bmod \mathrm{LCM}(p-1, q-1)\ .$$

   In the event that no such d exists, or in the very rare event that $d \le 2^{nBits/2}$, discard the results of all computations and repeat the process, starting at step 2.
4. Determine the modulus n as $n = p \cdot q$.
5. Perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n$-1. If an inconsistency is found, output an indication of a pair-wise consistency failure, and exit without further processing.
6. Output (n, e) as the public key, and (n, d) as the private key.

Output:

1. (n, e): the RSA public key, and
2. (n, d): the RSA private key in the basic format.

Errors: Indications of the following:

1. The security strength is incorrect,
2. The modulus length is incorrect,
3. The fixed public exponent is out of range, or
4. Pair-wise consistency failure.

Note that key-pair validation, as specified in Section 6.4.1.2.1, can be performed after step 5 and before step 6. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of *p*, *q*, and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.3.1.2 *rsakpg1-prime-factor*

rsakpg1-prime-factor is the generator in the RSAKPG1 family where the private key is in the prime factor format (*p*, *q*, *d*).

Function call: *rsakpg1-prime-factor*(*s*, *nBits*, *e*)

The inputs, outputs and errors are the same as in *rsakpg1-basic* (see 6.3.1.1), except that the private key is in the prime-factor format: (*p*, *q*, *d*).

The steps are the same as in *rsakpg1-basic***,** except that processing Step 6 is replaced by the following:

6. Output (*n*, *e*) as the public key, and (*p*, *q*, *d*) as the private key.

Note that key-pair validation, as specified in Section 6.4.1.2.2, can be performed after step 5 and before step 6. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of *p*, *q*, and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.3.1.3 *rsakpg1-crt*

rsakpg1-crt is the generator in the RSAKPG1 family where the private key is in the Chinese Remainder Theorem format (*n*, *e*, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

Function call: *rsakpg1-crt*(*s*, *nBits*, *e*)

The inputs, outputs and errors are the same as in *rsakpg1-basic* (see 6.3.1.1), except that the private key is in the Chinese Remainder Theorem format: (*n*, *e*, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

The steps are the same as in *rsakpg1-basic***,** except that processing steps 5 and 6 are replaced by the following:

5. Determine the components *dP*, *dQ* and *qInv*:

    a. $dP = d \bmod (p - 1)$.

    b. $dQ = d \bmod (q - 1)$.

c. $qInv = q^{-1} \bmod p$.

6. Perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n-1$. If an inconsistency is found, output an indication of a pair-wise consistency failure, and exit without further processing.
7. Output (n, e) as the public key, and $(n, e, d, p, q, dP, dQ, qInv)$ as the private key.

Note that key-pair validation, as specified in Section 6.4.1.2.3, can be performed after step 6 and before step 7. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of *p*, *q*, *dP*, *dQ*, *qInv,* and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.3.2 RSAKPG2 Family: RSA Key-Pair Generation with a Random Public Exponent

The RSAKPG2 family of key-pair generation methods consists of three RSA key-pair generators where the public exponent *e* is a random value in the range $65\,537 \leq e < 2^{256}$.

Three representations are addressed:

1. *rsakpg2-basic* generates the private key in the basic format (*n*, *d*),
2. *rsakpg2-prime-factor* generates the private key in the prime factor format (*p*, *q*, *d*), and
3. *rsakpg2-crt* generates the private key in the Chinese Remainder Theorem format (*n*, *e*, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

An implementation may perform a key-pair validation before outputting the key pair from the generation function. The key-pair validation methods for this family are specified in Section 6.4.1.3.

6.3.2.1 *rsakpg2-basic*

rsakpg2-basic is the generator in the RSAKPG2 family where the private key is in the basic format (*n*, *d*).

Function call: *rsakpg2-basic*(*s*, *nBits*, *eBits*)

Input:

1. *s*: the target security strength;
2. *nBits*: the intended length in bits of the RSA modulus; and
3. *eBits*: the intended length in bits of the public exponent − an integer such that $17 \leq eBits \leq 256$. Note that the public exponent **shall** be an odd integer, such that $65\,537 \leq e < 2^{256}$.

Process:

1. Check the values:

    a. If s is not the integer 112 or 128, output an indication that the security strength is incorrect, and exit without further processing.

    b. If s =112 and $nBits \neq 2048$, or if s =128 and $nBits \neq 3072$, output an indication that the modulus length is incorrect, and exit without further processing.

    c. If $eBits$ is not an integer such that $17 \leq eBits \leq 256$, output an indication that the exponent length is out of range, and exit without further processing.

2. Generate an odd public exponent e in the range $[2^{eBits-1} + 1, 2^{eBits} - 1]$ using an **approved** RBG that supports a minimum security strength of s bits (see Section 5.3).

3. Generate the prime factors p and q as specified in [FIPS 186]

4. Determine the private exponent d:

    $$d = e^{-1} \bmod \mathrm{LCM}(p - 1, q - 1).$$

    In the event that no such d exists, or in the very rare event that $d \leq 2^{nBits/2}$, discard the results of all computations and repeat the process, starting at step 2.

5. Determine the modulus n as $n = p \cdot q$.

6. Perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n-1$. If an inconsistency is found, output an indication of a pair-wise consistency failure, and exit without further processing.

7. Output (n, e) as the public key, and (n, d) as the private key.

Output:

1. (n, e): the RSA public key, and

2. (n, d): the RSA private key in the basic format.

Errors: Indications of the following:

1. The security strength is incorrect,

2. The modulus length is incorrect,

3. The exponent length is out of range, or

4. Pair-wise consistency failure.

Note that key-pair validation, as specified in Section 6.4.1.3.1, can be performed after step 6 and before step 7. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of *p*, *q*, and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.3.2.2 *rsakpg2-prime-factor*

rsakpg2-prime-factor is the generator in the RSAKPG2 family where the private key is in the prime-factor format (*p*, *q*, *d*).

Function call: *rsakpg2-prime-factor*(*s*, *nBits*, *eBits*)

The inputs, outputs and errors are the same as in *rsakpg2-basic* (see 6.3.2.1), except that the private key is in the prime-factor format: (*p*, *q*, *d*).

The steps are the same as in *rsakpg2-basic*, except that processing Step 7 is replaced by the following:

7. Output (*n*, *e*) as the public key, and (*p*, *q*, *d*) as the private key.

Note that key-pair validation as specified in Section 6.4.1.3.2 can be performed after step 6 and before step 7. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of *p*, *q*, and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.3.2.3 *rsakpg2-crt*

rsakpg2-crt is the generator in the RSAKPG2 family where the private key is in the Chinese Remainder Theorem format (*n*, *e*, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

Function call: *rsakpg2-crt*(*s*, *nBits*, *eBits*)

The inputs, outputs and errors are the same as in *rsakpg2-basic* (see 6.3.2.1), except that the private key is in the Chinese Remainder Theorem format: (*n*, *e*, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

The steps are the same as in *rsakpg2-basic,* except that processing Steps 6 and 7 are replaced by the following:

6. Determine the components *dP*, *dQ* and *qInv*:

    a. $dP = d \bmod (p - 1)$.

   b. $dQ = d \bmod (q - 1)$.

   c. $qInv = q^{-1} \bmod p$.

7. Perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n$-1. If an inconsistency is found, output an indication of a pair-wise consistency failure, and exit without further processing.

8. Output (n, e) as the public key, and (n, e, d, p, q, dP, dQ, $qInv$) as the private key.

Note that key-pair validation as specified in Section 6.4.1.3.3 can be performed after step 7 and before step 8. If an error is detected, output an indication of a key-pair validation failure, and exit without further processing.

A routine that implements this generation function **shall** destroy any local copies of p, q, dP, dQ, $qInv$, and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of an RSA key pair). Note that the requirement for destruction includes any locally stored portions of the output key pair.

6.4 Required Assurances

Secure key establishment depends upon the use of valid key-establishment keys. The security of key-establishment schemes also depends on limiting knowledge of the private keys to those who have been authorized to use them (i.e., their respective owners) and to the trusted third party that may have generated them[1]. In addition to preventing unauthorized entities from gaining access to private keys, it is also important that owners have access to their correct private keys.

To explain the assurance requirements, some terminology needs to be defined. The owner of a key pair is the entity that is authorized to use the private key that corresponds to the owner's public key, whether or not the owner generated the key pair. The recipient of a public key is the entity that is participating in a key-establishment transaction with the owner and obtains the owner's public key before or during the current transaction.

Prior to or during a key-establishment transaction, the participants in the transaction (i.e., parties U and V**) shall** obtain the appropriate assurances about the key pairs used during that transaction. The types of assurance that may be sought by one or both of the parties (U and/or V) concerning the components of a key pair (i.e., the private key and public key) are discussed in Sections 6.4.1 and 6.4.2.

6.4.1 Assurances Required by the Key-Pair Owner

Prior to the use of a key pair in a key-establishment transaction, the key-pair owner **shall** have assurance of the validity of the key pair. Assurance of key-pair validity provides assurance that a key pair was generated in accordance with the requirements in Sections 6.2 and 6.3. Key-pair validity implies public-key validity and assurance of possession of the correct private key. Assurance of key-pair validity can only be provided by an entity that has the private key (e.g.,

[1] The trusted third party is trusted not to use or reveal the private keys.

the owner). Depending on an organization’s requirements, a renewal of key-pair validity may be prudent. The method of obtaining initial and renewed assurance of key-pair validity is addressed in Section 6.4.1.1.

Assurance of key-pair validity can be renewed at any time (see Section 6.4.1.1). As time passes, an owner may lose possession of the correct value of the private-key component of their key pair, e.g. due to an error; for this reason, renewed (i.e., current) assurance of possession of a private key can be of value for some applications. See Section 6.4.1.5 for techniques that the owner can use to obtain renewed assurance of private-key possession separately from assurance of key-pair validity.

6.4.1.1 Obtaining Owner Assurance of Key-Pair Validity

Key-pair validation **shall** be performed prior to the first use of the key pair in a key-establishment transaction (see Section 4.1). Assurance of key-pair validity **shall** be obtained by its owner using (all of) the following steps.

1. Key-pair generation: Assurance that the key pair has been correctly formed, in a manner consistent with the criteria of Section 6.2, is obtained using one of the following two methods:
    a. Owner generation – The owner obtains the desired assurance if it generates the public/private key pair as specified in Section 6.3.
    b. TTP generation – The owner obtains the desired assurance when a trusted third party (TTP) who is trusted by the owner generates the public/private key pair as specified in Section 6.3 and provides it to the owner.
2. The owner **shall** perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n-1$. Note that if the owner generated the key pair (see method 1.a above), an initial pair-wise consistency test was performed during key generation (see Section 6.3). If a TTP generated the key pair and provided it to the owner (see method 1.b above), the owner **shall** perform the consistency check separately, prior to the first use of the key pair in a key-establishment transaction (see Section 4.1).
3. Key-pair validation: A key pair **shall** be validated using one of the following methods:
    a. The owner generated the key pair: The owner either
        1) Performs a successful key-pair validation during key-pair generation (see Section 6.3), or
        2) Performs a successful key-pair validation separately from key-pair generation (see Section 6.4.1.2 or 6.4.1.3).
    b. TTP key-pair validation: A trusted third party (trusted by the owner) either
        1) Performs a successful key-pair validation during key-pair generation (see Section 6.3), or
        2) Performs a successful key-pair validation separately from key-pair generation (as specified in Sections 6.4.1.2 or 6.4.1.3), and indicates the success to the

owner. Note that if the key-pair validation is performed separately from the key-pair generation, and the TTP does not have the key pair, then the party that generated the key pair or owns the key pair must provide it to the TTP.

c. The TTP generated the key pair (with or without performing key-pair validation), and the owner performs the key-pair validation – The owner performs a key-pair validation as specified in Section 6.4.1.4.

Note that the use of a TTP to generate a key pair or to perform key-pair validation for an owner means that the TTP must be trusted (by both the owner and any recipient) to not use the owner's private key to masquerade as the owner or otherwise compromise the key-establishment transaction.

The key pair can be revalidated at any time by the owner as follows:

A. Perform a pair-wise consistency test by verifying that $k = (k^e)^d \bmod n$ for some integer k satisfying $1 < k < n\text{-}1$, and

B. Perform a successful key-pair validation:

   1. If the intended value or length of the exponent is known, then perform a successful key-pair validation as specified in Section 6.4.1.2 or 6.4.1.3.

   2. If the intended value or length of the exponent is NOT known, then perform a successful key-pair validation as specified in Section 6.4.1.4.

6.4.1.2 RSAKPV1 Family: RSA Key-Pair Validation with a Fixed Exponent

The RSAKPV1 family of key-pair validation methods corresponds to the RSAKPG1 family of key-pair generation methods (see Section 6.3.1).

6.4.1.2.1 *rsakpv1-basic*

rsakpv1-basic is the key-pair validation method corresponding to *rsakpg1-basic* (see Section 6.3.1.1).

Function call: *rsakpv1-basic* (s, $nBits$, e_{fixed}, (n_{pub}, e_{pub}), (n_{priv}, d))

Input:

1. s: the target security strength;
2. $nBits$: the expected length in bits of the RSA modulus;
3. e_{fixed}: the intended fixed public exponent – an odd integer such that $65\,537 \le e_{fixed} < 2^{256}$;
4. (n_{pub}, e_{pub}): the RSA public key to be validated; and
5. (n_{priv}, d): the RSA private key to be validated in the basic format.

Process:

1. Check the ranges:

   a. If s is not the integer 112 or 128, output an indication that the security strength is incorrect, and exit without further processing.

   b. If *nBits* is not the integer 2048 or 3072, output an indication that the modulus length is incorrect, and exit without further processing.

   c. If e_{fixed} is not an odd integer such that $65\,537 \leq e_{fixed} < 2^{256}$, output an indication that the fixed exponent is out of range, and exit without further processing.

2. Compare the public exponents:

   If $e_{pub} \neq e_{fixed}$, output an indication that the request is invalid, and exit without further processing.

3. Check the modulus:

   a. If $n_{pub} \neq n_{priv}$, output an indication of an invalid key pair, and exit without further processing.

   b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

4. Prime factor recovery:

   a. Recover the prime factors p and q from the modulus n_{pub}, the public exponent e_{pub} and the private exponent d (see Appendix C):

      $(p, q) = \text{RecoverPrimeFactors}\ (n_{pub}, e_{pub}, d)$

   b. If RecoverPrimeFactors outputs an indication that the prime factors were not found, output an indication that the request is invalid, and exit without further processing.

   c. If $n_{pub} \neq pq$, then output an indication that the request is invalid, and exit without further processing.

5. Check the prime factors:

   a. Apply an **approved** primality test to test the prime number p (see Appendix C.3 in [FIPS 186]).

   b. If the primality test indicates that p is not prime, output an indication of an invalid key pair, and exit without further processing.

   c. If $(p < \sqrt{2}(2^{nBits/2-1}))$ or $(p > 2^{nBits/2} - 1)$, output an indication of an invalid key pair, and exit without further processing.

   d. If GCD $(p - 1, e_{pub}) \neq 1$, output an indication of an invalid key pair, and exit without further processing.

   e. Apply an **approved** primality test to test the prime number q (see Appendix C.3 in [FIPS 186]).

f. If the primality test indicates that q is not prime, output an indication of an invalid key pair, and exit without further processing.

g. If ($q < \sqrt{2}(2^{nBits/2-1})$) or ($q > 2^{nBits/2} - 1$), output an indication of an invalid key pair. and exit without further processing.

h. If GCD ($q - 1$, e_{pub}) $\neq 1$, output an indication of an invalid key pair, and exit without further processing.

i. If $|p - q| \leq 2^{nBits/2-100}$ output an indication of an invalid key pair, and exit without further processing.

6. Check that the private exponent d satisfies

   a. $2^{nBits/2} < d <$ LCM ($p - 1$, $q - 1$).

   and

   b. $1 = (d \cdot e_{pub})$ mod LCM ($p - 1$, $q - 1$).

   If either check fails, output an indication of an invalid key pair, and exit without further processing.

7. Output an indication that the key pair is valid.

Output:

1. *status*: An indication that the key pair is valid or an indication of an error.

Errors: Indications of the following:

1. The security strength is incorrect,
2. The modulus length is incorrect,
3. The fixed exponent is out of range,
4. The key pair is invalid.

A routine that implements this validation function **shall** destroy any local copies of p, q and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.2.2 *rsakpv1-prime-factor*

rsakpv1-prime-factor is the key-pair validation method corresponding to *rsakpg1-prime-factor* (see 6.3.1.2).

Function call: *rsakpv1-prime-factor* (s, $nBits$, e_{fixed}, (n_{pub}, e_{pub}), (p, q, d))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the prime-factor format: (p, q, d).

The steps are the same as in *rsakpv1-basic* except that in processing:

A. Step 3 is replaced by the following:

   3. Check the modulus:

      a. If $n_{pub} \neq pq$, output an indication of an invalid key pair, and exit without further processing.

      b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

B. Step 4 (prime factor recovery) is omitted.

A routine that implements this validation function **shall** destroy any local copies of p, q, and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.2.3 *rsakpv1-crt*

rsakpv1-crt is the key-pair validation method corresponding to *rsakpg1-crt*.

Function call: *rsakpv1-crt* (s, $nBits$, e_{fixed}, (n_{pub}, e_{pub}), (n_{priv}, e_{priv}, d, p, q, dP, dQ, $qInv$))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the Chinese Remainder Theorem format: (n_{priv}, e_{priv}, d, p, q, dP, dQ, $qInv$).

The steps are the same as in *rsakpv1-basic* except that in processing:

A. Step 2 is replaced by the following:

   2. Compare the public exponents:

      If ($e_{pub} \neq e_{fixed}$) or ($e_{pub} \neq e_{priv}$), output an indication of an invalid key pair, and exit without further processing.

B. Step 3 is replaced by

   3. Check the modulus:

      a. If $n_{pub} \neq pq$, or $n_{pub} \neq n_{priv}$, output an indication of an invalid key pair, and exit without further processing.

      b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

C. Step 4 (prime factor recovery) is omitted,

D. Step 7 is replaced by the following:

   7. Check the CRT components: Check that the components dP, dQ and $qInv$ satisfy

      a. $1 < dP < (p - 1)$.

b. $1 < dQ < (q - 1)$.

c. $1 < qInv < p$.

d. $1 = (dP \cdot e_{fixed}) \bmod (p - 1)$.

e. $1 = (dQ \cdot e_{fixed}) \bmod (q - 1)$.

f. $1 = (qInv \cdot q) \bmod p$.

If any of the criteria in Section 6.2.1 are not met, output an indication of an invalid key pair, and exit without further processing.

8. Output an indication that the key pair is valid.

A routine that implements this validation function **shall** destroy any local copies of p, q, d, dP, dQ, and $qInv$, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.3 RSAKPV2 Family: RSA Key-Pair Validation with a Random Exponent

The RSAKPV2 family of key-pair validation methods corresponds to RSAKPG2 family of key-pair generation methods (see Section 6.3.2).

6.4.1.3.1 *rsakpv2-basic*

rsakpv2-basic is the validation method corresponding to *rsakpg2-basic* (see Section 6.3.2.1).

Function call: *rsapkv2-basic* (s, $nBits$, $eBits$, (n_{pub}, e), (n_{priv}, d))

The method is the same as the *rsapkv1-basic* method in Section 6.4.1.2.1, except that:

A. The e_{fixed} input parameter becomes $eBits$, which is the expected length in bits of the public exponent, an integer such that $17 \leq eBits \leq 256$.

B. Step 1c is replaced by:

   c. If ($eBits < 17$) or ($eBits > 256$), output an indication that the exponent is out of range, and exit without further processing.

C. Step 2 is replaced by:

   2. Check the public exponent.

      If the public exponent e_{pub} is not odd, or if the length in bits of the public exponent e_{pub} is not $eBits$, output an indication of an invalid key pair, and exit without further processing.

A routine that implements this validation function **shall** destroy any local copies of p, q, and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.3.2 *rsakpv2-prime-factor*

rsakpv2-prime-factor is the key-pair validation method corresponding to *rsakpg2-prime-factor* key-pair generation method (see Section 6.3.2.2).

Function call: *rsakpv2-prime-factor* (*s*, *nBits*, *eBits*, (n_{pub}, e_{pub}), (*p*, *q*, *d*))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the prime factor format: (*p*, *q*, *d*).

The steps are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that:

A. The e_{fixed} input parameter becomes *eBits*, which is the expected length in bits of the public exponent, an integer such that $17 \leq eBits \leq 256$.

B. Step 1c is replaced by:

   c. If ($eBits < 17$) or ($eBits > 256$), output an indication that the exponent is out of range, and exit without further processing.

C. Step 2 is replaced by:

   2. Check the public exponent.

      If the public exponent e_{pub} is not odd, or if the length in bits of the public exponent e_{pub} is not *eBits*, output an indication of an invalid key pair, and exit without further processing.

D. Step 3 is replaced by the following:

   3. Check the modulus:

      a. If $n_{pub} \neq pq$, output an indication of an invalid key pair, and exit without further processing.

      b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

E. Step 4 (prime factor recovery) is omitted.

A routine that implements this validation function **shall** destroy any local copies of *p*, *q*, and *d*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.3.3 *rsakpv2-crt*

rsakpv2-crt is the key-pair validation method corresponding to *rsakpg2-crt* key-pair generation method (see Section 6.3.1.3).

Function call: *rsakpv2-crt* (*s*, *nBits*, *eBits*, (n_{pub}, e_{pub}), (n_{priv}, e_{priv}, *d*, *p*, *q*, *dP*, *dQ*, *qInv*))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the Chinese Remainder Theorem format: (n_{priv}, e_{priv}, *d*, *p*, *q*, *dP*, *dQ*, *qInv*).

The steps are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that:

A. The e_{fixed} input parameter becomes *eBits*, which is the expected length in bits of the public exponent, an integer such that $17 \leq eBits \leq 256$.

B. Step 1c is replaced by:

    c. If ($eBits < 17$) or ($eBits > 256$), output an indication that the exponent is out of range, and exit without further processing.

C. Step 2 is replaced by the following:

    2. Compare the public exponents:

       If ($e_{pub} \neq e_{priv}$) or (e_{pub} is not odd) or (length in bits of e_{pub} is not *eBits*), output an indication of an invalid key pair, and exit without further processing.

D. Step 3 is replaced by

    3. Check the modulus:

        a. If ($n_{pub} \neq pq$) or ($n_{pub} \neq n_{priv}$) output an indication of an invalid key pair, and exit without further processing.

        b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

E. Step 4 (prime factor recovery) is omitted,

F. Step 7 is replaced by the following:

    7. Check the CRT components: Check that the components *dP*, *dQ* and *qInv* satisfy

        a) $1 < dP < (p - 1)$.

        b) $1 < dQ < (q - 1)$.

        c) $1 < qInv < p$.

        d) $1 = (dP \cdot e_{pub}) \bmod (p - 1)$.

        e) $1 = (dQ \cdot e_{pub}) \bmod (q - 1)$.

        f) $1 = (qInv \cdot q) \bmod p$.

       If any of the criteria in Section 6.2.1 are not met, output an indication of an invalid key pair, and exit without further processing.

    8. Output an indication that the key pair is valid.

A routine that implements this validation function **shall** destroy any local copies of *p*, *q*, *d*, *dP*, *dQ*, and *qInv*, as well as any other locally stored values used or produced during its execution.

Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.4 RSA Key-Pair Validation (Intended Exponent-Creation Method Unknown)

Public-key validation may be performed when the intended fixed value or intended length of the public exponent is unknown by the entity performing the validation (i.e., the entity is unaware of whether the key pair was generated as specified in Section 6.3.1 or as specified in Section 6.3.2). The entity performing the validation (i.e., the key-pair owner or a TTP trusted by the owner) knows only the key pair to be validated and its representation (i.e., either *basic*, *prime factor* or *crt*).

6.4.1.4.1 *basic-pkv*

In this format, the private key is represented as (n, d).

Function call: *basic_pkv* (s, *nBits*, (n_{pub}, e_{pub}), (n_{priv}, d))

The method is the same as the *rsapkv1-basic* method in Section 6.4.1.2.1, except that:

A. A value for e_{fixed} is not available as an input parameter.

B. Step 1.c is replaced by:

   If e_{pub} is not an odd integer such that $65\,537 \leq e_{pub} < 2^{256}$, output an indication that the fixed exponent is out of range, and exit without further processing.

C. Step 2 is not performed.

A routine that implements this validation function **shall** destroy any local copies of p, q, and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.4.2 *prime-factor-pkv*

In this format, the private key is represented as (p, q, d).

Function call: *prime-factor_pkv* (s, *nBits*, (n_{pub}, e_{pub}), (p, q, d))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the prime factor format: (p, q, d).

The steps are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that:

A. A value for e_{fixed} is not available as an input parameter.

B. Step 1.c is replaced by:

   If e_{pub} is not an odd integer such that $65\,537 \leq e_{pub} < 2^{256}$, output an indication that the fixed exponent is out of range, and exit without further processing.

C. Step 2 is not performed.

D. Step 3 is replaced by the following:

3. Check the modulus:

    a. If $n_{pub} \neq pq$, output an indication of an invalid key pair, and exit without further processing.

    b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

E. Step 4 (prime factor recovery) is omitted.

A routine that implements this validation function **shall** destroy any local copies of p, q, and d, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.4.3 *crt_pkv*

In this format, the private key is represented as (n, e, d, p, q, dP. dQ, $qInv$).

Function call: *crt_pkv*(s, *nBits*, (n_{pub}, e_{pub}), (n_{priv}, e_{priv}, d, p, q, dP, dQ, $qInv$))

The inputs, outputs and errors are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that the private key is in the Chinese Remainder Theorem (CRT) format: (n_{priv}, e_{priv}, d, p, q, dP, dQ, $qInv$).

The steps are the same as in *rsakpv1-basic* (see Section 6.4.1.2.1), except that:

A. A value for e_{fixed} is not available as an input parameter.

B. Step 1c is replaced by:

    If e_{pub} is not an odd integer such that $65\,537 \leq e_{pub} < 2^{256}$, output an indication that the fixed exponent is out of range, and exit without further processing.

C. Step 2 is not performed.

D. Step 3 is replaced by

    3. Check the modulus:

        a. If ($n_{pub} \neq pq$) or ($n_{pub} \neq n_{priv}$), output an indication of an invalid key pair, and exit without further processing.

        b. If the length in bits of the modulus n_{pub} is not *nBits*, output an indication of an invalid key pair, and exit without further processing.

E. Step 4 (prime factor recovery) is omitted,

F. Step 7 is replaced by the following:

    7. Check the CRT components: Check that the components dP, dQ and $qInv$ satisfy

        a) $1 < dP < (p - 1)$.

        b) $1 < dQ < (q - 1)$.

c) $1 < qInv < p$.

d) $1 = (dP \cdot e_{pub}) \bmod (p - 1)$.

e) $1 = (dQ \cdot e_{pub}) \bmod (q - 1)$.

f) $1 = (qInv \cdot q) \bmod p$.

If any of the criteria in Section 6.2.1 are not met, output an indication of an invalid key pair, and exit without further processing.

8. Output an indication that the key pair is valid.

A routine that implements this validation function **shall** destroy any local copies of *p*, *q*, *dP*, *dQ*, and *qInv*, as well as any other locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally).

6.4.1.5 Owner Assurance of Private-Key Possession

An owner's initial assurance of possession of his private key is obtained when assurance of key-pair validity is obtained (see Section 6.4.1.1); assurance of key-pair validity is required prior to the owner's use of a key pair for key establishment. As time passes, an owner could lose possession of the private key of a key pair. For this reason, renewing the assurance of possession may be appropriate for some applications (i.e., assurance of possession can be refreshed). A discussion of the effect of time on the assurance of private-key possession is provided in SP 800-89 [SP 800-89].

Renewed assurance that the owner continues to possess the correct associated private key **shall** be obtained in one or more of the following ways:

1. The key-pair owner renews assurance of key-pair validity – The owner obtains assurance of renewed key-pair validity (see Section 6.4.1.1), thereby also obtaining renewed assurance of private key possession.

2. The key-pair owner receives renewed assurance via key confirmation – The owner employs the key pair to successfully engage another party in a key-agreement transaction using a scheme from the KAS2 family that incorporates key confirmation. The key confirmation **shall** be performed with the owner as key-confirmation recipient in order to obtain assurance that the private key functions correctly.

   - The KAS2-Party_V-confirmation scheme in Section 8.3.3.2 can be used to provide assurance to party U that party V possesses the private key associated with party V's public key ($PubKey_V$) used during the key-agreement transaction.

   - The KAS2-Party_U-confirmation scheme in Section 8.3.3.3 can be used to provide assurance to party V that party U possesses the private key associated with party U's public key ($PubKey_U$) used during the key-agreement transaction.

   - The KAS2-bilateral-confirmation scheme in Section 8.3.3.4 can be used to provide assurance to each party that the other party possesses the private key associated with that party's public key that is used during the key-agreement transaction.

3. The owner receives assurance via an encrypted certificate - The key-pair owner uses the private key while engaging in a key-establishment transaction with a Certificate Authority (trusted by the owner) using a scheme in this Recommendation, after providing the CA with the corresponding public key. As part of this transaction, the CA generates a (new) certificate containing the owner's public key and encrypts that certificate using (some portion of) the symmetric keying material that has been established. Only the encrypted form of the certificate is provided to the owner. By successfully decrypting the certificate and verifying the CA's signature, the owner obtains assurance of possession of the correct private key (at the time of the key-establishment transaction).

The key-pair owner (or agents trusted to act on the owner's behalf) **should** determine that the method used for obtaining renewed assurance of the owner's possession of the correct private key is sufficient and appropriate to meet the security requirements of the owner's intended application(s).

6.4.2 Assurances Required by a Public-Key Recipient

In this Recommendation, the recipient of a public key is an entity that does not (and should not) have access to the corresponding private key of the other party. The recipient of a candidate public key **shall** have:

1. Assurance of the arithmetic validity of the other party's public key before using it in a key-establishment transaction with its claimed owner, and
2. Assurance that the claimed public-key owner (i.e., the other party) actually possesses the private key corresponding to that public key.

6.4.2.1 Obtaining Assurance of Public-Key Validity for a Received Public Key

The recipient **shall** obtain assurance of public-key validity using one or more of the following methods:

1. Recipient Partial Public-Key Validation - The recipient performs a successful partial public-key validation (see Section 6.4.2.2).
2. TTP Partial Public-Key Validation – The recipient receives assurance that a trusted third party (trusted by the recipient) has performed a successful partial public-key validation (see Section 6.4.2.2).
3. TTP Key-Pair Validation – The recipient receives assurance that a trusted third party (trusted by the recipient and the owner) has performed key-pair validation in accordance with Section 6.4.1.1 (step 3.b).

Note that the use of a TTP to perform key-pair validation (method 3) implies that both the owner and any recipient of the public key trust that the TTP will not use the owner's private key to masquerade as the owner or otherwise compromise the key-establishment transaction.

6.4.2.2 Partial Public-Key Validation for RSA

Partial public-key validation for RSA consists of conducting plausibility tests. These tests determine whether the public modulus and public exponent are plausible, not necessarily whether they are completely valid, i.e., they may not conform to all RSA key-generation requirements as specified in this Recommendation. Plausibility tests can detect unintentional errors with a reasonable probability. Note that full RSA public-key validation is not specified in this Recommendation, as it is an area of research. Therefore, if an application requires assurance of full public-key validation, then another **approved** key-establishment method **shall** be used.

Plausibility tests **shall** include the tests specified in SP 800-89 [SP 800-89], Section 5.3.3, with the caveat that the length of the modulus **shall** be a length that is specified in this Recommendation.

6.4.2.3 Recipient Assurances of an Owner's Possession of a Private Key

When two parties engage in a key-establishment transaction, there is (at least) an implicit claim of ownership made whenever a public key is provided on behalf of a particular party. That party is considered to be a *claimed* owner of the corresponding key pair – as opposed to being a *true* owner – until adequate assurance can be provided that the party is actually the one authorized to use the private key. The claimed owner can provide such assurance by demonstrating its knowledge of that private key.

The recipient of another party's public key **shall** obtain an initial assurance that the other party (i.e., the claimed owner of the public key) actually possesses the associated private key, either prior to or concurrently with performing a key-establishment transaction with that other party. Obtaining this assurance is addressed in Sections 6.4.2.3.1 and 6.4.2.3.2. As time passes, renewing the assurance of possession may be appropriate for some applications; assurance of possession can be renewed as specified in Section 6.4.2.3.2. A discussion of the effect of time on the assurance of private-key possession is provided in SP 800-89 [SP 800-89].

As part of the proper implementation of this Recommendation, system users and/or agents trusted to act on their behalf **should** determine which of the methods for obtaining assurance of possession meet their security requirements. The application tasked with performing key establishment on behalf of a party **should** determine whether or not to proceed with a key-establishment transaction, based upon the perceived adequacy of the method(s) used. Such knowledge may be explicitly provided to the application in some manner, or may be implicitly provided by the operation of the application itself.

If a binding authority is the public-key recipient: At the time of binding an owner's identifier to his public key, the binding authority (i.e., a trusted third party, such as a CA) **shall** obtain assurance that the owner is in possession of the correct private key. This assurance **shall** either be obtained using one of the methods specified in Section 6.4.2.3.2 (e.g., with the binding authority acting as the public-key recipient) or by using an **approved** alternative (see SP 800-57, Part 1, Sections 5.2 and 8.1.5.1.1.2).

Recipients not acting in the role of a binding authority: The recipients **shall** obtain this assurance either through a trusted third party (see Section 6.4.2.3.1) or directly from the owner (i.e., the other party) (see Section 6.4.2.3.2) before using the derived keying material for purposes beyond those required during the key-establishment transaction itself. If the recipient chooses to obtain this assurance directly from the other party (i.e., the claimed owner of that public key), then to comply with this Recommendation, the recipient **shall** use one of the methods specified in Section 6.4.2.3.2.

Note that the requirement that assurance of possession be obtained before using the established keying material for purposes *beyond* those of the key-establishment transaction itself does not prohibit the parties to a key-establishment transaction from using a portion of the derived or transported keying material *during* the key-establishment transaction for purposes required by that key-establishment scheme. For example, in a transaction involving a key-agreement scheme that incorporates key confirmation, the parties establish a (purported) shared secret, derive keying material, and — as part of that same transaction — use a portion of the derived keying material as the MAC key in their key-confirmation computations.

6.4.2.3.1 Recipient Obtains Assurance from a Trusted Third Party

The recipient of a public key may receive assurance that its owner (i.e., the other party in the key-establishment transaction) is in possession of the correct private key from a trusted third party (trusted by the recipient), either before or during a key-establishment transaction that makes use of that public key. The methods used by a third party trusted by the recipient to obtain that assurance are beyond the scope of this Recommendation (see however, the discussion in Section 6.4.2.3.2 and Section 8.1.5.1.1.2 of SP 800-57 [SP 800-57]).

The recipient of a public key (or agents trusted to act on behalf of the recipient) **should** know the method(s) used by the third party, in order to determine that the assurance obtained on behalf of the recipient is sufficient and appropriate to meet the security requirements of the recipient's intended application(s).

6.4.2.3.2 Recipient Obtains Assurance Directly from the Claimed Owner (i.e., the Other Party)

The recipient of a public key can directly obtain assurance of the claimed owner's current possession of the corresponding private key by successfully completing a key-establishment transaction that explicitly incorporates key confirmation, with the claimed owner serving as the key-confirmation provider. Note that the recipient of the public key in question will also be the key-confirmation recipient. Also note that this use of key confirmation is an additional benefit beyond its use to confirm that two parties possess the same keying material.

There are a number of key-establishment schemes specified in this Recommendation that can be used. In order to claim conformance with this Recommendation, the key-establishment transaction during which the recipient of a public key seeks to obtain assurance of its owner's current possession of the corresponding private key **shall** employ one of the following **approved** key-establishment schemes:

1. The KAS1-Party_V-confirmation scheme in Section 8.2.3.2 can be used to provide assurance to party U that party V possesses the private key corresponding to party V's public key ($PubKey_V$) that is used during the key-agreement transaction.
2. The KAS2-Party_V-confirmation scheme in Section 8.3.3.2 can be used to provide assurance to party U that party V possesses the private key corresponding to party V's public key ($PubKey_V$) that is used during the key-agreement transaction.
3. The KAS2-Party_U-confirmation scheme in Section 8.3.3.3 can be used to provide assurance to party V that party U possesses the private key corresponding to party U's public key ($PubKey_U$) that is used during the key-agreement transaction.

4. The KAS2-bilateral-confirmation scheme in Section 8.3.3.4 can be used to provide assurance to each party that the other party possesses the private key corresponding to the other party's public key that is used during the key-agreement transaction.
5. The KTS-OAEP-Party_V-confirmation scheme in Section 9.2.4.2 can be used to provide assurance to party U (the key-transport sender) that party V (the key-transport receiver) possesses the private key corresponding to party V's public key ($PubKey_V$) that is used during the key-transport transaction.
6. The KTS-KEM-KWS-Party_V-confirmation scheme in Section 9.3.4.2 can be used to provide assurance to party U (the key-transport sender) that party V (the key-transport receiver) possesses the private key corresponding to party V's public-key ($PubKey_V$) that is used during the key-transport transaction.

The recipient of a public key (or agents trusted to act on the recipient's behalf) **shall** determine whether or not using one of the key-establishment schemes in this Recommendation to obtain assurance of possession through key confirmation is sufficient and appropriate to meet the security requirements of the recipient's intended application(s). Other **approved** methods (e.g., see Section 5.4.4 of SP 800-57-Part 1 [SP 800-57]) of directly obtaining this assurance of possession from the owner are also allowed. If obtaining assurance of possession directly from the owner is not acceptable, then assurance of possession **shall** be obtained indirectly as discussed in Section 6.4.2.3.1.

Successful key confirmation (performed in the context described in this Recommendation) demonstrates that the correct private key has been used in the key-confirmation provider's calculations, and thus also provides assurance that the claimed owner is the true owner.

The assurance of possession obtained via the key-confirmation schemes identified above may be useful even when the recipient has previously obtained independent assurance that the claimed owner of a public key is indeed its true owner. This may be appropriate in situations where the recipient desires renewed assurance that the owner possesses the correct private key (and that the owner is still able to use it correctly), including situations where there is no access to a trusted party who can provide renewed assurance of the owner's continued possession of the private key.

7 Primitives and Operations

7.1 Encryption and Decryption Primitives

RSAEP and RSADP are the basic encryption and decryption primitives from the RSA cryptosystem [RSA 1978], specified in [PKCS 1]. RSAEP produces ciphertext from keying material using a public key; RSADP recovers the keying material from the ciphertext using the corresponding private key. The primitives assume that the RSA public key is valid.

7.1.1 RSAEP

RSAEP produces ciphertext using an RSA public key.

Function call: RSAEP((n, e), k)

Input:

1. (n, e): the RSA public key.
2. k: an integer such that $1 < k < n – 1$.

Assumption: The RSA public key is valid (see Section 6.4).

Process:

1. If k does not satisfy $1 < k < n – 1$, output an indication that k is out of range, and exit without further processing.
2. Let $c = (k)^e \bmod n$.
3. Output c.

Output:

c: the ciphertext, an integer such that $1 < c < n – 1$, or an error indicator.

A routine that implements this primitive **shall** destroy any local copies of the input k, as well as any other potentially sensitive locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of c).

7.1.2 RSADP

RSADP is the basic decryption primitive. It recovers plaintext from ciphertext using an RSA private key.

Function call: RSADP((n, d), c)

Input:

1. (n, d): the RSA private key.
2. c: the ciphertext, such that $1 < c < n – 1$.

Process:

1. If the ciphertext c does not satisfy $1 < c < n – 1$, output an indication that the ciphertext is out of range, and exit without further processing.
2. Let $k = c^d \bmod n$.
3. Output k.

Output:

k: an integer such that $1 < k < n – 1$, or an error indicator.

Note:

Care **should** be taken to ensure that an implementation of RSADP does not reveal even partial information about the value of k. An opponent who can reliably obtain particular bits of *k* for sufficiently many chosen ciphertext values may be able to obtain the full decryption of an arbitrary ciphertext by applying the bit-security results of Håstad and Näslund [HN 1998].

A routine that implements this primitive **shall** destroy any local copies of the input *d*, as well as any other potentially sensitive locally stored values used or produced during its execution (such as any locally stored portions of *k*). Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of *k*). Note that the requirement for destruction includes any locally stored portions of the output.

7.2 Encryption and Decryption Operations

7.2.1 RSA Secret-Value Encapsulation (RSASVE)

The RSASVE generate operation is used by one party in a key-establishment transaction to generate and encrypt a secret value to produce ciphertext using the public key-establishment key of the other party. When this ciphertext is received by that other party, and the secret value is recovered (using the RSASVE recover operation and the corresponding private key-establishment key), the secret value is then considered to be a shared secret. Secret-value encapsulation employs a Random Bit Generator (RBG) to generate the secret value.

The RSASVE generate and recovery operations specified in Sections 7.2.1.2 and 7.2.1.3, respectively, are based on the RSAEP and RSADP primitives (see Section 7.1). These operations are used by the KAS1 and KAS2 key-agreement families (see Sections 8.2 and 8.3), and by the RSA-KEM KWS key-transport family (see Sections 9.3 and 7.2.3).

7.2.1.1 RSASVE Components

RSASVE uses the following components:

1.	RBG:	An **approved** random bit generator (see Section 5.3).
2.	RSAEP:	RSA Encryption Primitive (see Section 7.1.1).
3.	RSADP:	RSA Decryption Primitive (see Section 7.1.2).

7.2.1.2 RSASVE Generate Operation

RSASVE.GENERATE generates a secret value and corresponding ciphertext using an RSA public key.

Function call: **RSASVE.GENERATE**((*n*, *e*))

Input:

(*n*, *e*): an RSA public key.

Assumptions: The RSA public key is valid.

Process:

1. Compute the value of *nLen* as the length in bytes of the modulus *n*.

2. Generation:

a. Using the RBG (see Section 5.3), generate an *nLen* byte string, *Z*.

b. Convert *Z* to an integer *z* (See Appendix B.2):

z = BS2I(Z, *nLen*).

c. If *z* does not satisfy $1 < z < n - 1$, then go to step 2a.

3. RSA encryption:

a. Apply the RSAEP encryption primitive (see Section 7.1.1) to the integer *z* using the public key (*n*, *e*) to produce an integer ciphertext *c*:

c = RSAEP((*n*, *e*), *z*).

b. Convert the ciphertext *c* to a ciphertext byte string *C* of length *nLen* bytes (see Appendix B.1):

C = I2BS(*c*, *nLen*).

4. Output the string *Z* as the secret value, and the ciphertext *C*.

Output:

Z: the secret value to be shared (a byte string of length *nLen* bytes), and C: the ciphertext (a byte string of length *nLen* bytes).

A routine that implements this operation **shall** destroy any locally stored portions of *Z* and *z*, as well as any other potentially sensitive locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of *Z* and *C*). Note that the requirement for destruction includes any locally stored portions of the secret value *Z* included in the output.

7.2.1.3 RSASVE Recovery Operation

RSASVE.RECOVER recovers a secret value from ciphertext using an RSA private key. Once recovered, the secret value is considered to be a shared secret.

Function call: RSASVE.RECOVER((*n*, *d*), *C*)

Input:

1. (*n*, *d*): an RSA private key.

2. *C*: the ciphertext; a byte string of length *nLen* bytes.

Assumptions: The RSA private key is part of a valid key pair.

Process:

1. Compute the value of *nLen* as the length in bytes of the modulus *n*.

2. Length checking:

   If the length of the ciphertext C is not *nLen* bytes, output an indication of a decryption error, and exit without further processing.

3. RSA decryption:

   a. Convert the ciphertext C to an integer ciphertext c (see Appendix B.2):

   $$c = \text{BS2I}(C).$$

   b. Apply the RSADP decryption primitive (see Section 7.1.2) to the ciphertext c using the private key (n, d) to produce an integer z:

   $$z = \text{RSADP}((n, d), c)\ .$$

   c. If RSADP indicates that the ciphertext is out of range, output an indication of a decryption error, and exit without further processing.

   d. Convert the integer z to a byte string Z of length *nLen* bytes (see Appendix B.1):

   $$Z = \text{I2BS}(z, nLen).$$

4. Output the string Z as the secret value (i.e., the shared secret), or an error indicator.

Output:

Z: the secret value/shared secret (a byte string of length *nLen* bytes), or an error indicator.

Note:

Care **should** be taken to ensure that an implementation does not reveal information about the encapsulated secret value (i.e., the value of the integer z or its byte string equivalent Z). For instance, the observable behavior of the I2BS routine **should not** reveal even partial information about the byte string Z. An opponent who can reliably obtain particular bits of Z for sufficiently many chosen ciphertext values may be able to obtain the full decryption of an arbitrary RSA-encrypted value by applying the bit-security results of Håstad and Näslund [HN 1998].

A routine that implements this operation **shall** destroy any local copies of the input d, any locally stored portions of Z and z, and any other potentially sensitive locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of Z). Note that the requirement for destruction includes any locally stored portions of the output.

7.2.2 RSA with Optimal Asymmetric Encryption Padding (RSA-OAEP)

RSA-OAEP consists of asymmetric encryption and decryption operations that are based on an **approved** hash function, an **approved** random bit generator, a mask-generation function, and the

RSAEP and RSADP primitives. These operations are used by the KTS-OAEP key-transport schemes (see Section 9.2).

In the RSA-OAEP encryption operation, a data block is constructed by the sender (party U) from the keying material to be transported and the hash of additional input (see Section 9.1) that is shared by party U and the intended receiving party (party V). A random byte string is generated, after which both the random byte string and the data block are masked in a way that binds their values. The masked values are used to form the plaintext that is input to the RSAEP primitive, along with the public key-establishment key of party V. The resulting RSAEP output further binds the random byte string, the keying material and the hash of the additional data in the ciphertext that is sent to party V.

In the RSA-OAEP decryption operation, the ciphertext and the receiving party's (i.e., party V's) private key-establishment key are input to the RSADP primitive, recovering the masked values as output. The mask-generation function is then used to reconstruct and remove the masks that obscure the random byte string and the data block. After removing the masks, party V can examine the format of the recovered data and compare its own computation of the hash of the additional data to the hash value contained in the unmasked data block, thus obtaining some measure of assurance of the integrity of the recovered data – including the transported keying material.

RSA-OAEP can process up to $nLen – 2HLen – 2$ bytes of keying material, where *nLen* is the length of the recipient's RSA modulus in bytes (i.e., 256 or 384, in this Recommendation), and *HLen* is the length (in bytes) of the values output by the underlying hash function.

7.2.2.1 RSA-OAEP Components

RSA-OAEP uses the following components:

1. H: An **approved** hash function (see Section 5.1). *HLen* is used to denote the length (in bytes) of the hash function output.
2. MGF: The mask-generation function (see Section 7.2.2.2).
3. RBG: An **approved** random bit generator (see Section 5.3).
4. RSAEP: RSA Encryption Primitive (see Section 7.1.1).
5. RSADP: RSA Decryption Primitive (see Section 7.1.2).

7.2.2.2 The Mask Generation Function (MGF)

MGF is a mask-generation function based on an **approved** hash function (see Section 5.1). The purpose of the MGF is to generate a string of bits that may be used to "mask" other bit strings. The MGF is used by the RSA-OAEP-based schemes specified in Section 9.2.

Let *hash* be an **approved** hash function, and let *hashLen* denote the length of the hash function output in bytes.

For the purposes of this Recommendation, the MGF **shall not** be invoked more than once by each party during a given transaction using a given MGF seed (i.e., a mask **shall** be derived only once by each party from a given MGF seed).

Function call: MGF(*mgfSeed*, *maskLen*)

Auxiliary Function:

hash: an **approved** hash function (see Section 5.1).

Implementation-Dependent Parameters:

1. *hashLen*: an integer that indicates the length (in bytes) of the output block of the auxiliary hash function, *hash*.
2. *max_hash_inputLen*: an integer that indicates the maximum-permitted length (in bytes) of the bit string, *x*, that is used as input to the auxiliary hash function, *hash*.

Input:

1. *mgfSeed*: a byte string from which the mask is generated.
2. *maskLen*: the intended length of the mask (in bytes).

Process:

```
1. If $maskLen > 2^{32}\ hashLen$, output an error indicator, and exit from this process without
   performing the remaining actions.
2. If $mgfSeed$ is more than $max\_hash\_inputLen$ bytes long, then output an error indicator,
   and exit this process without performing the remaining actions.
3. Set $T=\{\}$, the empty string.
4. For $counter$ from 0 to $\lceil maskLen / hashLen \rceil - 1$, do the following:
       a) Let $D = \text{I2BS}(counter, 4)$   (see Appendix B.1).
       b) Let $T = T \| hash(mgfSeed \| D)$.
5. Output the leftmost $maskLen$ bytes of $T$ as the byte string $mask$.
```

Output:

The byte string *mask* (of length *maskLen* bytes), or an error indicator.

A routine that implements this function **shall** destroy any local copies of the input *mgfSeed*, any locally stored portions of *mask* (e.g., any portion of *T*), and any other potentially sensitive locally stored values used or produced during its execution. Their destruction **shall** occur prior to or during any exit from the routine (whether exiting early, because of an error, or exiting normally, with the output of *mask*). Note that the requirement for destruction includes any locally stored portions of the output.

7.2.2.3 RSA-OAEP Encryption Operation

The RSA-OAEP.Encrypt operation produces a ciphertext from keying material and additional input using an RSA public key, as shown in Figure 4. See Section 9.1 for more information on the additional input. Let *HLen* be the length in bytes of the output of hash function *H*.

Function call: RSA-OAEP.ENCRYPT((*n*, *e*), *K*, *A*)

Input*:*

1. (*n*, *e*): the receiver's RSA public key.
2. *K*: the keying material; a byte string of length at most *nLen* – 2*HLen* – 2, where *nLen* = 256 or 384 in this Recommendation.
3. *A*: additional input; a byte string (may be the empty string) to be cryptographically bound to the keying material (see Section 9.1).

Assumptions: The RSA public key is valid.

Process*:*

1. *nLen* = the length of *n* in bytes.
2. Length checking:
    a. *KLen* = the length of *K* in bytes.
    b. If *KLen* > *nLen* – 2*HLen* – 2, then output an indication that the keying material is too long, and exit without further processing.
3. OAEP encoding:
    a. Apply the selected hash function to compute:

    $$HA = \mathrm{H}(A).$$

    HA is a byte string of length *HLen*. If *A* is an empty string, then *HA* is the hash value for the empty string.

    b. Construct a byte string *PS* consisting of *nLen* – *KLen* – 2*HLen* – 2 zero bytes. The length of *PS* may be zero.

    c. Concatenate *HA, PS,* a single byte with a hexadecimal value of 01, and the keying material *K* to form data *DB* of length *nLen* – *HLen* – 1 bytes as follows:

    $$DB = HA \,\|\, PS \,\|\, 01 \,\|\, K,$$

    where 01 represents the bit string 00000001.

d. Using the RBG (see Section 5.3), generate a random byte string *mgfSeed* of length *HLen* bytes.

e. Apply the mask-generation function in Section 7.2.2.2 to compute:

$$dbMask = \mathrm{MGF}(mgfSeed, nLen - HLen - 1).$$

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

g. Apply the mask-generation function in Section 7.2.2.2 to compute:

$$mgfSeedMask = \mathrm{MGF}(maskedDB, HLen).$$

h. Let $maskedMGFSeed = mgfSeed \oplus mgfSeedMask$.

i. Concatenate a single byte with hexadecimal value 00, *maskedMGFSeed*, and *maskedDB* to form an encoded message *EM* of length *nLen* bytes as follows:

$$EM = 00 \,\|\, maskedMGFSeed \,\|\, maskedDB$$

where 00 represents the bit string 00000000.

4. RSA encryption:

a. Convert the encoded message *EM* to an integer *em* (see Appendix B.2):

$$em = \mathrm{BS2I}(EM).$$

b. Apply the RSAEP encryption primitive (see Section 7.1.1) to the integer *em* using the public key (*n*, *e*) to produce a ciphertext integer *c*:

$$c = \mathrm{RSAEP}((n, e), em).$$

c. Convert the ciphertext integer *c* to a ciphertext byte string *C* of length *nLen* bytes (see Appendix B.1):

$$C = \mathrm{I2BS}(c, nLen).$$

5. Zeroize all intermediate values and output the ciphertext *C*.

Output: C: the ciphertext (a byte string of length *nLen* bytes), or an error indicator.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (e.g., *K* and any sensitive portions of *A*), as well as any other potentially sensitive locally stored values used or produced during its execution (including *HA*, *DB*, *mfgSeed*, *dbMask*, *maskedDB*, *mgfSeedMask*, *maskedMGFSeed*, *EM*, and *em*). Their destruction **shall** occur prior to or during any exit from the routine – whether exiting early, because of an error, or exiting normally, with the output of *C*.

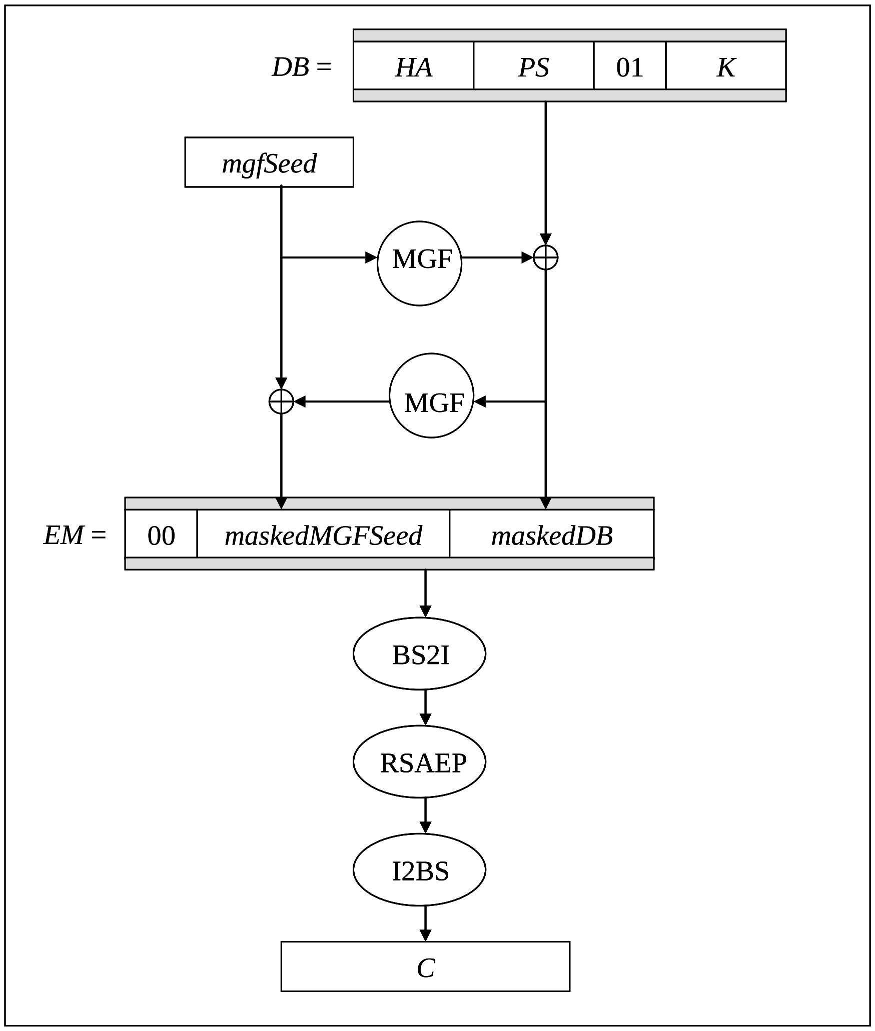

Figure 4: RSA-OAEP Encryption Operation

7.2.2.4 RSA-OAEP Decryption Operation

RSA-OAEP.DECRYPT recovers keying material from a ciphertext and additional input using an RSA private key as shown in Figure 5. Let *HLen* be the length in bytes of the output of hash function *H*.

Function call: RSA-OAEP.DECRYPT((n, d), C, A)

Input:

1. (n, d): the receiver's RSA private key.
2. C: the ciphertext; a byte string.
3. A: additional input; a byte string (may be the empty string) whose cryptographic binding to the keying material is to be verified (see Section 9.1).

Assumptions: The RSA private key is valid.

Process:

1. Initializations:

   a. *nLen* = the length of *n* in bytes. For this Recommendation, *nLen* = 256 or 384.

   b. *DecryptErrorFlag* = *False*.

2. Check for erroneous input:

   a. If the length of the ciphertext *C* is not *nLen* bytes, output an indication of erroneous input, and exit without further processing.

   b. Convert the ciphertext byte string *C* to a ciphertext integer *c* (see Section B.2):

   $$c = \text{BS2I}(C)$$

   c. If the ciphertext integer *c* is not such that $1 < c < n - 1$, output an indication of erroneous input, and exit without further processing.

3. RSA decryption:

   a. Apply the RSADP decryption primitive (see Section 7.1.2) to the ciphertext integer *c* using the private key (*n*, *d*) to produce an integer *em*:

   $$em = \text{RSADP}((n, d), c).$$

   b. Convert the integer e*m* to an encoded message *EM*, a byte string of length *nLen* bytes (see Appendix B.1):

   $$EM = \text{I2BS}(em, nLen).$$

4. OAEP decoding:

   a. Apply the selected hash function (see Section 5.1) to compute:

   $$HA = \text{Hash}(A).$$

   HA is a byte string of length *HLen* bytes.

   b. Separate the encoded message *EM* into a single byte *Y*, a byte string *maskedMGFSeed′* of length *HLen* bytes, and a byte string *maskedDB′* of length $nLen - HLen - 1$ bytes as follows:

   $$EM = Y \,\|\, maskedMGFSeed' \,\|\, maskedDB'.$$

   c. Apply the mask-generation function specified in Section 7.2.2.2 to compute:

$$mgfSeedMask' = \mathrm{MGF}(maskedDB', HLen).$$

d. Let $mgfSeed' = maskedMGFSeed' \oplus mgfSeedMask'$.

e. Apply the mask-generation function specified in Section 7.2.2.2 to compute:

$$dbMask' = \mathrm{MGF}(mgfSeed', nLen - HLen - 1).$$

f. Let $DB' = maskedDB' \oplus dbMask'$.

g. Separate DB' into a byte string HA' of length $HLen$ bytes and a byte string X of length $nLen - 2HLen - 1$ bytes as follows:

$$DB' = HA' \,||\, X.$$

5. Check for RSA-OAEP decryption errors:

    a. If Y is not a 00 byte, then $DecryptErrorFlag = True$.
    b. If HA' does not equal HA, then $DecryptErrorFlag = True$.
    c. If X does not have the form $PS \,||\, 01 \,||\, K$, where PS consists of zero or more consecutive 00 bytes, then $DecryptErrorFlag = True$.

    The type(s) of any error(s) found **shall not** be reported.
    (See the notes below for more information.)

6. Output of the decryption process:

    a. If $DecryptErrorFlag = True$, then output an indication of an (unspecified) decryption error, and exit without further processing. (See the notes below for more information.)

    b. Otherwise, output K, the portion of the byte string X that follows the leading 01 byte.

Output:

K: the recovered keying material (a byte string of length at most $nLen - 2HLen - 2$ bytes), or an error indicator.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (including d and any sensitive portions of A), any locally stored portions of K, and any other potentially sensitive locally stored values used or produced during its execution (including *DecryptErrorFlag*, *em*, *EM*, *HA*, *Y*, *maskedMGFSeed '*, *maskedDB'*, *mgfSeedMask '*, *mfgSeed '*, *dbMask '*, *DB'*, *HA'*, and *X*). Their destruction **shall** occur prior to or during any exit from the routine – whether exiting because of an error, or exiting normally, with the output of K. Note that the requirement for destruction includes any locally stored portions of the recovered keying material.

Notes:

1. Care **should** be taken to ensure that the different error conditions that may be detected in step 5 above cannot be distinguished from one another by an opponent, whether by error message or by process timing. Otherwise, an opponent may be able to obtain useful information about the decryption of a chosen ciphertext *C*, leading to the attack observed by Manger in [Manger 2001]. A single error message **should** be employed and output the same way for each type of decryption error. There **should** be no difference in the observable behavior for the different RSA-OAEP decryption errors.

2. In addition, care **should** be taken to ensure that even if there are no errors, an implementation does not reveal partial information about the encoded message *em* or *EM*. For instance, the observable behavior of the mask-generation function **should not** reveal even partial information about the MGF seed employed in the process (since that could compromise portions of the *maskedDB′* segment of *EM*). An opponent who can reliably obtain particular bits of *EM* for sufficiently many chosen-ciphertext values may be able to obtain the full decryption of an arbitrary ciphertext by applying the bit-security results of Håstad and Näslund [HN 1998].

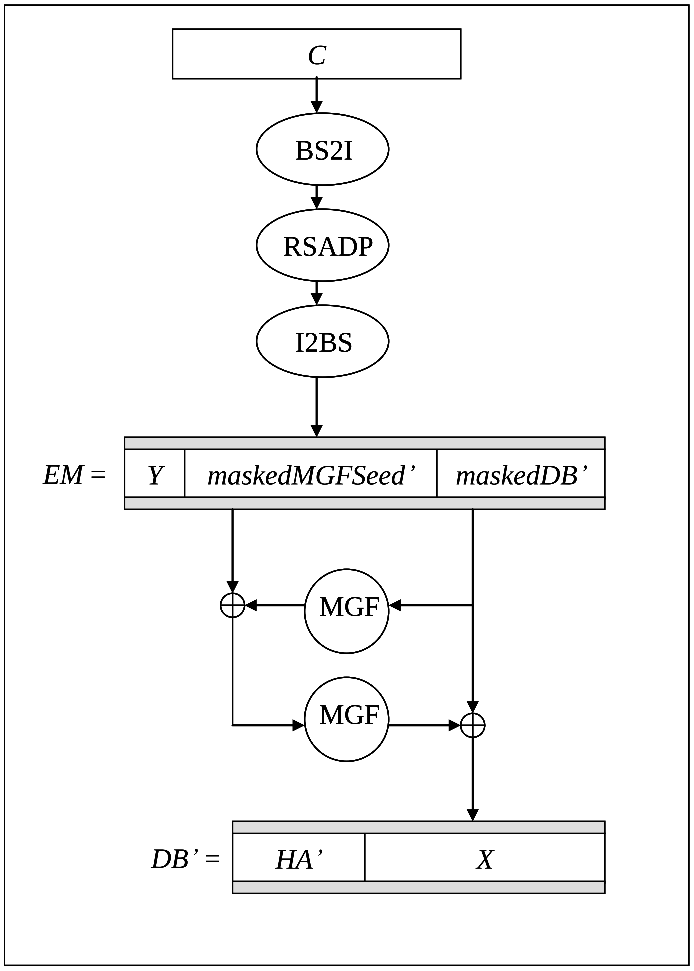

Figure 5: RSA-OAEP Decryption Operation

7.2.3 RSA-based Key-Encapsulation Mechanism with a Key-Wrapping Scheme (RSA-KEM-KWS)

RSA-KEM-KWS is used by the KTS-KEM-KWS key-transport schemes (see Section 9.3). RSA-KEM-KWS operations include a key-encapsulation method based on the RSASVE secret-value encapsulation operations and an **approved** key-derivation method. These operations are used to communicate a symmetric key-wrapping key to the intended receiving party. RSA-KEM-KWS operations also include an **approved** symmetric key-wrapping method, which is used to convey the actual keying material to the intended receiving party.

RSA-KEM-KWS can process keying material of any length supported by the key-wrapping algorithm.

7.2.3.1 RSA-KEM-KWS Components

RSA-KEM-KWS uses the following components:

1. KDM: A key-derivation method (see Section 5.5).

2. KWA: A symmetric key-wrapping algorithm, consisting of a wrapping operation KWA.WRAP and an unwrapping operation KWA.UNWRAP (see Section 7.2.3.2).

3. RSASVE: A secret-value encapsulation technique consisting of a pair of operations: one that generates a secret value and encrypts it to produce ciphertext (the RSASVE.GENERATE operation specified in Section 7.2.1.2), and another that recovers the secret value from the ciphertext (the RSASVE.RECOVER operation specified in Section 7.2.1.3).

4. RBG: A random bit generator (see Section 5.3).

7.2.3.2 Symmetric Key-Wrapping Methods

Symmetric key-wrapping is used to wrap (i.e., encrypt and integrity-protect) keying material. In this Recommendation, the KTS-KEM-KWS schemes specified in Section 9.3 use a key-wrapping operation to produce ciphertext *C* from keying material *K* using an **approved** key-wrapping method and a key-wrapping key *KWK*.

Three methods of key wrapping are **approved** for RSA-KEM-KWS: CCM, KW and KWP; CCM is specified in [SP 800-38C], while KW and KWP are specified in [SP 800-38F]. All three methods are modes of operation for AES, as specified in FIPS 197.

For environments in which additional input may need to be wrapped with the keying material to be transported, the CCM mode **shall** be used; otherwise, any of the three **approved** key-wrapping methods may be used.

7.2.3.2.1 Key-Wrapping using CCM

The input to the CCM mode specified in SP 800-38C includes a nonce *Nonce*, additional input[2] *A* and the keying material to be wrapped[3] *K*; the additional input could be a null string.

See Appendix A.1 in [SP 800-38C] for restrictions on the (individual and combined) lengths of the nonce, the additional input and the keying material to be wrapped.

Also required for the CCM mode is the length of the MAC tag to be produced[4] *TBits*; see Appendix B.2 in [SP 800-38C] for guidance on the selection of the length of the MAC tag. The wrapping operation encrypts the nonce, the additional input and the keying material to be wrapped using a key-wrapping key[5] *KWK*, resulting in C_1, which includes the ciphertext resulting from the encryption operation, together with a MAC tag of length *TBits*.

The additional input **shall** be available to both party U and party V (e.g., by an exchange of information or using information already known by both parties). Information that may be appropriate for inclusion in the additional input is discussed in Section 9.1.

Party U, who wraps the keying material, **shall** provide the nonce to the receiving party, party V. Either party U and party V **shall have** agreed (in advance) on the MAC-tag length, or party U **shall** send the MAC-tag length to party V, along with C_1.

The key-wrapping operation using CCM is

Function call: KWA.WRAP(*KWK*, *K*, *Nonce*, *TBits*, *A*)

Input:

1. *KWK*: The key-wrapping key; a 128-, 192- or 256-bit string.
2. *K*: The keying material to be wrapped; a byte string.
3. *Nonce*: A nonce, as specified in Section 5.4; a bit string.
4. *TBits*: The bit length of the MAC tag to be generated; an integer.
5. *A*: The additional input (see Section 9.1); a byte string.

Process:

1. If ((the length of *KWK* is not in the set {128, 192, 256}) OR (the value of *TBits* or the lengths of *Nonce*, *K* and/or *A* are not considered valid for the CCM mode[6])), then return an error indicator, and exit without further processing.

---

[2] Called associated data *A* in SP 800-38C.

[3] Called the payload *P* in SP 800-38C.

[4] Called *Tlen* in SP 800-38C.

[5] Called *K* in SP 800-38C.

[6] As specified in [SP 800-38C].

2. C_1 = **CCM.Encrypt**(*KWK*, *TBits*, *Nonce*, *K*, *A*).
3. Return C_1.

Output:

The ciphertext C_1 (a byte string) or an error indicator.

Note that the inputs to the **CCM.Encrypt** operation in process step 2 do not exactly match the specification of the Generation-Encryption process in [SP 800-38C], in which (the equivalents of) *KWK* and *TBits* are listed as prerequisites, while the nonce, additional input and keying material to be wrapped are listed as inputs.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (including *KWK*, *K*, and any sensitive portions of *A*), as well as any other potentially sensitive locally stored values used or produced during its execution. (The **CCM.Encrypt** routine **should** do the same.) Their destruction **shall** occur prior to or during any exit from the routine – whether exiting because of an error, or exiting normally, with the output of C_1.

7.2.3.2.2 Key-Unwrapping using CCM

When party V receives C_1, the plaintext keying material *K* may be recovered from C_1 using the key-wrapping key *KWK*, the received or agreed-upon length of the MAC tag *TBits*, the received nonce *Nonce* and the known value of any additional input *A* using the CCM mode. The unwrapping operation recovers the keying material *K* from C_1 (the encrypted keying material, concatenated with a MAC tag) using the key-wrapping key *KWK*, *Nonce* and *A*, and verifies their integrity using the MAC tag.

Restrictions on the nonce *Nonce*, the ciphertext C_1, the additional input *A* and the length of the MAC tag *TBits* are provided in [SP 800-38C].

Function: KWA.UNWRAP(*KWK*, C_1, *Nonce*, *TBits*, *A*,)

Input:

1. *KWK*: The key-wrapping key; a 128-, 192- or 256-bit string.
2. C_1: The ciphertext to be unwrapped; a byte string.
3. *Nonce*: A nonce, as specified in Section 5.4; a bit string.
4. *TBits*: The bit length of the MAC tag to be generated; an integer.
5. *A*: The additional input (see Section 9.1); a byte string.

Process:

1. If ((the length of *KWK* is not in the set {128, 192. 256}) OR (the value of *TBits*, or the lengths of *Nonce*, C_1 and/or *A* are not considered valid for the CCM mode[7])), then return an error indicator, and exit without further processing.
2. (*status*, *K*) = **CCM.Decrypt**(*KWK*, *TBits*, *Nonce*, *A*, C_1).

[7] As specified in [SP 800-38C].

3. If (*status* indicates an error), return *status*, and exit without further processing.
4. Return *K*.

Output:

The plaintext keying material *K* (a byte string), or an error indicator.

Note that the inputs to the **CCM.Decrypt** operation in process step 2 do not exactly match the specification of the Decryption-Verification process in [SP 800-38C], in which (the equivalents of) *KWK* and *TBits* are listed as prerequisites, while the nonce, the additional input and C_1 are listed as inputs.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (including *KWK* and any sensitive portions of *A*), any locally stored portions of *K*, and any other potentially sensitive locally stored values used or produced during its execution. (The **CCM.Decrypt** routine **should** do the same.) Their destruction **shall** occur prior to or during any exit from the routine – whether exiting early, because of an error, or exiting normally, with the output of *K*. Note that the requirement for destruction includes any locally stored portions of the unwrapped (i.e., plaintext) keying material.

7.2.3.2.3 Key Wrapping Using KW or KWP

The KW and KWP modes used for key wrapping do not include methods for handling additional input; therefore, these methods **shall not** be used when additional input needs to be wrapped with the keying material *K*.

The input to the KW or KWP modes specified in [SP 800-38F] is the keying material to be wrapped[8] *K*. The wrapping operation encrypts and integrity protects the keying material using a key-wrapping key[9] *KWK*. Limitations on the length of *K* are provided in Section 5.3.1 of [SP 800-38F].

Function: KWA.WRAP(*KWK*, *K*)

Input:

1. *KWK*: The key-wrapping key.
2. *K*: The keying material to be wrapped; a semiblock string for KW, or a byte string for KWP (see SP 800-38F for details).

Process:

1. If the length of *K* is not valid, then return an error indicator and exit without further processing.
2. C_1 = **Wrap**(*KWK*, *K*).
3. Return C_1.

[8] Called the plaintext *P* in [SP 800-38F].

[9] Called *K* in [SP 800-38C].

Output: The ciphertext C_1, or an error indicator.

In process step 2, **Wrap** is either **KW-AE** or **KWP-AE**, as specified in [SP 800-38F].

Also, note that the inputs to the **Wrap** operation in step 2 do not exactly match the specification for the KW and KWP wrapping methods in [SP 800-38F], in which *KWK* is listed as a prerequisite, while *K* is listed as an input.

A routine that implements this operation **shall** destroy any local copies of the input values *KWK* and *K*, as well as any other potentially sensitive locally stored values used or produced during its execution. (The **Wrap** routine **should** do the same.) Their destruction **shall** occur prior to or during any exit from the routine – whether exiting because of an error, or exiting normally, with the output of C_1.

7.2.3.2.4 Key Unwrapping Using KW or KWP

The unwrapping operation recovers the keying material *K* from the ciphertext C_1 using the key-wrapping key *KWK*. Limitations on the length of C_1 are provided in Section 5.3.1 of [SP 800-38F].

Function: KWA.UNWRAP(*KWK*, C_1)

Input:

1. *KWK*: The key-wrapping key.
2. C_1: The ciphertext to be unwrapped; a byte string.

Process:

1. If the length of C_1 is not valid, then return an error indicator, and exit without further processing.
2. (*status, K*) = **Unwrap**(*KWK*, C_1).
3. If (*status* indicates an error), return *status*, and exit without further processing.
4. Return *K*.

Output:

The plaintext keying material *K*, or an indication of an error.

In process step 2, **Unwrap** is either **KW-AD** or **KWP-AD**, as specified in [SP 800-38F].

Note that in process step 2, the returned values have been slightly altered from those specified in [SP 800-38F]. In [SP 800-38F], either the plaintext key or a "FAIL" indicator is returned, whereas process step 2 is specified with two return values: an indication of the status of the operation (e.g., SUCCESS or FAIL) and the plaintext key if the **Unwrap** operation doesn't indicate "FAIL.".

In addition, the inputs to the **Unwrap** operation in process step 2 do not exactly match the specification in [SP 800-38F], in which *KWK* is listed as a prerequisite, while C_1 is listed as an input.

A routine that implements this operation **shall** destroy any local copies of the input value *KWK*, any locally stored portions of *K*, and any other potentially sensitive locally stored values used or produced during its execution (the **Unwrap** routine **should** do the same.) Their destruction **shall** occur prior to or during any exit from the routine – whether exiting early, because of an error, or exiting normally, with the output of *K*. Note that the requirement for destruction includes any locally stored portions of the unwrapped (i.e., plaintext) keying material.

7.2.3.3 RSA-KEM-KWS Encryption Operation

RSA-KEM-KWS.ENCRYPT is illustrated in Figure 6. The public key-establishment key of the intended receiving party (i.e., party V) is input to RSASVE.GENERATE, obtaining a secret value *Z* and corresponding ciphertext byte string C_0. This secret value, along with any required *OtherInfo* shared by the sender and the intended receiving party (see Section 5.5), is used as input to the key-derivation method to obtain a key-wrapping key. This key-wrapping key is used by the key-wrapping method to encrypt the keying material, producing a ciphertext byte string C_1. Depending on the key-wrapping method used, other parameters or data may be required.

The output of the RSA-KEM-KWS encryption operation is the concatenation of C_0 and C_1.

Function call: RSA-KEM-KWS.ENCRYPT((*n*, *e*), *kwkBits, K*, *A*)

Input:

1. (*n*, *e*): the receiver's RSA public key.
2. *kwkBits*: the length of the key-wrapping key in bits; an integer.
3. *K*: the keying material to be wrapped; a byte string.
4. *Nonce*: A nonce, as specified in Section 5.4; a bit string.
5. *TBits*: The bit length of the MAC tag to be generated; an integer.
6. *A*: The additional input (see Sections 7.2.3.2.1 and 9.1); a byte string (may be the empty string).
7. *OtherInfo*: A bit string of context-specific data (see Section 5.5.1.2 for details).

Assumptions: The RSA public key is valid.

Process:

1. *nLen* = the length of *n* in bytes.
2. Length checking:
    a. *KLen* = the length of *K* in bytes.
    b. If *KLen* is not consistent with the lengths supported by the key-wrapping method used in Section 7.2.3.2, output an indication that the keying material length is not supported, and exit without further processing.
3. Secret-value generation and encapsulation:

Use the RSASVE.GENERATE operation specified in Section 7.2.1.2 to generate a secret-value byte string Z and a corresponding RSA-ciphertext byte string C_0 using party V's public key, where both Z and C_0 are *nLen* bytes in length.

$$(Z, C_0) = \text{RSASVE.GENERATE}((n, e)).$$

4. Key derivation:

   Derive a key-wrapping key *KWK* of length *kwkBits* bits from the byte string Z

   $$KWK = \text{KDM}(Z, kwkBits, OtherInfo),$$

   where the *OtherInfo* is known by both parties (see Section 5.5).

5. Key-wrapping:

   Wrap the keying material K using the key-wrapping key *KWK* (see Section 7.2.3.2) to produce a KWA-ciphertext byte string C_1. Depending on the key-wrapping method used, a nonce, the length of the MAC tag to be generated, and additional input are required, although the additional input could be a null string (see Section 7.2.3.2).

   $$C_1 = \text{KWA.WRAP}(KWK, K \{, Nonce, TBits, A\}) .$$

6. Concatenation:

   Concatenate the RSA-ciphertext byte string C_0 and the KWA-ciphertext byte string C_1 to form a ciphertext byte string C:

   $$C = C_0 \parallel C_1.$$

Output:

The ciphertext C (a byte string), or an error indicator.

Errors: An indication that the keying material length is not supported.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (including d and any sensitive portions of A and *OtherInfo*), any locally stored portions of K, and any other potentially sensitive locally stored values used or produced during its execution (such as Z and *KWK*). The RSASVE.RECOVER and KWA.UNWRAP routines **shall** destroy their own locally stored quantities, as specified in Sections 7.2.1.3 and 7.2.3.2. All of this required destruction **shall** occur prior to or during any exit from the RSA-KEM-KWS.DECRYPT routine – whether exiting because of an error, or exiting normally, with the output of K. Note that the requirement for destruction includes any locally stored portions of the plaintext keying material.

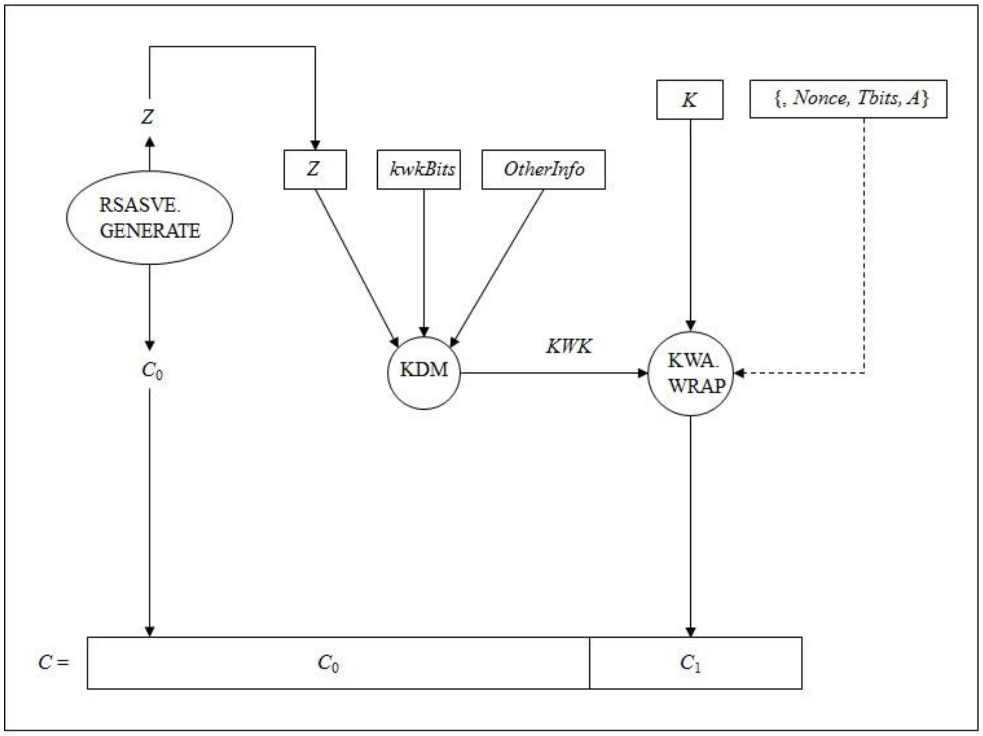

Figure 6: RSA-KEM-KWS Encryption Operation

7.2.3.4 RSA-KEM-KWS Decryption Operation

RSA-KEM-KWS.DECRYPT is illustrated in Figure 7. The private key-establishment key of the intended receiving party (i.e., party V) and C_0 are input to RSASVE.RECOVER, which returns the secret value *Z*. This secret value (along with any required *OtherInfo*) is used as input to the key-derivation method to recover the key-wrapping key. The key-wrapping key is then used to decrypt C_1 and recover the transported keying material. Depending on the key-wrapping method used, a nonce, a MAC-tag length and additional input are also required for the process.

Function call: RSA-KEM-KWS.DECRYPT((*n*, *d*), *C*, *kwkBits* {, *Nonce*, *TBits*, *A*})

Input:

1. (*n*, *d*): the recipient's RSA private key.
2. *C*: the ciphertext; a byte string.
3. *kwkBits*: the length of the key-wrapping key in bits; an integer.
4. *Nonce*: A nonce, as specified in Section 5.4; a bit string.
5. *TBits*: The bit length of the MAC tag to be generated; an integer.
6. *A*: additional input; a byte string.
7. *OtherInfo*: A bit string of context-specific data (see Section 5.5.1.2 for details).

Assumptions: The RSA private key is valid, and the value of *KBits* is known.

Process:

1. *nLen* = the length of *n* in bytes, where *nLen* = 256 or 384 in this Recommendation.

2. Length checking:

   a. *cLen* = the length of the ciphertext string *C* in bytes.

   b. If $cLen \le nLen$, or if $cLen - nLen$ is not consistent with the lengths supported by the symmetric key-wrapping algorithm, output an indication of a decryption error, and exit without further processing.

   c. If *kwkbits* is not among the lengths appropriate for the block-cipher algorithm used by the key-wrapping method, output an indication of a decryption error, and exit without further processing.

3. Separation:

   Separate the ciphertext byte string *C* into an RSA-ciphertext byte string C_0 of length *nLen* bytes and a KWA-ciphertext byte string C_1 of length $cLen - nLen$ bytes:

   $$C = C_0 \| C_1.$$

4. Recover the secret value, which then becomes the shared secret:

   Recover the secret-value byte string *Z* from the RSA-ciphertext byte string C_0 using the RSASVE.RECOVER operation specified in Section 7.2.1.3.

   $$Z = \text{RSASVE.RECOVER}((n, d), C_0)$$

   If an indication of a decryption error is returned, output an indication of a decryption error, and exit without further processing.

5. Key derivation:

   Derive a key-wrapping key *KWK* of length *kwkBits* bits from the byte string *Z*

   $$KWK = \text{KDM}(Z, kwkBits, OtherInfo),$$

   where the *OtherInfo* is known by both parties (see Section 5.9).

6. Key unwrapping:

   Unwrap the KWA-ciphertext byte string C_1 using the key-wrapping key *KWK* and, depending on the key-wrapping method used, the nonce, MAC-tag length and additional input needed to recover the keying material *K* (see Section 7.2.3.2), and verify the correctness of *A*:

   $$K = \text{KWA.UNWRAP}(KWK, C_1 \{, Nonce, TBits, A\}).$$

If the unwrapping operation outputs an error indicator, output an indication of a decryption error, and exit without further processing.

7. Output the keying material *K*.

Output:

The recovered keying material *K* (a byte string) that was wrapped, or an error indicator.

Errors: An indication of a decryption error.

A routine that implements this operation **shall** destroy any local copies of sensitive input values (including *d* and any sensitive portions of *A* and *OtherInfo*), any locally stored portions of *K*, and any other potentially sensitive locally stored values used or produced during its execution (such as *Z* and *KWK*). The **RSASVE.RECOVER** and **KWA.UNWRAP** routines **shall** destroy their own locally stored quantities, as specified in Sections 7.2.1.3 and 7.2.3.2. All of this required destruction **shall** occur prior to or during any exit from the **RSA-KEM-KWS.DECRYPT** routine – whether exiting because of an error, or exiting normally with the output of *K*. Note that the requirement for destruction includes any locally stored portions of the plaintext keying material.

Notes:

1. Care **should** be taken to ensure that the different error conditions in Steps 2, 4, and 6 cannot be distinguished from one another by an adversary, whether by error message or timing. Otherwise, an adversary may be able to obtain useful information about the decryption of a chosen ciphertext *C*, leading to the attack observed by Manger in [Manger 2001]. A single error message **should** be employed and output the same way for each error type. There **should** be no difference in timing or other behavior for the different errors.

2. In addition, care **should** be taken to ensure that even if there are no errors, an implementation does not reveal partial information about the secret value *Z*. For instance, the observable behavior of the KDM **should not** reveal even partial information about the *Z* value employed in the key-derivation process. An adversary who can reliably obtain particular bits of *Z* for sufficiently many chosen-RSA-ciphertext values may be able to obtain the full decryption of an arbitrary RSA-ciphertext by applying the bit-security results mentioned in Annex B.5.2.2 (last paragraph) of ANS X9.44 [ANS X9.44].

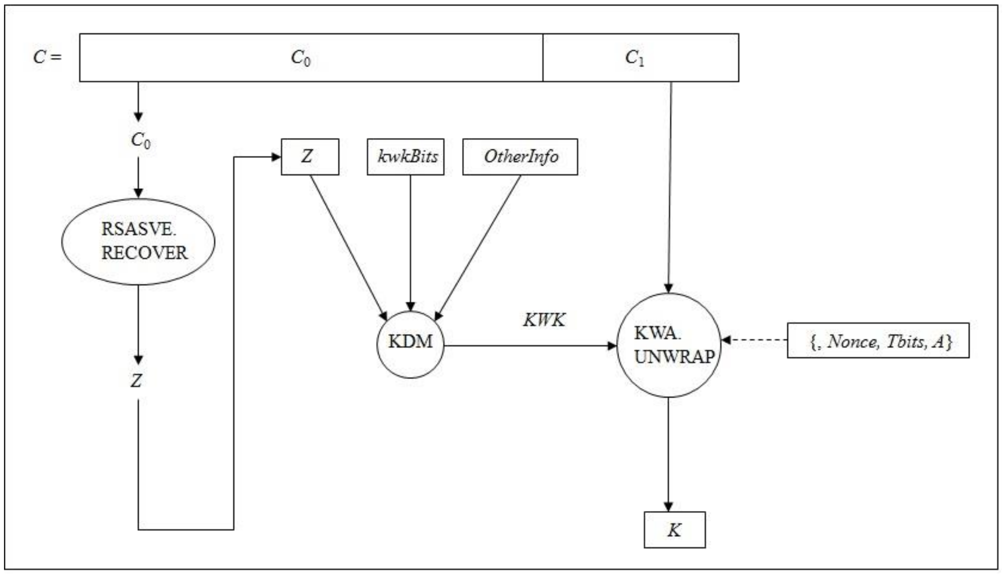

Figure 7: RSA-KEM-KWS Decryption Operation

8 Key-Agreement Schemes

In a key-agreement scheme, two parties, party U and party V, establish keying material over which neither has complete control of the result, but both have influence. This Recommendation provides two families of key-agreement schemes: KAS1 and KAS2. The KAS1 family consists of the KAS1-basic and KAS1-Party_V-confirmation schemes, and the KAS2 family consists of the KAS2-basic, KAS2-Party_V-confirmation, KAS2-Party_U-confirmation, and KAS2-bilateral-confirmation schemes. These schemes are based on secret-value encapsulation (see Section 7.2.1).

Key confirmation is included in some of these schemes to provide assurance that the participants share the same keying material; see Section 5.6 for the details of key confirmation. When possible, each party **should** have such assurance. Although other methods are often used to provide this assurance, this Recommendation makes no statement as to the adequacy of these other methods. Key confirmation may also provide assurance of private-key possession.

For both of the KAS1 and KAS2 schemes, Party V **shall** have an identifier, ID_V, that has an association with the key pair that is known (or discoverable) and trusted by party U (i.e., there **shall** be a trusted association between ID_V and party V's public key). For the KAS2 key-agreement schemes, party U **shall** also have such an identifier, ID_U.

A general flow diagram is provided for each key-agreement scheme. The dotted-line arrows represent the distribution of public keys by the parties themselves or by a third party, such as a Certification Authority (CA). The solid-line arrows represent the distribution of nonces or cryptographically protected values that occur during the key-agreement scheme. Note that the flow diagrams in this Recommendation omit explicit mention of various validation checks that are required. The flow diagrams and descriptions in this Recommendation assume a successful completion of the key-agreement process.

For each scheme, there are conditions that must be satisfied to enable proper use of that scheme. These conditions are listed as *assumptions*. Failure to meet all such conditions could yield undesirable results, such as the inability to communicate or the loss of security. As part of the proper implementation of this Recommendation, system users and/or agents trusted to act on their behalf (including application developers, system installers, and system administrators) are responsible for ensuring that all assumptions are satisfied at the time that a key-establishment transaction takes place.

8.1 Common Components for Key Agreement

The key-agreement schemes in this Recommendation have the following common components:

1. RSASVE: RSA secret-value encapsulation, consisting of a generation operation RSASVE.GENERATE and a recovery operation RSASVE.RECOVER (see Section 7.2.1).
2. KDM: A key-derivation method (see Section 5.5).

8.2 KAS1 Key Agreement

For each of the KAS1 key-agreement schemes, even if both parties have key-establishment key pairs, only party V's key-establishment key pair is used.

The KAS1 key-agreement schemes have the following general form:

1. Party U generates a secret value (which will become a shared secret) and a corresponding ciphertext using the RSASVE.GENERATE operation and party V's public key-establishment key, and then sends the ciphertext to party V.
2. Party V recovers the secret value from the ciphertext using the RSASVE.RECOVER operation and its private key-establishment key; the secret value is then considered to be the shared secret. Party V generates a nonce and sends it to party U.
3. Both parties then derive keying material from the shared secret and "other information", including party V's nonce, using a key-derivation method. The length of the keying material that can be agreed on is limited only by the length that can be output by the key-derivation method.
4. If key confirmation (KC) is incorporated in the scheme, then the derived keying material is parsed into two parts, *MacKey* and *KeyData*, and a string *MacData* is formed (see Sections 5.6 and 8.2.3.2.), *MacKey* and *MacData* are used to compute a MAC tag of length *MacTagLen* bytes (see Sections 5.2.1, 5.2.2, 5.6.1 and 5.6.3), and *MacTag* is sent from party V (the KC provider) to party U (the KC recipient). If the MAC tag computed by party V matches the MAC tag computed by party U, then the successful establishment of keying material is confirmed to party U.

The following schemes are defined:

1. **KAS1-basic,** the basic scheme without key confirmation (see Section 8.2.2).
2. **KAS1-Party_V-confirmation,** a variant of **KAS1-basic** with unilateral key confirmation from party V to party U (see Section 8.2.3).

For the security properties of the KAS1 key-agreement schemes, see Section 10.1.

8.2.1 KAS1 Assumptions

1. Party V has been designated as the owner of a key-establishment key pair that was generated as specified in Section 6.3. Party V has assurance of possession of the correct value for its private key as specified in Section 6.4.1.5.
2. Party U and party V have agreed upon an **approved** key-derivation method (see Section 5.5), as well as an **approved** algorithm to be used with that method (e.g., a specific hash function) and other associated parameters related to the cryptographic elements to be used.
3. If key confirmation is used, party U and party V have agreed upon an **approved** MAC algorithm and associated parameters, including the lengths of *MacKey* and *MacTag* (see Section 5.2).
4. When an identifier is used to label either party during the key-agreement process, both parties are aware of the particular identifier employed for that purpose. In particular, when an identifier is used to label party V during the key-agreement process, that identifier's association with party V's public key is trusted by party U. When an identifier is used to label party U during the key-agreement process, it has been selected/assigned in accordance with the requirements of the protocol relying upon the use of the key-agreement scheme.
5. Party U has obtained assurance of the validity of party V's public key, as specified in Section 6.4.2.

The following is an assumption for using any keying material derived during a KAS1 key-agreement scheme for purposes beyond those of the scheme itself.

> Party U has obtained (or will obtain) assurance that party V is (or was) in possession of the private key corresponding to the public key used during the key-agreement transaction, as specified in Section 6.4.2.3.

This assumption recognizes the possibility that assurance of private-key possession may be provided/obtained by means of key confirmation performed as part of a particular KAS1 transaction.

8.2.2 KAS1-basic

KAS1-basic is the basic key-agreement scheme in the KAS1 family. In this scheme, party V does not contribute to the formation of the shared secret; instead, a nonce is used as a party V-selected contribution to the key-derivation method, ensuring that both parties influence the derived keying material.

Let ($PubKey_V$, $PrivKey_V$) be party V's key-establishment key pair. Let *KBits* be the intended length in bits of the keying material to be established. The parties **shall** perform the following or an equivalent sequence of steps, as illustrated in Figure 8.

Party U **shall** execute the following key-agreement transformation in order to a) establish a shared secret *Z* with party V, and b) derive secret keying material from *Z*.

Actions: Party U generates a shared secret and derives secret keying material as follows:

1. Use the RSASVE.GENERATE operation in Section 7.2.1.2 to generate a secret value *Z* and a corresponding ciphertext *C* using party V's public key-establishment key *PubKey*$_V$. Note that the secret value Z will become a shared secret when recovered by Party V.
2. Send the ciphertext *C* to party V.
3. Obtain party V's nonce N_V from party V. If N_V is not available, return an error indicator without performing the remaining actions.
4. Construct the other information (e.g., *OtherInfo*) for key derivation (see Section 5.5) using the nonce N_V and the identifiers ID_U and ID_V, if available.
5. Use the agreed-upon key-derivation method (see Section 5.5) to derive secret keying material with the specified length from the shared secret value *Z* and other input. If the key-derivation method outputs an error indicator, return an error indicator without performing the remaining actions.
6. Output the *DerivedKeyingMaterial*.

Any local copies of *Z*, *OtherInfo*, *DerivedKeyingMaterial* and any intermediate values used during the execution of party U's actions **shall** be destroyed prior to or during the termination of the actions in steps 3, 4, and 6.

Party V **shall** execute the following key-agreement transformation in order to a) establish a shared secret *Z* with party U, and b) derive secret keying material from *Z*.

Actions: Party V obtains the shared secret and derives secret keying material as follows:

1. Receive a ciphertext *C* from party U.
2. Use the RSASVE.RECOVER operation in Section 7.2.1.3 to recover the secret value *Z* from the ciphertext *C* using the private key-establishment key *PrivKey*$_V$; hereafter, *Z* is considered to be a shared secret. If the call to RSASVE.RECOVER outputs an error indicator, return an error indicator without performing the remaining actions.
3. Obtain a nonce N_V (see Section 5.4), and send N_V to party U.
4. Construct the other information *OtherInfo* for key derivation (see Section 5.5) using the nonce N_V and the identifiers ID_U and ID_V, if available.
5. Use the agreed-upon key-derivation method to derive secret keying material with the specified length from the shared secret value *Z* and other input. If the key-derivation method outputs an error indicator, return an error indicator without performing the remaining actions.
6. Output the *DerivedKeyingMaterial*.

Any local copies of *Z*, *PrivKey*$_V$, *OtherInfo*, *DerivedKeyingMaterial* and any intermediate values used during the execution of party V's actions **shall** be destroyed prior to or during the termination of the actions in steps 2, 5 and 6.

Party U		**Party** V

		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	$PubKey_V$ $\dashleftarrow$	
(Z, C) = RSASVE.GENERATE($PubKey_V$)	C $\longrightarrow$	Z = RSASVE.RECOVER($PrivKey_V$, C)
Compute *DerivedKeyingMaterial* and destroy Z	N_V $\longleftarrow$	Compute *DerivedKeyingMaterial* and destroy Z

Figure 8: KAS1-basic Scheme

The messages may be sent in a different order, i.e., N_V may be sent before C.

It is extremely important that an implementation not reveal any sensitive information. It is also important to conceal partial information about the shared secret Z to prevent chosen-ciphertext attacks on the secret-value encapsulation scheme.

8.2.3 KAS1 Key Confirmation

The **KAS1-Party_V-confirmation** scheme is based on the **KAS1-basic** scheme.

8.2.3.1 KAS1 Key-Confirmation Components

The components for KAS1 key agreement with key confirmation are the components listed in Section 8.1, plus the following:

MAC: A message authentication code algorithm with the following parameters (see Section 5.2),

a. *MacKeyLen*: the length in bytes of *MacKey*, and

b. *MacTagLen*: the length in bytes of *MacTag*.

For KAS1 key confirmation, the length of the keying material **shall** be at least 14 bytes for a 256-byte (i.e., 2048-bit) modulus, and 16 bytes for a 384-byte (i.e., 3072-bit) modulus. The keying material is usually longer so that other keying material is available for subsequent operations. *MacKey* **shall** be the first *MacKeyLen* bytes of the keying material and **shall** be used only for the key-confirmation operation of a single transaction.

8.2.3.2 KAS1-Party_V-confirmation

Figure 9 depicts a typical flow for a KAS1 scheme with unilateral key confirmation from party V to party U. In this scheme, party V and party U assume the roles of key-confirmation provider and recipient, respectively.

To provide (and receive) key confirmation (as described in Section 5.6.1.1), both parties set $EphemData_V = N_V$, and $EphemData_U = C$:

Party V provides $MacTag_V$ to party U (as specified in Section 5.6.1.1, with P = V and R = U), where $MacTag_V$ is computed (as specified in Section 5.2.1) using

$$MacData_V = \text{“KC\_1\_V”} \,\|\, ID_V \,\|\, ID_U \,\|\, N_V \,\|\, C\{\,\|\, Text_V\}.$$

Party U uses the identical format and values to compute $MacTag_V$, and then verifies that the newly computed $MacTag_V$ matches the $MacTag_V$ value provided by party V.

The *MacKey* used during key confirmation **shall** be destroyed by party V immediately after the computation of $MacTag_V$, and by party U immediately after the verification of the received $MacTag_V$ or a (final) determination that the received $MacTag_V$ is in error.

Party U		Party V
		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	*PubKey* $\dashleftarrow$	
(Z, C) = RSASVE.GENERATE($PubKey_V$)	C $\longrightarrow$	Z = RSASVE.RECOVER($PrivKey_V$, C)
Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy Z	N_V $\longleftarrow$	Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy Z
$MacTag_V$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]	$MacTag_V$ $\longleftarrow$	$MacTag_V$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)

Figure 9: KAS1-Party_V-confirmation Scheme (from Party V to Party U)

Certain messages may be combined or sent in a different order (e.g., N_V and $MacTag_V$ may be sent together, or N_V may be sent before C).

8.3 KAS2 Key Agreement

In this family of key-agreement schemes, key-establishment key pairs are used by both party U and party V.

The schemes in this family have the following general form:

1. Party U generates a secret value (which will become a component of the shared secret) and a corresponding ciphertext using the RSASVE.GENERATE operation and party V's public key-establishment key, and sends the ciphertext to party V.
2. Party V recovers party U's secret component from the ciphertext received from party U using the RSASVE.RECOVER operation and its private key-establishment key.
3. Party V generates a secret value (which will become a second component of the shared secret) and the corresponding ciphertext using the RSASVE.GENERATE operation and party U's public key-establishment key, and sends the ciphertext to party U.
4. Party U recovers party V's secret component from the ciphertext received from party V using the RSASVE.RECOVER operation and its private key-establishment key.
5. Both parties concatenate the two secret components to form the shared secret, and then derive keying material from the shared secret and "other information" using a key-derivation method. The length of the keying material that can be agreed on is limited only by the length that can be output by the key-derivation method.
6. Party U and/or party V may additionally provide key confirmation. If key confirmation is incorporated, then the derived keying material is parsed into two parts, *MacKey* and *KeyData*. *MacKey* is then used to compute a MAC tag of *MacTagLen* bytes on *MacData* (see Sections 5.2.1, 5.2.2, 5.6.1 and 5.6.3). *MacTag* is sent from the KC provider to the KC recipient. If the MAC tag computed by the provider matches the MAC tag computed by the recipient, then the successful establishment of keying material is confirmed by the recipient.

The following schemes are defined:

1. **KAS2-basic,** the basic scheme without key confirmation (see Section 8.3.2).
2. **KAS2-Party_V-confirmation,** a variant of **KAS2-basic** with unilateral key confirmation from party V to party U (see Section 8.3.3.2).
3. **KAS2-Party_U-confirmation,** a variant of **KAS2-basic** with unilateral key confirmation from party U to party V (see Section 8.3.3.3).
4. **KAS2-bilateral-confirmation,** a variant of **KAS2-basic** with bilateral key confirmation between party U and party V (see Section 8.3.3.4).

For the security properties of the KAS2 key-agreement schemes, see Section 10.2.

8.3.1 KAS2 Assumptions

1. Each party has been designated as the owner of a key-establishment key pair that was generated as specified in Section 6.3. Prior to or during the key-agreement process, each

party has obtained assurance of its possession of the correct value for its own private key as specified in Section 6.4.1.5.

2. The parties have agreed upon an **approved** key-derivation method (see Section 5.6), as well as an **approved** algorithm to be used with that method (e.g., a hash function) and other associated parameters to be used for key derivation.
3. If key confirmation is used, party U and party V have agreed upon an **approved** MAC algorithm and associated parameters, including the lengths of *MacKey* and *MacTag* (see Section 5.2). The parties must also agree on whether one party or both parties will send *MacTag*, and in what order.
4. When an identifier is used to label a party during the key-agreement process, that identifier has a trusted association to that party's public key. (In other words, whenever both the identifier and public key of one participant are employed in the key-agreement process, they are associated in a manner that is trusted by the other participant.) When an identifier is used to label a party during the key-agreement process, both parties are aware of the particular identifier employed for that purpose.
5. Each party has obtained assurance of the validity of the public keys that are used during the transaction, as specified in Section 6.4.2.3.

The following is an assumption for using any keying material derived during a KAS2 key-agreement scheme for purposes beyond those of the scheme itself.

Each party has obtained (or will obtain) assurance that the other party is (or was) in possession of the private key corresponding to their public key that was used during the key-agreement transaction, as specified in Section 6.4.2.3.

This assumption recognizes the possibility that assurance of private-key possession may be provided/obtained by means of key confirmation performed as part of a particular KAS2 transaction.

8.3.2 KAS2-basic

Figure 10 depicts the typical flow for the **KAS2-basic** scheme. The parties exchange secret values that are concatenated together to form the mutually determined shared secret to be input to the key-derivation method.

Party U **shall** execute the following key-agreement transformation in order to a) establish a mutually determined shared secret Z with party V, and b) derive secret keying material from Z.

Actions: Party U generates a shared secret and derives secret keying material as follows:

1. Use the RSASVE.GENERATE operation in Section 7.2.1.2 to generate a secret value Z_U and a corresponding ciphertext C_U using party V's public key-establishment key $PubKey_V$.
2. Send the ciphertext C_U to party V.
3. Receive a ciphertext C_V from party V. If C_V is not available, return an error indicator without performing the remaining actions.
4. Use the RSASVE.RECOVER operation in Section 7.2.1.3 to recover Z_V from the ciphertext C_V using the private key-establishment key $PrivKey_U$. If the call to

RSASVE.RECOVER outputs an error indicator, return an error indicator without performing the remaining actions.

5. Construct the mutually determined shared secret Z from Z_U and Z_V

$$Z = Z_U \| Z_V.$$

6. Construct the other information (e.g., *OtherInfo*) for key derivation (see Section 5.5) using the identifiers ID_U and ID_V, if available.

7 Use the agreed-upon key-derivation method (see Section 5.6) to derive secret keying material with the specified length from the shared secret Z and other input. If the key-derivation method outputs an error indicator, return an error indicator without performing the remaining actions.

8. Output the *DerivedKeyingMaterial*.

Any local copies of Z, Z_U, Z_V, $PrivKey_U$, *OtherInfo*, *DerivedKeyingMaterial* and any intermediate values used during the execution of party U's actions **shall** be destroyed prior to or during the termination of the actions in steps 3, 4, 7 and 8.

Party V **shall** execute the following key-agreement transformation in order to a) establish a mutually determined shared secret Z with party U, and b) derive secret keying material from Z.

Actions: Party V generates a shared secret and derives secret keying material as follows:

1. Receive a ciphertext C_U from party U.
2. Use the RSASVE.RECOVER operation in Section 7.2.1.3 to recover Z_U from the ciphertext C_U using the private key-establishment key $PrivKey_U$. If the call to RSASVE.RECOVER outputs an error indicator, return an error indicator without performing the remaining actions.
3. Use the RSASVE.GENERATE operation in Section 7.2.1.2 to generate a secret value Z_V and a corresponding ciphertext C_V using party U's public key-establishment key $PubKey_U$.
4. Send the ciphertext C_V to party U.
5. Construct the mutually determined shared secret Z from Z_U and Z_V

$$Z = Z_U \| Z_V.$$

6. Construct the other information (e.g., *OtherInfo*) for key derivation (see Section 5.5) using the identifiers ID_U and ID_V, if available.
7. Use the agreed-upon key-derivation method (see Section 5.5) to derive secret keying material *DerivedKeyingMaterial* of length *KBits* from the shared secret Z and *OtherInfo*. If the key-derivation method outputs an error indicator, return an error indicator without performing the remaining actions.
8. Output the *DerivedKeyingMaterial*.

Any local copies of Z, Z_U, Z_V, $PrivKey_V$, *OtherInfo*, *DerivedKeyingMaterial* and any intermediate values used during the execution of party V's actions **shall** be destroyed prior to or during the termination of the actions in steps 2, 7 and 8.

Party U		**Party** V
($PubKey_U$, $PrivKey_U$)		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
	$PubKey_U$ — — — $\rightarrow$	Obtain party U's public key-establishment key
(Z_U, C_U) = RSASVE.GENERATE($PubKey_V$)	C_U $\longrightarrow$	Z_U = RSASVE.RECOVER($PrivKey_V$, C_U)
Z_V = RSASVE.RECOVER($PrivKey_U$, C_V)	C_V $\longleftarrow$	(Z_V, C_V) = RSASVE.GENERATE($PubKey_U$)
$Z = Z_U \mid\mid Z_V$		$Z = Z_U \mid\mid Z_V$
Compute *DerivedKeyingMaterial* and destroy Z		Compute *DerivedKeyingMaterial* and destroy Z

Figure 10: KAS2-basic Scheme

The messages may be sent in a different order, i.e., C_V may be sent before C_U.

It is extremely important that an implementation not reveal any sensitive information. It is also important to conceal partial information about Z_U, Z_V and Z to prevent chosen-ciphertext attacks on the secret-value encapsulation scheme. In particular, the observable behavior of the key-agreement process **should not** reveal partial information about the shared secret Z.

8.3.3 KAS2 Key Confirmation

The KAS2 key-confirmation schemes are based on the **KAS2-basic** scheme.

8.3.3.1 KAS2 Key-Confirmation Components

The components for KAS2 key agreement with key confirmation are the components in Section 8.1, plus the following:

3. MAC: A message authentication code algorithm with the following parameters (see Section 5.2)

    a. *MacKeyLen*: the length in bytes of *MacKey*.

b. *MacTagLen*: the length in bytes of *MacTag*.

For KAS2 key confirmation, the length of the keying material **shall** be at least 14 bytes for a 256-byte (i.e., 2048-bit) modulus, and at least 16 bytes for a 384-byte (i.e., 3072-bit) modulus. The keying material is usually longer so that other keying material is available for subsequent operations. *MacKey* **shall** be the first *MacKeyLen* bytes of the keying material and **shall** be used only for the key-confirmation operation.

8.3.3.2 KAS2-Party_V-confirmation

Figure 11 depicts a typical flow for a KAS2 scheme with unilateral key confirmation from party V to party U. In this scheme, party V and party U assume the roles of the key-confirmation provider and recipient, respectively.

To perform key confirmation (as described in Section 5.6.1.1), both parties set $EphemData_V = C_V$, and $EphemData_U = C_U$.

Party V provides $MacTag_V$ to party U (as specified in Section 5.6.1.1, with P = V and R = U), where $MacTag_V$ is computed (as specified in Section 5.2.1) on

$$MacData_V = \text{“KC\_1\_V”} \parallel ID_V \parallel ID_U \parallel C_V \parallel C_U \{ \parallel Text_V \}.$$

Party U (the KC recipient) uses the identical format and values to compute $MacTag_V$ and then verifies that the newly computed $MacTag_V$ equals $MacTag_V$ as provided by party V.

The MAC key used during key confirmation (i.e., *MacKey*) **shall** be destroyed by party V immediately after the computation of $MacTag_V$, and by party U immediately after the verification of the received $MacTag_V$ or a (final) determination that the received $MacTag_V$ is in error.

Party U		Party V
$(PubKey_U, PrivKey_U)$		$(PubKey_V, PrivKey_V)$
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
	$PubKey_U$ — — — $\rightarrow$	Obtain party U's public key establishment-key
(Z_U, C_U) = RSASVE.Generate($PubKey_V$)	C_U $\longrightarrow$	Z_U = RSASVE.Recover($PrivKey_V$, C_U)
Z_V = RSASVE.RECOVER($PrivKey_U$, C_V)	C_V $\longleftarrow$	(Z_V, C_V) = RSASVE.GENERATE($PubKey_U$)
$Z = Z_U \parallel Z_V$		$Z = Z_U \parallel Z_V$

Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy *Z*		Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy *Z*
$MacTag_V$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]	$MacTag_V$ $\longleftarrow$	$MacTag_V$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]

Figure 11: KAS2-Party_V-confirmation Scheme (from Party V to Party U)

Certain messages may be combined or sent in a different order (e.g., C_V and $MacTag_V$ may be sent together, or C_V may be sent before C_U).

8.3.3.3 KAS2-Party_U-confirmation

Figure 12 depicts a typical flow for a KAS2 scheme with unilateral key confirmation from party U to party V. In this scheme, party U and party V assume the roles of key-confirmation provider and recipient, respectively.

To provide (and receive) key confirmation (as described in Section 5.6.1.1), both parties set $EphemData_V = C_V$, and $EphemData_U = C_U$.

Party U provides $MacTag_U$ to party V (as specified in Section 5.6.1.1, with P = U and R = V), where $MacTag_U$ is computed (as specified in Section 5.2.1) on

$$MacData_U = \text{“KC\_1\_U”} \parallel ID_U \parallel ID_V \parallel C_U \parallel C_V \{ \parallel Text_U \}.$$

Party V (the KC recipient) uses the identical format and values to compute $MacTag_U$ and then verifies that the newly computed $MacTag_U$ matches the $MacTag_U$ value provided by party U.

The MAC key used during key confirmation **shall** be destroyed by party U immediately after the computation of $MacTag_U$, and by party V immediately after the verification of the received $MacTag_U$ or a (final) determination that the received $MacTag_U$ is in error.

Party U		**Party** V
$(PubKey_U, PrivKey_U)$		$(PubKey_V, PrivKey_V)$
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
	$PubKey_U$ — — — $\rightarrow$	Obtain party U's public key-establishment key
(Z_U, C_U) = RSASVE.GENERATE($PubKey_V$)	$\xrightarrow{\quad C_U \quad}$	Z_U = RSASVE.RECOVER($PrivKey_V$, C_U)
Z_V = RSASVE.RECOVER($PrivKey_U$, C_V)	$\xleftarrow{\quad C_V \quad}$	(Z_V, C_V) = RSASVE.GENERATE($PubKey_U$)
$Z = Z_U \| Z_V$		$Z = Z_U \| Z_V$
Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy *Z*		Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy *Z*
$MacTag_U$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_U$)]	$\xrightarrow{\quad MacTag_U \quad}$	$MacTag_U$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_U$)]

Figure 12: KAS2-Party_U-confirmation Scheme (from Party U to Party V)

Note that C_V may be sent before C_U; in which case C_U and $MacTag_U$ may be sent together.

8.3.3.4 KAS2-bilateral-confirmation

Figure 13 depicts a typical flow for a KAS2 scheme with bilateral key confirmation. In this scheme, party U and party V assume the roles of both the KC provider and recipient in order to obtain bilateral key confirmation.

To provide bilateral key confirmation (as described in Section 5.6.1.2), party U and party V exchange and verify *MacTags* that have been computed (as specified in Section 5.9.1.1) using $EphemData_U = C_U$, and $EphemData_V = C_V$.

Party V provides $MacTag_V$ to party U (as specified in Section 5.6.1.1, with P = V and R = U); $MacTag_V$ is computed by party V (and verified by party U) on

$$MacData_V = \text{"KC\_2\_V"} \| ID_V \| ID_U \| C_V \| C_U \{ \| Text_V \}.$$

Party U provides $MacTag_U$ to party V (as specified in Section 5.6.1.1, with P = U and R = V); $MacTag_U$ is computed by party U (and verified by party V) on

$$MacData_U = \text{“KC\_2\_U”} \,\|\, ID_U \,\|\, ID_V \,\|\, C_U \,\|\, C_V \{ \,\|\, Text_U \}.$$

The MAC key used during key confirmation **shall** be destroyed by each party immediately following its use to compute and verify the MAC tags used for key confirmation. Once party U has computed $MacTag_U$ and has either verified the received $MacTag_V$ or made a (final) determination that the received $MacTag_U$ is in error, party U **shall** immediately destroy its copy of *MacKey*. Similarly, after party V has computed $MacTag_V$ and has either verified the received $MacTag_U$ or made a (final) determination that the received $MacTag_U$ is in error, party V **shall** immediately destroy its copy of *MacKey*.

Party U		**Party** V
($PubKey_U$, $PrivKey_U$)		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
	$PubKey_U$ — — — $\rightarrow$	Obtain party U's public key-establishment key
(Z_U, C_U) = RSASVE.GENERATE($PubKey_V$)	C_U $\longrightarrow$	Z_U = RSASVE.RECOVER($PrivKey_V$, C_U)
Z_V = RSASVE.RECOVER($PrivKey_U$, C_V)	C_V $\longleftarrow$	(Z_V, C_V) = RSASVE.GENERATE($PubKey_V$)
$Z = Z_U \,\|\, Z_V$		$Z = Z_U \,\|\, Z_V$
Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy Z		Compute *DerivedKeyingMaterial* = *MacKey* || *KeyData* and destroy Z
$MacTag_V$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]	$MacTag_V$ $\longleftarrow$	$MacTag_V$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]
$MacTag_U$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_U$)]	$MacTag_U$ $\longrightarrow$	$MacTag_U$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_U$)]

Figure 13: KAS2-bilateral-confirmation Scheme

Certain messages may be sent in a different order (and/or combined with others), e.g., C_V may be sent before C_U and/or $MacTag_V$ may be sent before $MacTag_U$.

9 Key-Transport Schemes

In a key-transport scheme, two parties, party U (the *sender*) and party V (the *receiver*), establish keying material selected initially by party U. The keying material may be cryptographically bound to additional input (see Section 9.1).

In this Recommendation, two families of key-transport schemes are specified: KTS-OAEP and KTS-KEM-KWS.

Key confirmation is included in some of these schemes to provide assurance to the sender that the participants share the same keying material (see Section 5.6 for further details on key confirmation).

In key-transport schemes that provide key confirmation (see Sections 9.2.4.2 and 9.3.4.2), the transported keying material **shall** contain a MAC key as the first bits of the keying material such that

$$TransportedKeyingMaterial = MacKey \,||\, KeyData.$$

MacKey will be used for the computation and verification of the MAC tag. *KeyData* is the keying material to be used subsequent to the key-transport transaction. The MAC key **shall** be generated anew for each instance of a key-establishment transaction using an **approved** random bit generator that supports the security strength required for the key-establishment transaction. The MAC key length (in bits) **shall** be equal to or greater than the security strength associated with the modulus length (in bits) used in the key-establishment scheme (see SP 800-57-Part 1 [SP 800-57]). *KeyData* may be null, or may contain keying material to be used subsequent to the key-transport transaction. *MacKey* **shall** be used during key confirmation and then immediately destroyed.

In key-transport schemes that do not provide key confirmation (see Sections 9.2.4.1 and 9.3.4.1), the *TransportedKeyingMaterial* = *KeyData*.

A general flow diagram is provided for each key-transport scheme. The dotted-line arrows represent the distribution of public keys by the parties themselves or by a third party, such as a Certification Authority (CA). The solid-line arrows represent the distribution of cryptographically protected values that occur during the key-transport or key-confirmation process. Note that the flow diagrams in this Recommendation omit explicit mention of various validation checks that are required. The flow diagrams and descriptions in this Recommendation assume a successful completion of the key-transport process.

As in Section 8, there are conditions that must be satisfied for each key-transport scheme to enable proper use of that scheme. These conditions are listed as *assumptions*. Failure to meet any of such conditions could yield undesirable results, such as the inability to communicate or the loss of security. As part of the proper implementation of this Recommendation, system users and/or agents trusted to act on their behalf (including application developers, system installers, and system administrators) are responsible for ensuring that all assumptions are satisfied at the time that a key-establishment transaction takes place.

9.1 Additional Input

Additional input to the key-transport process may be employed to ensure that the transported keying material is adequately "bound" to the context of the key-transport transaction. The use of additional input, *A*, is explicitly supported by the key-transport schemes specified in Sections 9.2 and 9.3. Each party to a key-transport transaction **shall** know whether or not additional input is employed in that transaction.

Context-specific information that may be appropriate for inclusion in the additional input:

- Public information about parties U and V, such as names, e-mail addresses, and/or other identifiers.
- An identifier and/or other information associated with the RSA public key employed in the key-transport transaction. One might, for example, include (a hash of) a certificate that contains that RSA public key.
- Other public and/or private information shared between parties U and V before or during the transaction, such as nonces, counters, or pre-shared secret data. (The inclusion of private information is limited to situations in which the additional input, *A*, is afforded adequate confidentiality protection.)
- An indication of the protocol or application employing the key-transport scheme.
- Protocol-related information, such as a label or session identifier.
- An indication of the key-transport scheme used during the transaction.
- An indication of various parameter or primitive choices (e.g., hash functions, MAC algorithms, *MacTag* lengths, etc.).
- An indication of how the transported keying material should be parsed, including an indication of which algorithm(s) will use the (parsed) keying material.

Both parties to the key-transport transaction **shall** know the format of the additional input, *A*, and **shall** acquire *A* in time to use it as required by the scheme. The methods used for formatting and distributing the additional input are application-defined. System users and/or agents trusted to act on their behalf **should** determine that the information selected for inclusion in *A* and the methods used for formatting and distributing *A* meet the security requirements of those users.

9.2 KTS-OAEP: Key-Transport Using RSA-OAEP

The KTS-OAEP family of key-transport schemes is based on the RSA-OAEP encrypt and decrypt operations (see Section 7.2.2), which are, in turn, based on the asymmetric encryption and decryption primitives, RSAEP and RSADP (see Section 7.1). In this family, only party V's key pair is used.

The key-transport schemes of this family have the following general form:

1. Party U (the sender) encrypts the keying material (and possibly additional input – see Section 7.2.2.3) to be transported using the RSA-OAEP.ENCRYPT operation and party V's

(the receiver's) public key-establishment key to produce ciphertext, and sends the ciphertext to party V.

2. Party V decrypts the ciphertext using its private key-establishment key and the RSA-OAEP.DECRYPT operation to recover the transported keying material.

3. If key confirmation is incorporated, then the transported keying material is parsed into two parts, a transaction-specific (random) value for *MacKey*, followed by *KeyData.* The *Mackey* portion of the keying material and an **approved** MAC algorithm are used by each party to compute a MAC tag (of an appropriate, agreed-upon length) on what should be the same *MacData* (see Sections 5.6 and 9.2.4.2). The MAC tag computed by party V (the key-confirmation provider) is sent to party U (the key-confirmation recipient). If the value of the MAC tag sent by party V matches the MAC tag value computed by party U, then party U obtains a confirmation of the success of the key-transport transaction.

The common components of the schemes in the KTS-OAEP family are listed in Section 9.2.2. The following schemes are then defined:

1. **KTS-OAEP-basic**, the basic scheme without key confirmation (see Section 9.2.3).

2. **KTS-OAEP-Party_V-confirmation**, a variant of **KTS-OAEP-basic** with unilateral key confirmation from party V to party U (see Section 9.2.4).

For the security attributes of the KTS-OAEP family, see Section 10.3.

9.2.1 KTS-OAEP Assumptions

1. Party V has been designated as the owner of a key-establishment key pair that was generated as specified in Section 6.3. Party V has obtained assurance of its possession of the correct value for its private key as specified in Section 6.4.1.5.
2. The parties have agreed upon an **approved** hash function appropriate for use with the mask-generation function used by RSA-OAEP (see Sections 5.1, and 7.2.2).
3. Prior to or during the transport process, the sender and receiver have either agreed upon the form and content of the additional input *A* (a byte string to be cryptographically bound to the transported keying material so that the cipher is a cryptographic function of both values), or agreed that *A* will be an empty string (see Section 9.1 above).
4. If key confirmation is used, the parties have agreed upon an **approved** MAC algorithm and associated parameters (see Section 5.2).
5. When an identifier is used to label either party during the key-transport process, both parties are aware of the particular identifier employed for that purpose. In particular, the association of the identifier used to label party V with party V's public key is trusted by party U. When an identifier is used to label party U during the key-transport process, it has been selected/assigned in accordance with the requirements of the protocol relying upon the use of the key-transport scheme.
6. Party U has obtained assurance of the validity of party V's public key, as specified in Section 6.4.2.

7. Prior to or during the key-transport process, party U has obtained (or will obtain) assurance that party V is (or was) in possession of the (correct) private key corresponding to the public key-establishment key used during the transaction, as specified in Section 6.4.2.
8. Prior to or during the key-transport process, the keying material to be transported has been/is determined and has a format as specified at the beginning of Section 9.

9.2.2 Common components

The schemes in the KTS-OAEP family have the following common component:

1. RSA-OAEP: asymmetric operations, consisting of an encryption operation RSA-OAEP.ENCRYPT and a decryption operation RSA-OAEP.DECRYPT (see Section 7.2.2).

9.2.3 KTS-OAEP-basic

KTS-OAEP-basic is the basic key-transport scheme in the KTS-OAEP family without key confirmation.

Let ($PubKey_V$, $PrivKey_V$) be party V's (the receiver's) key-establishment key pair. Let K be the keying material to be transported from party U (the sender) to party V; note that the length of K is restricted by the length of the RSA modulus and the length of the hash-function used during the RSA-OAEP process (see Section 7.2.2.3). The parties **shall** perform the following or an equivalent sequence of steps, which are also illustrated in Figure 14.

Party U **shall** execute the following steps in order to transport keying material to party V.

Party U Actions:

1. Encrypt the keying material K using party V's public key-establishment key $PubKey_V$ and the additional input A, to produce a ciphertext C (see Section 7.2.2.3):

   $$C = \text{RSA-OAEP.E}{\small\text{NCRYPT}}(PubKey_V, K, A).$$

2. If an error indication has been returned, then return an error indication without performing the remaining actions.

3. Send the ciphertext C to party V.

Any local copies of K, A, and any intermediate values used during the execution of party U's actions **shall** be destroyed prior to or during step 2.

Party V **shall** execute the following steps when receiving keys transported from party U.

Party V Actions:

1. Receive the ciphertext C.

2. Decrypt the ciphertext C using the private key-establishment key $PrivKey_V$ and the additional input A, to recover the transported keying material K (see Section 7.2.2.4):

$$K = \text{RSA-OAEP.D}\textsc{ecrypt}(PrivKey_V, C, A).$$

If the decryption operation outputs an error indicator, return an error indication without performing the remaining actions.

3. Output K.

Any local copies of K, $PrivKey_V$, A, and any intermediate values used during the execution of party V's actions **shall** be destroyed prior to or during step 3.

Party U		**Party** V
K to be transported		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
C = RSA-OAEP. ENCRYPT($PubKey_V$, K, A)	C $\longrightarrow$	K = RSA-OAEP. DECRYPT($PrivKey_V$, C, A)

Figure 14: KTS-OAEP-basic Scheme

9.2.4 KTS-OAEP Key Confirmation

The **KES-OAEP-Party_V-confirmation** scheme is based on the **KTS-OAEP-basic scheme**.

9.2.4.1 KTS-OAEP Common Components for Key Confirmation

The components for **KTS-OAEP** with key confirmation are the same as for **KTS-OAEP-basic** (see Section 9.2.2), plus the following:

MAC: A message authentication code algorithm with the following parameters (see Section 5.2):

a. *MacKeyLen*: the length in bytes of *MacKey*.

b. *MacTagLen*: the length in bytes of *MacTag*.

For KAS2 key confirmation, the length of the keying material **shall** be at least 14 bytes for a 256-byte (i.e., 2048-bit) modulus, and 16 bytes for a 384-byte (i.e., 3072-bit) modulus, and usually longer so that keying material other than *MacKey* is available for subsequent operations. *MacKey* **shall** be the first *MacKeyLen* bytes of the keying material and **shall** be used only for the key-confirmation operation.

9.2.4.2 KTS-OAEP-Party_V-confirmation

KTS-OAEP-Party_V-confirmation is a variant of **KTS-OAEP-basic** with unilateral key confirmation from party V to party U.

Figure 15 depicts a typical flow for the **KTS-OAEP-Party_V-confirmation** scheme. In this scheme, party V and party U assume the roles of key-confirmation provider and recipient, respectively.

To provide (and receive) key confirmation (as described in Section 5.6.1.1), both parties form *MacData* with $EphemData_V = Null$, and $EphemData_U = C$:

Party V provides $MacTag_V$ to party U (as specified in Section 5.6.1.1, with P = V and R = U), where $MacTag_V$ is computed (as specified in Section 5.2.1) using

$$MacData_V = \text{“KC\_1\_V”} \parallel ID_V \parallel ID_U \parallel Null \parallel C\{ \parallel Text_V\}.$$

Party U uses the identical format and values to compute $MacTag_V$, and then verifies that the newly computed $MacTag_V$ matches the $MacTag_V$ value provided by party V.

The MAC tag used during key confirmation **shall** be destroyed by party V immediately after the computation of $MacTag_V$, and by party U immediately after the verification of the received $MacTag_V$ or a (final) determination that the received $MacTag_V$ is in error.

Party U		**Party** V
$K = MacKey \parallel KeyData$		$(PubKey_V, PrivKey_V)$
Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
C = RSA-OAEP. ENCRYPT($PubKey_V$, K, A)	C $\longrightarrow$	K = RSA-OAEP. DECRYPT($PrivKey_V$, C, A)
		$MacKey \parallel KeyData = K$
$MacTag_V$ =? $T_{MacTagBits}$[MAC($MacKey$, $MacData_V$)]	$MacTag_V$ $\longleftarrow$	$MacTag_V$ = $T_{MacTagBits}$[MAC($MacKey$, $MacData_V$)]

Figure 15: KTS-OAEP-Party_V-confirmation Scheme

9.3 KTS-KEM-KWS: Key-Transport using RSA-KEM-KWS

The KTS-KEM-KWS family of key-transport schemes is based on the RSA-KEM-KWS encrypt and decrypt operations (see Section 7.2.3). These operations employ the asymmetric RSASVE secret-value encapsulation operations and an **approved** key-derivation method to establish a key-wrapping key that is transaction specific. The key-wrapping key is used with an **approved** symmetric key-wrapping method to wrap (and unwrap) the keying material to be transported. In this family, only party V's key pair is used, and the transported keying material may be any length permitted by the key-wrapping method.

The key-transport schemes of this family have the following general form:

1. Using the RSA-KEM-KWS.ENCRYPT operation, party U (the sender) first generates a secret byte string *Z* and a corresponding ciphertext component by employing the RSASVE.GENERATE operation and the public key-establishment key of party V (the receiver). The byte string Z (along with *OtherInfo* or its equivalent) is then used as input to the key-derivation method to derive a transaction-specific key-wrapping key (*KWK*) of an appropriate, agreed-upon bit length *kwkBits*. The keying material to be transported is wrapped using the *KWK* and the symmetric key-wrapping method to produce a second ciphertext component. The two ciphertext components are sent to party V.

2. Using the RSA-KEM-KWS.DECRYPT operation, party V employs the RSASVE.RECOVER operation and its private key-establishment key to obtain *Z* from the first ciphertext component. Party V then employs the key-derivation method (with inputs *Z*, *kwkBits*, and *OtherInfo* or its equivalent) to derive the same *KWK* that was used by party U. The *KWK* and the symmetric key-unwrapping algorithm are used to obtain the transported keying material from the second ciphertext component.

3. If key confirmation is incorporated, the transported keying material consists of a transaction-specific (random) MAC key *MacKey*, followed by *KeyData*. *MacKey* and an **approved** MAC algorithm are used by each party to compute a MAC tag (of an appropriate, agreed-upon length) on what should be the same *MacData* (see Section 5.6). The value of *MacTag* computed by the party V (the key-confirmation provider) is sent to party U (the key-confirmation recipient). If the value of *MacTag* sent by party V matches the *MacTag* value computed by party U, then party U obtains a confirmation of the success of the key-transport transaction.

Common components of the schemes in the KTS-KEM-KWS family are listed in Section 9.3.2. Two schemes are then defined:

1. **KTS-KEM-KWS-basic**, the basic scheme without key confirmation (see Section 9.3.3).

2. **KTS-KEM-KWS-Party_V-confirmation**, a variant with unilateral key confirmation from the receiver (Party V) to the sender (Party U) (see Section 9.3.4).

For the security attributes of the KTS-KEM-KWS family, see Section 10.4.

9.3.1 KTS-KEM-KWS Family Assumptions

1. Party V has been designated as the owner of a key-establishment key pair that was generated as specified in Section 6.3. Party V has obtained assurance of the validity of its key pair as specified in Section 6.4.1.5, and has obtained assurance of its possession of the correct value for its private key as specified in Section 6.4.1.5.
2. The parties have agreed upon an **approved** key-derivation method (see Section 5.6), as well as an **approved** algorithm to be used with that method (e.g., a hash function) and other associated parameters to be used for key derivation.
3. The sender and receiver have agreed upon an **approved** key-wrapping method (i.e., either CCM, KW or KWP). The cryptographic algorithm used in the key-wrapping operation is either AES-128, AES-192 or AES-256, with key lengths of 128, 192 or 256 bits, respectively. The appropriate key length is provided as the value of *kwkBits* in

Section 9.3.3. The key-wrapping method protects the transported keying material at a security strength that is equal to or greater than the target security strength of the applicable key transport scheme (see Section 7.2.3). If the CCM mode is used during key wrapping, the sender and receiver have agreed on the counter-generation function, the formatting function, and the length of the MAC tag to be used during the key-wrapping operation (see Section 7.2.3.2.1). If the KW or KWP mode is used for key wrapping, the sender and receiver have agreed on the cipher function and the valid plaintext lengths to be used during key wrapping.

4. If the CCM mode will be used for key wrapping, prior to or during the transport process, the parties have either agreed upon the format and content of the additional input *A* (a byte string to be cryptographically bound to the transported keying material in that the cipher is a cryptographic function of both values), or agreed that *A* will be the empty string (see Section 9.1 above). Note that for the KW and KWP modes, additional input is not accommodated.
5. If key confirmation is used, the parties have agreed upon an **approved** MAC algorithm and associated parameters (see Section 5.2).
6. When an identifier is used to label either party during the key-transport process, both parties are aware of the particular identifier employed for that purpose. In particular, when an identifier is used to label party V during the key-transport process, that identifier's association to party V's public key is trusted by party U. When an identifier is used to label party U during the key-transport process, it has been selected/assigned in accordance with the requirements of the protocol relying upon the use of the key-transport scheme.
7. Party U has obtained assurance of the validity of party V's public key, as specified in Section 6.4.2.
8. Prior to or during the key-transport process, party U has obtained (or will obtain) assurance that party V is (or was) in possession of the private key corresponding to the public key-establishment key used during the transaction, as specified in Section 6.4.2.3.
9. Prior to or during the key-transport process, the keying material to be transported has been (or will be) determined, with a format as specified at the beginning of Section 9.

9.3.2 Common Components of the KTS-KEM-KWS Schemes

The schemes in the KTS-KEM-KWS family have the following common component:

1. RSA-KEM-KWS: Consisting of an encryption operation RSA-KEM-KWS.ENCRYPT and a decryption operation RSA-KEM-KWS.DECRYPT (see Section 7.2.3).

9.3.3 KTS-KEM-KWS-basic

KTS-KEM-KWS-basic is the basic key-transport scheme in the KTS-KEM-KWS family without key confirmation.

Let ($PubKey_V$, $PrivKey_V$) be party V's key-establishment key pair. Let K be a local copy of the keying material to be transported from party U to party V. The parties **shall** perform the following or an equivalent sequence of steps, which are also illustrated in Figure 16.

Party U **shall** execute the following steps in order to transport keying material to party V.

Party U Actions:

1. Using party V's public key-establishment key $PubKey_V$, the length $kwkBits$ of the key to be used for key-wrapping, and keying material K, generate a ciphertext C (see Section 7.2.3.3), which includes an encrypted Z value as C_0 and the wrapped keying material as C_1. If the CCM mode is to be used for key wrapping, also generate a nonce (*Nonce*), and include the nonce, the agreed-upon length of the MAC tag (*TBits*) and the additional input A as input when invoking the RSA-KEM-KWS.ENCRYPT routine, i.e.:

   C = RSA-KEM-KWS.ENCRYPT($PubKey_V$, *kwkBits*, K, {, *Nonce*, *TBits*, A}).

2. If an error indication is returned, return an error indication without performing the remaining actions.

3. Send the ciphertext C to party V, and, if the CCM mode was used for key wrapping, also send *Nonce*.

Any local copies of K and any intermediate values used during the execution of party U's actions **shall** be destroyed prior to or during steps 2 and 3.

Party V **shall** execute the following steps when receiving keys transported from party V.

Party V Actions:

1. Receive the transported keying material C, and if the CCM mode will be used for key unwrapping, also receive *Nonce*.

2. Using the private key-establishment key $PrivKey_V$, the ciphertext C, and the length *kwkBits* of the key-wrapping key, recover the keying material K (see Section 7.2.3.4). If the CCM mode is to be used for key unwrapping, also include the received *Nonce*, the agreed-upon length of the MAC tag (*TBits*) and the additional input A as input when invoking the RSA-KEM-KWS.DECRYPT routine, i.e.:

   K = RSA-KEM-KWS.DECRYPT($PrivKey_V$, C, *kwkBits* {, *Nonce*, *TBits*, A}).

3. If the decryption operation outputs an error indicator, return an error indication without performing the remaining actions.

4. Output K.

Any local copies of K, $PrivKey_V$, and any intermediate values used during the execution of party V's actions **shall** be destroyed prior to or during steps 3 and 4.

Party U		**Party** V
K to be transported		($PubKey_V$, $PrivKey_V$)

Obtain party V's public key-establishment key	$PubKey_V$ $\leftarrow$ — — —	
C = RSA-KEM-KWS. ENCRYPT($PubKey_V$, K {, $Nonce$, $TBits$, A})	C {, $Nonce$} $\longrightarrow$	K = RSA-KEM-KWS. DECRYPT($PrivKey_V$, C {, $Nonce$, $TBits$, A})

Figure 16: KTS-KEM-KWS-basic Scheme

Note that in Figure 16, *KLen* is not shown as an input to the decrypt function.

9.3.4 KTS-KEM-KWS Key Confirmation

The **KTS-KEM-KWS-Party_V-confirmation** scheme is based on the **KTS-KEM-KWS-basic** scheme.

9.3.4.1 KTS-KEM-KWS Common Components for Key Confirmation

The components for **KTS-KEM-KWS-Party_V-confirmation** are the same as for **KTS-KEM-KWS-basic** (see Section 9.3.2), plus the following:

MAC: A message authentication code algorithm with the following parameters (see Section 5.2):

a. the *MacKeyLen*: length in bytes of *MacKey*.

b. the *MacTagLen*: length in bytes of *MacTag*.

For KTS-KEM-KWS key confirmation, the length of the keying material **shall** be at least 14 bytes when *nLen* is 128 bytes and 16 bytes when *nLen* is 384 bytes, and usually longer so that keying material other than *MacKey* is available for subsequent operations. *MacKey* **shall** be the first *MacKeyLen* bytes of the keying material and **shall** be used only for key confirmation.

9.3.4.2 KTS-KEM-KWS-Party_V-confirmation

KTS-KEM-KWS-Party_V-confirmation is a variant of **KTS-KEM-KWS-basic** with unilateral key confirmation from party V to party U.

Figure 17 depicts a typical flow for the **KTS-KEM-KWS-Party_V-confirmation** scheme. In this scheme, party V and party U assume the roles of the key-confirmation provider and recipient, respectively.

To provide (and receive) key confirmation (as described in Section 5.6.1.1), both parties set $EphemData_V = Null$, and $EphemData_U = C$.

Party V provides $MacTag_V$ to party U (as specified in Section 5.6.1.1, with P = V and R = U), where $MacTag_V$ is computed (as specified in Section 5.2.1) using

$$MacData_V = \text{“KC\_1\_V”} \parallel ID_V \parallel ID_U \parallel Null \parallel C\{ \parallel Text_V\}.$$

Party U uses the identical format and values to compute $MacTag_V$ and then verifies that the newly computed $MacTag_V$ matches the $MacTag_V$ value provided by party V.

The *MacKey* value used during key confirmation **shall** be destroyed by party V immediately after the computation of $MacTag_V$, and by party U immediately after the verification of the received $MacTag_V$ or a (final) determination that the received $MacTag_V$ is in error.

Party U		**Party** V
K = *MacKey* ‖ *KeyData*		($PubKey_V$, $PrivKey_V$)
Obtain party V's public key-establishment key	$PubKey_V$ ← — — —	
C = RSA-KEM-KWS. ENCRYPT($PubKey_V$, *K* {, *Nonce*, *TBits*, *A*})	*C* {, *Nonce*} ——————→	*K* = RSA-KEM.KWS. DECRYPT($PrivKey_V$, *C* {, *Nonce*, *TBits*, *A*})
		MacKey || *KeyData* = *K*
$MacTag_V$ =? $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]	$MacTag_V$ ←—————	$MacTag_V$ = $T_{MacTagBits}$[MAC(*MacKey*, $MacData_V$)]

Figure 17: KTS-KEM-KWS-Party_V-confirmation Scheme

Note that in Figure 17, *KLen* is not shown in the decrypt operation.

10 Rationale for Selecting a Specific Scheme

The subsections that follow describe security properties that may be considered when a user and/or developer is choosing a key-establishment scheme from among the various schemes described in this Recommendation. The descriptions are intended to highlight certain similarities and differences between families of key-establishment schemes and/or between schemes within a particular family; they do not constitute an in-depth analysis of all possible security properties of every scheme under all adversary models.

The (brief) discussions will focus on the extent to which each participant in a particular transaction has assurance that fresh keying material has been successfully established with the intended party (and no one else). To that end, it is important to distinguish between the actual identifier of a participant in a key-establishment transaction and the role (party U or party V) assumed by that participant during the transaction. To simplify matters, in what follows, assume that the actual identifiers of the (honest) participants in a key-establishment transaction are the proverbial "Alice," acting as party U, and "Bob," acting as party V. (Pretend, for the sake of discussion, that these identifiers are unique among the universe of possible participants.) The identifier associated with their malevolent adversary is "Eve." The discussions will also consider the ill effects of certain compromises that might occur. The basic security properties that are cited depend on such factors as how a shared secret is calculated, how keying material is established, and what types of key-confirmation (if any) are incorporated into a given scheme.

Note 1: In order to provide concise descriptions of security properties possessed by the various schemes, it is necessary to make some assumptions concerning the format and type of data that is used as input during key derivation. The following assumptions are made solely for the purposes of Sections 10.1 through 10.4; they are not intended to preclude the options specified elsewhere in this Recommendation.

1. When discussing the security properties of schemes, it is assumed that the *OtherInfo* input to a (single-step) key-derivation function employed during a particular key-agreement transaction uses either the concatenation format or the ASN.1 format (see Section 5.5.1.2). It is also assumed that *OtherInfo* includes sufficiently specific identifiers for the participants in the transaction, an identifier for the key-establishment scheme being used during the transaction, and additional input (e.g., a nonce, and/or session identifier) that may provide assurance to one or both participants that the derived keying material will reflect the specific context in which the transaction occurs (see Section 5.5.1.2 and Appendix B of [SP 800-56A] for further discussion concerning context-specific information that may be appropriate for inclusion in *OtherInfo*).
2. In general, *OtherInfo* may include additional secret information (already shared between parties U and V), but that is not assumed to be the case in the analysis of the security properties that follows.
3. In cases where an **approved** extraction-then-expansion key-derivation procedure is employed (see Section 5.5.2), it is assumed that the equivalent of this *OtherInfo* is used as the *Context* input during the key-expansion step, as specified in [SP 800-56C].
4. Finally, it is assumed that all required nonces employed during a transaction are random nonces that contain a component consisting of a random bit string formed in accordance with the recommendations of Section 5.4.

Note 2: Different schemes may possess different security properties. A scheme should be selected based on how well the scheme fulfills system requirements. For instance, if messages are exchanged over a large-scale network where each exchange consumes a considerable amount of time, a scheme with fewer exchanges during a single key-agreement transaction might be preferable to a scheme with more exchanges, even though the latter may possess more security benefits. It is important to keep in mind that a key-establishment scheme is usually a component of a larger protocol that may offer security-related assurances beyond those that can be provided by the key-establishment scheme alone. For example, the protocol may include specific features that limit opportunities for accidental or intentional misuse of the key-establishment component of the protocol. Protocols, per se, are not specified in this Recommendation.

10.1 Rationale for Choosing a KAS1 Key-Agreement Scheme

In both schemes included in the KAS1 family, only Bob (assumed to be acting as party V) is required to own an RSA key pair that is used in the key-agreement transaction. Assume that the identifier used to label party V during the transaction is one that is associated with Bob's RSA public key in a manner that is trusted by Alice (who is acting as party U). This can provide Alice with some level of assurance that she has correctly identified the party with whom she will be establishing keying material if the transaction is successfully completed.

Each KAS1 scheme requires Alice to employ the RSASVE.GENERATE operation to select a (random) secret value *Z* and encrypt it as ciphertext *C* using Bob's RSA public key. Unless Bob's corresponding private key has been compromised, Alice has assurance that no unintended entity (i.e., no one but Bob) could employ the RSASVE.RECOVER operation to obtain *Z* from *C*. Absent the compromise of Bob's RSA private key and/or *Z*, Alice may attain a certain level of confidence that she has correctly identified party V as Bob. Alice's level of confidence is dependent upon:

- The specificity of the identifier that is associated with Bob's RSA public key,
- The degree of trust in the association between that identifier and the public key,
- The assurance of the validity of the public key, and
- The availability of evidence that the keying material has been correctly derived by Bob using *Z* (and the other information input to the agreed-upon key-derivation method), e.g. through key confirmation with Bob as the provider.

In general, Bob has no assurance that party U is Alice, since Bob has no assurance concerning the accuracy of any identifier that may be used to label party U (unless, for example, the protocol using a key-agreement scheme from the KAS1 family also includes additional elements that establish a trusted association between an identifier for Alice and the ciphertext, *C*, that she contributes to the transaction while acting as party U).

The assurance of freshness of the derived keying material that can be obtained by a participant in a KAS1 transaction is commensurate with the participant's assurance that different input will be supplied to the agreed-upon key-derivation method during each such transaction. Alice can obtain assurance that fresh keying material will be derived based on her unilateral selection and contribution of the random Z value. Bob can obtain similar assurance owing to his selection and contribution of the nonce N_V, which is also used as input to the agreed-upon key-derivation method.

The KAS1-Party_V-confirmation scheme permits party V to provide evidence to party U that keying material has been correctly derived. When the KAS1-Party_V-confirmation scheme is employed during a key-agreement transaction, party V provides a key-confirmation MAC tag, $MacTag_V$, to party U as specified in Section 8.2.3.2. This allows Alice (who is acting as party U, the key-confirmation recipient) to obtain assurance that party V has possession of the *MacKey* derived from the shared secret *Z* (and nonce N_V) and has used it with the appropriate $MacData_V$ to compute the received $MacTag_V$. In the absence of a compromise of secret information (e.g., Bob's RSA private key and/or *Z*), Alice can also obtain assurance that the appropriate identifier has been used to label party V, and that the participant acting as party V is indeed Bob, the owner of the RSA public key associated with that identifier.

Specifically, by successfully comparing the received value of $MacTag_V$ with her own computation, Alice (acting as party U, the key-confirmation recipient) may obtain assurance that

1. Party V has correctly recovered *Z* from *C*, and, therefore, possesses the RSA private key corresponding to Bob's RSA public key – from which it may be inferred that party V is Bob;
2. Both parties have correctly computed (at least) the same *MacKey* portion of the derived keying material;

3. Both parties agree on the values (and representation) of ID_V, ID_U, N_V, C, and any other data included in $MacData_V$; and
4. Bob (acting as party V) has actively participated in the transaction.

Consequently, when the KAS1-Party_V-confirmation scheme is employed during a particular key-agreement transaction (and neither Z nor Bob's RSA private key has been compromised), Alice can obtain assurance of the active (and successful) participation in the transaction by Bob.

The acquisition of Bob's RSA private key by their adversary, Eve, may lead to the compromise of shared secrets and derived keying material from past, current, and future legitimate transactions (i.e., transactions that involve honest parties and are not actively influenced by an adversary) that employ the compromised private key. For example, Eve may be able to compromise a particular KAS1 transaction between Alice and Bob as long as she acquires the ciphertext, C, contributed by Alice and the nonce, N_V, contributed by Bob (as well as any other data used as input during key derivation). In addition to compromising legitimate KAS1 transactions, once Eve has learned Bob's RSA private key, she may be able to impersonate Bob while acting as party V in future KAS1 transactions (with Alice or any other party). Other schemes and applications that rely on the compromised private key may also be adversely affected. (See the appropriate subsection for details.)

Even without knowledge of Bob's private key, if Eve learns the value of Z that has been (or will be) used in a particular KAS1 transaction between Alice and Bob, then she may be able to derive the keying material resulting from that transaction as easily as Alice and Bob (as long as Eve also acquires the value of N_V and any other data used as input during key derivation). Alternatively, armed with knowledge of the Z value that has been (or will be) selected by Alice, Eve might be able to insert herself into the transaction (in the role of party V) while masquerading as Bob.

10.2 Rationale for Choosing a KAS2 Key-Agreement Scheme

In the schemes included in the KAS2 family, both Alice (assumed to be acting as party U) and Bob (assumed to be acting as party V) are required to own an RSA key pair that is used in their key-agreement transaction. Assume that the identifier used to label party V during the transaction is one that is associated with Bob's RSA public key in a manner that is trusted by Alice. Similarly, assume that the identifier used to label party U during the transaction is one that is associated with Alice's RSA public key in a manner that is trusted by Bob. This can provide each party with some level of assurance concerning the identifier of the other party, with whom keying material will be established if the transaction is successfully completed.

Each KAS2 scheme requires Alice to employ the RSASVE.GENERATE operation to select a (random) secret value Z_U and encrypt it as ciphertext C_U using Bob's RSA public key. Unless Bob's corresponding private key has been compromised, Alice has assurance that no unintended entity (i.e., no one but Bob) could employ the RSASVE.RECOVER operation to obtain Z_U from C_U. Similarly, each KAS2 scheme requires Bob to employ the RSASVE.GENERATE operation to select a (random) secret value Z_V and encrypt it as ciphertext C_V using Alice's RSA public key. Unless Alice's corresponding private key has been compromised, Bob has assurance that no unintended entity (i.e., no one but Alice) could employ the RSASVE.RECOVER operation to obtain Z_V from C_V.

Absent the compromise of Bob's RSA private key and/or Z_U, Alice may attain a certain level of confidence that she has correctly identified party V as Bob. Alice's level of confidence is commensurate with:

- The specificity of the identifier that is associated with Bob's RSA public key,
- The degree of trust in the association between that identifier and Bob's public key,
- The assurance of validity of the public key, and
- The availability of evidence that the keying material has been correctly derived by Bob using $Z = Z_U \| Z_V$ (and the other information input to the agreed-upon key-derivation method), e.g. through key-confirmation with Bob as the provider.

Similarly, absent the compromise of Alice's private key and/or Z_V, Bob may attain a certain level of confidence that he has correctly identified party U as Alice. Bob's level of confidence is commensurate with:

- The specificity of the identifier that is associated with Alice's RSA public key,
- The degree of trust in the association between that identifier and Alice's public key,
- The assurance of validity of the public key, and
- The availability of evidence that the keying material has been correctly derived by Alice using $Z = Z_U \| Z_V$ (and the other information input to the agreed-upon key-derivation method), e.g. through key-confirmation with Alice as the provider.

The assurance of freshness of the derived keying material that can be obtained by a participant in a KAS2 transaction is commensurate with the participant's assurance that different input will be supplied to the agreed-upon key-derivation method during each such transaction. Alice can obtain assurance that fresh keying material will be derived, based on her selection and contribution of the random Z_U component of Z. Bob can obtain similar assurance owing to his selection and contribution of the random Z_V component of Z.

Evidence that keying material has been correctly derived may be provided by using one of the three schemes from the KAS2 family that incorporates key confirmation. The KAS2-Party_V-confirmation scheme permits party V (Bob) to provide evidence of correct key derivation to party U (Alice); the KAS2-Party_U-confirmation scheme permits party U (Alice) to provide evidence of correct key derivation to party V (Bob); the KAS2-bilateral-confirmation scheme permits each party to provide evidence of correct key derivation to the other party.

When the KAS2-Party_V-confirmation scheme or the KAS2-bilateral-confirmation scheme is employed during a key-agreement transaction, party V provides a key-confirmation MAC tag, $MacTag_V$, to party U as specified in Section 8.3.3.2 or Section 8.3.3.4, respectively. This allows Alice (who is the recipient of $MacTag_V$) to obtain assurance that party V has possession of the *MacKey* derived from the shared secret Z and has used it with the appropriate $MacData_V$ to compute the received $MacTag_V$. In the absence of a compromise of secret information (e.g., Bob's RSA private key and/or Z_U), Alice can also obtain assurance that the appropriate identifier has been used to label party V, and that the participant acting as party V is indeed Bob, the owner of the RSA public key associated with that identifier.

Similarly, when the KAS2-Party_U-confirmation scheme or the KAS2-bilateral-confirmation scheme is employed during a key-agreement transaction, party U provides a key-confirmation MAC tag, $MacTag_U$, to party V as specified in Section 8.3.3.3 or Section 8.3.3.4, respectively. This allows Bob (who is the recipient of $MacTag_U$) to obtain assurance that party U has possession of the *MacKey* derived from the shared secret *Z* and has used it with the appropriate $MacData_U$ to compute the received $MacTag_U$. In the absence of a compromise of secret information (e.g., Alice's RSA private key and/or Z_V), Bob can also obtain assurance that the appropriate identifier has been used to label party U, and that the participant acting as party U is indeed Alice, the owner of the RSA public key associated with that identifier.

Specifically, by successfully comparing the value of a received MAC tag with his/her own computation, a key-confirmation recipient in a KAS2 transaction (be it Alice or Bob) may obtain the following assurances.

1. He/She has correctly decrypted the ciphertext that was produced by the other party and, thus, that he/she possesses the RSA private key corresponding to the RSA public key that was used by the other party to produce that ciphertext – from which it may be inferred that the other party had access to the RSA public key owned by the key-confirmation recipient. For example, if Alice is a key-confirmation recipient, she may obtain assurance that she has correctly decrypted the ciphertext C_V using her RSA private key, and so may also obtain assurance that her corresponding RSA public key was used by party V to produce C_V.

2. The ciphertext sent to the other party was correctly decrypted and, thus, the other party possesses the RSA private key corresponding to the RSA public key that was used to produce that ciphertext – from which it may be inferred that the other party is the owner of that RSA public key. For example, if Alice is a key-confirmation recipient, she can obtain assurance that party V has correctly decrypted the ciphertext C_U using the RSA private key corresponding to Bob's RSA public key – from which she may infer that party V is Bob.

3. Both parties have correctly computed (at least) the same *MacKey* portion of the derived keying material.

4. Both parties agree on the values (and representation) of ID_V, ID_U, C_V, C_U, and any other data included as input to the MAC algorithm.

5. Assuming that there has been no compromise of either participant's RSA private key and/or either component of *Z*, a key-confirmation recipient in a KAS2 transaction can obtain assurance of the active (and successful) participation in that transaction by the owner of the RSA public key associated with the key-confirmation provider. For example, if Alice is a key-confirmation recipient, she can obtain assurance that Bob has actively – and successfully – participated in that KAS2 transaction.

The acquisition of a single RSA private key by their adversary, Eve, will not (by itself) lead to the compromise of derived keying material from legitimate KAS2 transactions between Alice and Bob that employ the compromised RSA key pair. (In this context, a "legitimate transaction" is one in which Alice and Bob act honestly, and there is no active influence exerted by Eve.) However, if Eve acquires an RSA private key, she may be able to impersonate that RSA key pair's owner while participating in KAS2 transactions. (For example, If Eve acquires Alice's private key, she may be able to impersonate Alice – acting as party U or as party V – in KAS2 transactions with Bob or any other party). Other schemes and applications that rely on the compromised private key may also be adversely affected. (See the appropriate subsection for details.)

Similarly, the acquisition of one (but not both) of the secret Z components, Z_U or Z_V, would not (by itself) compromise the keying material derived during a legitimate KAS2 transaction between Alice and Bob in which the compromised value was used as one of the two components of Z. However, armed with knowledge of only one Z component, Eve could attempt to launch an active attack against the party that generated it. For example, if Eve learns the value of Z_U that has been (or will be) contributed by Alice, then Eve might be able to insert herself into the transaction by masquerading as Bob (while acting as party V). Likewise, an adversary who knows the value of Z_V that has been (or will be) selected by Bob might be able to participate in the transaction by masquerading as Alice (while acting as party U).

10.3 Rationale for Choosing a KTS-OAEP Key-Transport Scheme

In each of the key-transport schemes included in the KTS-OAEP family, only Bob (assumed to be acting as party V, the key-transport receiver) is required to own an RSA key pair that is used in the transaction. Assume that the identifier used to label party V during the transaction is one that is associated with Bob's RSA public key in a manner that is trusted by Alice (who is acting as party U, the key-transport sender). This can provide Alice with some level of assurance that she has correctly identified the party with whom she will be establishing keying material if the key-transport transaction is successfully completed.

Each KTS-OAEP scheme requires Alice to employ the RSA-OAEP.ENCRYPT operation to encrypt the selected keying material (and any additional input) as ciphertext C, using Bob's RSA public key. Unless Bob's corresponding private key has been compromised, Alice has assurance that no unintended entity (i.e., no one but Bob) could employ the RSA-OAEP.DECRYPT operation to obtain the transported keying material from C. Absent the compromise of Bob's RSA private key (or some compromise of the keying material itself – perhaps prior to transport), Alice may attain a certain level of confidence that she has correctly identified party V as Bob. Alice's level of confidence is commensurate with:

- The specificity of the identifier that is associated with Bob's RSA public key,
- The degree of trust in the association between that identifier and the public key,
- The assurance of validity of the public key, and
- The availability of evidence that the transported keying material has been correctly recovered from C by Bob, e.g. through key confirmation with Bob as the provider.

In general, Bob has no assurance that party U is Alice, since Bob has no assurance concerning the accuracy of any identifier that may be used to label party U (unless, for example, the protocol using a key-transport scheme from the KTS-OAEP family also includes additional elements that establish a trusted association between an identifier for Alice and the ciphertext, C, that she sends to Bob while acting as party U).

Due to Alice's unilateral selection of the keying material, only she can obtain assurance of its freshness. (Her level of confidence concerning its freshness is dependent upon the actual manner in which the keying material is generated by/for her.) Given that Bob simply accepts the keying material that is transported to him by Alice, he has no assurance that it is fresh.

The randomized plaintext encoding used during the RSA-OAEP.ENCRYPT operation can provide assurance to Alice that the value of C will change from one KTS-OAEP transaction with Bob to the next, which may help obfuscate the occurrence of a repeated transport of the same keying material from Alice to Bob, should that ever be necessary.

The KTS-OAEP-Party_V-confirmation scheme permits party V to provide evidence to party U that keying material has been correctly recovered from the ciphertext *C*. When the KTS-OAEP-Party_V-confirmation scheme is employed during a key-transport transaction, party V provides a key-confirmation MAC tag ($MacTag_V$) to party U as specified in Section 9.2.4.2. This allows Alice (who is acting as party U, the key-confirmation recipient) to obtain assurance that party V has recovered the fresh MAC key (*MacKey*) that was included in the transported keying material and that party V has used it with the appropriate $MacData_V$ to compute the received $MacTag_V$. In the absence of a compromise of secret information (e.g., Bob's RSA private key and/or the MAC key), Alice can also obtain assurance that the appropriate identifier has been used to label party V, and that the participant acting as party V is indeed Bob, the owner of the RSA public key associated with that identifier.

Specifically, by successfully comparing the received value of $MacTag_V$ with her own computation, Alice (acting as party U, the key-confirmation recipient) may obtain assurance that

1. Party V has correctly recovered *MacKey* from *C*, and, therefore, possesses the RSA private key corresponding to Bob's RSA public key – from which it may be inferred that party V is Bob;
2. Both parties agree on the values (and representation) of ID_V, ID_U, *C*, and any other data included in $MacData_V$; and
3. Bob has actively participated in the transaction (as party V), assuming that neither the transported MAC key nor Bob's RSA private key has been compromised. Alice's level of confidence is commensurate with her confidence in the freshness of the MAC key.

The acquisition of Bob's RSA private key by their adversary, Eve, may lead to the compromise of keying material established during past, current, and future legitimate transactions (i.e., transactions that involve honest parties and are not actively influenced by an adversary) that employ the compromised private key. For example, Eve may be able to compromise a particular KTS-OAEP transaction between Alice and Bob, as long as she also acquires the ciphertext, *C*, sent from Alice to Bob. In addition to compromising legitimate KTS-OAEP transactions, once Eve has learned Bob's RSA private key, she may be able to impersonate Bob while acting as party V in future KTS-OAEP transactions (with Alice or any other party). Other schemes and applications that rely on the compromised private key may also be adversely affected. (See the discussions of other schemes in this section.)

Even without knowledge of Bob's private key, if the KTS-OAEP-Party_V-confirmation scheme is used during a particular key-transport transaction, and Eve learns the value of *MacKey* that Alice will send to Bob, then it may be possible for Eve to mislead Alice about Bob's (active and successful) participation. As long as Eve also acquires the value of *C* intended for Bob (and any other data needed to form $MacData_V$), it may be possible for Eve to correctly compute $MacTag_V$ and return it to Alice as if it had come from Bob (who may not even be aware that Alice has initiated a transaction with him). Such circumstances could arise, for example, if (in violation of this Recommendation) Alice were to use the same MAC key while attempting to transport keying material to multiple parties (including both Bob and Eve).

10.4 Rationale for Choosing a KTS-KEM-KWS Key-Transport Scheme

In each of the key-transport schemes included in the KTS-KEM-KWS family, only Bob (assumed to be acting as party V, the key-transport receiver) is required to own an RSA key pair that is used in the transaction. Assume that the identifier used to label party V during the transaction is one that is associated with Bob's RSA public key in a manner that is trusted by Alice (who is acting as party U, the key-transport sender). This can provide Alice with some level of assurance that she has correctly identified the party with whom she will be establishing keying material if the key-transport transaction is successfully completed.

Each KTS-KEM-KWS scheme requires Alice (as part of the RSA-KEM-KWS.ENCRYPT operation) to employ the RSASVE.GENERATE operation to select a (random) secret value Z and encrypt it as ciphertext C_0 using Bob's RSA public key. A key-wrapping key, KWK, is derived from Z (and other information input to the agreed-upon key-derivation method) and then KWK is used by Alice to wrap the selected keying material (and any additional input) as ciphertext C_1, using the agreed-upon symmetric key-wrapping method. The concatenation of C_0 and C_1 forms the ciphertext C that Alice sends to party V.

Unless Bob's RSA private key has been compromised, Alice has assurance that no unintended entity (i.e., no one but Bob) could employ the RSASVE.RECOVER operation to obtain Z from C_0, derive KWK, and then use that key to obtain the plaintext keying material from C_1. Absent the compromise of Bob's RSA private key, Z, and/or KWK (or some compromise of the plaintext keying material itself – perhaps prior to transport), Alice may attain a certain level of confidence that she has correctly identified party V as Bob. Alice's level of confidence is commensurate with:

- The specificity of the identifier that is associated with Bob's RSA public key,
- The degree of trust in the association between that identifier and the public key,
- The assurance of validity of the public key,
- The perceived strength of the key-wrapping method, and
- The availability of evidence that the transported keying material has been correctly unwrapped by Bob, e.g. through key confirmation with Bob as the provider.

In general, Bob has no assurance that party U is Alice, since Bob has no assurance concerning the accuracy of any identifier that may be used to label party U (unless, for example, the protocol using a key-transport scheme from the KTS-KEM-KWS family also includes additional elements that establish a trusted association between an identifier for Alice and the ciphertext, C, that she sends to Bob while acting as party U).

Due to Alice's unilateral selection of the keying material that is transported, only she can obtain assurance of its freshness. (Her level of confidence concerning its freshness is dependent upon the actual manner in which the keying material is generated by/for her.) Given that Bob simply accepts the keying material that is transported to him by Alice, he has no assurance that it is fresh.

The use of a transaction-specific (random) Z (and hence, a transaction-specific KWK) can provide assurance to Alice that the values of both C_0 and C_1 will change from one KTS-KEM-KWS transaction with Bob to the next, which may help obfuscate the occurrence of a repeated transport of the same keying material from Alice to Bob, should that ever be necessary.

The KTS-KEM-KWS-Party_V-confirmation scheme permits party V to provide evidence to party U that keying material has been correctly recovered from the ciphertext *C*. When the KTS-KEM-KWS-Party_V-confirmation scheme is employed during a key-transport transaction, party V provides a key-confirmation MAC tag ($MacTag_V$) to party U as specified in Section 9.3.4.2. This allows Alice (who is acting as party U, the key-confirmation recipient) to obtain assurance that party V has recovered the fresh MAC key (*MacKey*) that was included in the transported keying material and that party V has used it with the appropriate $MacData_V$ to compute the received $MacTag_V$. In the absence of a compromise of secret information (e.g., Bob's RSA private key, *Z, KWK*, and/or *MacKey*), Alice can also obtain assurance that the appropriate identifier has been used to label party V, and that the participant acting as party V is indeed Bob, the owner of the RSA public key associated with that identifier.

Specifically, by successfully comparing the received value of $MacTag_V$ with her own computation, Alice (acting as party U, the key-confirmation recipient) may obtain assurance that

- Party V has correctly recovered *Z* from C_0, and, therefore, possesses the RSA private key corresponding to Bob's RSA public key – from which it may be inferred that party V is Bob;
- Bob has correctly derived the key-wrapping key, *KWK*, from Z and the other information input to the key-derivation method – from which it may be inferred that both parties agree on the value(s) (and representation) of that other information;
- Bob has successfully used *KWK* to recover *MacKey* from C_1;
- Both parties agree on the values (and representation) of ID_V, ID_U, *C*, and any other data included in $MacData_V$; and
- Bob has actively participated in the transaction (as party V), assuming that neither Bob's RSA private key, *Z*, *KWK*, nor *MacKey* has been compromised. Alice's level of confidence is commensurate with her confidence in the freshness of the MAC key.

The acquisition of Bob's RSA private key by their adversary, Eve, may lead to the compromise of keying material established during past, current, and future legitimate transactions (i.e., transactions that involve honest parties and are not actively influenced by an adversary) that employ the compromised private key. For example, Eve may be able to compromise a particular KTS-KEM-KWS transaction between Alice and Bob, given that she acquires the ciphertext (and other data) sent from Alice to Bob. (Besides the ciphertext, Eve must acquire the other information that was used along with *Z* to derive the key-wrapping key, and any other inputs necessary for the key-unwrapping process). In addition to compromising legitimate KTS-KEM-KWS transactions, once Eve has learned Bob's RSA private key, she may be able to impersonate Bob while acting as party V in future KTS-KEM-KWS transactions (with Alice or any other party). Other schemes and applications that rely on the compromised private key may also be adversely affected. (See the appropriate subsection for details.)

Even without knowledge of Bob's private key, if Eve learns the value of *Z* that has been (or will be) used in a particular KTS-KEM-KWS transaction between Alice and Bob, then Eve may be able to derive the key-wrapping key, *KWK*, and unwrap the transported keying material as easily as Bob (as long as she also acquires the ciphertext and other requisite data; a similar result would follow from Eve's acquisition of *KWK* by other means.) Alternatively, armed with knowledge of the *Z* value that has been (or will be) selected by Alice (or armed with the corresponding value of *KWK*), Eve might be able to insert herself into the transaction (in the role of party V) while masquerading as Bob.

If the KTS-KEM-KWS-Party_V-confirmation scheme is used during a particular key-transport transaction, and Eve learns the value of *MacKey* that Alice will send to Bob, then it may be possible for Eve to mislead Alice about Bob's (active and successful) participation. As long as Eve also acquires the value of *C* intended for Bob (and any other data needed to form $MacData_V$), it may be possible for Eve to correctly compute $MacTag_V$ and return it to Alice as if it had come from Bob (who may not even be aware that Alice has initiated a transaction with him). Such circumstances could arise, for example, if (in violation of this Recommendation) Alice were to use the same MAC key while attempting to transport keying material to multiple parties (including both Bob and Eve).

11 Key Recovery

For some applications, the secret keying material used to protect data or to process protected data may need to be recovered (for example, if the normal reference copy of the secret keying material is lost or corrupted). In this case, either the secret keying material or sufficient information to reconstruct the secret keying material needs to be available (for example, the keys and other inputs to the scheme used to perform the key-establishment process).

Keys used during the key-establishment process **shall** be handled in accordance with the following:

1. One or both keys of a key pair **may** be saved.
2. A key-wrapping key **may** be saved.

In addition, the following information that is used during key-establishment may need to be saved:

3. The nonce(s),
4. The ciphertext,
5. Additional input, and
6. *OtherInfo*, or its equivalent.

General guidance on key recovery and the protections required for each type of key is provided in the Recommendation for Key Management [SP 800-57].

12 Implementation Validation

When the NIST Cryptographic Algorithm Validation System (CAVS) has established a validation program for this Recommendation, a vendor **shall** have its implementation tested and validated by the Cryptographic Algorithm Validation Program (CAVP) and Cryptographic Module Validation Program (CMVP) in order to claim conformance to this Recommendation. Information on the CAVP and CMVP is available at http://csrc.nist.gov/cryptval/.

An implementation claiming conformance to this Recommendation **shall** include one or more of the following capabilities:

- Key-pair generation as specified in Section 6.3.
- Explicit public-key validation as specified in Section 6.4.3.
- A key-agreement scheme from Section 8, together with an **approved** random bit generator and an **approved** key-derivation method from Section 5.5.
- A key-transport scheme as specified in Section 9, together with an **approved** random bit generator, an **approved** hash function, an **approved** symmetric key-wrapping method, as specified in Section 7.2.3.2, and an **approved** key-derivation method from Section 5.5 for RSA-KEM-KWS based schemes.

An implementer **shall** also identify the appropriate specifics of the implementation, including:

- The hash function to be used (see Section 5.1).
- The *MacKey* length(s). The minimum length is 14 bytes for a 256-byte (i.e., 2048-bit) modulus, and 16 bytes for a 384-byte (i.e., 3072-bit) modulus.
- The length of the MAC tag (the minimum length is 64 bits, i.e., 8 bytes).
- The key-establishment schemes available (see Sections 8 and 9).
- The key-derivation method to be used if a key-agreement scheme is implemented, including the format of *OtherInfo* or its equivalent (see Section 5.5).
- The type of nonces to be generated (see Section 5.4).
- How assurance of private-key possession and assurance of public-key validity are expected to be achieved by both the owner and the recipient.
- If a key-transport scheme is implemented, the key-wrapping method used and whether or not a capability is available to handle additional input (see Section 7.2.3.2).
- The RBG used, and its security strength (see Section 5.3).

Appendix A: References

A.1 Normative References

[FIPS 140]	FIPS 140-2, Security Requirements for Cryptographic Modules, May 25, 2001. FIPS 140-3 is currently under development.
[FIPS 140 IG]	FIPS 140-2 Implementation Guidance; available at http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-2/FIPS1402IG.pdf
[FIPS 180]	FIPS 180-4 Secure Hash Standard, March 2012.
[FIPS 186]	FIPS 186-4, Digital Signature Standard, July 2013.
[FIPS 197]	FIPS 197, Advanced Encryption Standard, November 2001.
[FIPS 198]	FIPS 198-1, The Keyed-Hash Message Authentication Code (HMAC), July 2008.
[FIPS 202]	Draft FIPS 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2014.
[SP 800-38B]	NIST SP 800-38B, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, May 2005.
[SP 800-38C]	NIST SP 800-38C, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, May 2004.
[SP 800-38F]	NIST SP 800-38F, Recommendation for Block Cipher Modes of Operation: Methods for Key-wrapping, December 2012.
[SP 800-56A]	NIST SP 800-56A, Recommendation for Pair-Wise Key-establishment Schemes Using Discrete Logarithm Cryptography, Revision 2, May 2013.
[SP 800-56C]	NIST SP 800-56C, Recommendation for Key Derivation through Extraction-then-Expansion, November 2011.
[SP 800-57]	NIST SP 800-57-Part 1, Recommendation for Key Management, Revision 3, July 2012.
[SP 800-67]	NIST SP 800-67, Recommendation for the Triple Data Encryption Algorithm, Revision 1, January 2012.
[SP 800-89]	NIST SP 800-89, Recommendation for Obtaining Assurances for Digital Signature Applications, November 2006.
[SP 800-90A]	Recommendation for Random Number Generation Using Deterministic Random Bit Generators, January 2012.
[SP 800-90B]	DRAFT Recommendation for the Entropy Sources Used for Random Bit Generation, August 2012.
[SP 800-90C]	DRAFT Recommendation for Random Bit Generator (RBG) Constructions, August 2012.

[SP 800-108] NIST SP 800-108, Recommendation for Key Derivation Using Pseudorandom Functions, October 2009.

[SP 800-133] NIST SP 800-133, Recommendation for Cryptographic Key Generation, November 2012.

[SP 800-135] NIST SP 800-135, Recommendation for Existing Application-Specific Key Derivation Functions, Revision 1, December 2011.

[ANS X9.44] ANS X9.44 Public Key Cryptography for the Financial Services Industry: Key Establishment Using Integer Factorization Cryptography, August 2007.

[ISO/IEC 8825] ISO/IEC 8825-1, Information Technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), December 2008.

[PKCS 1] Public Key Cryptography Series (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, February 2003.

A.2 Informative References

[Manger 2001] *A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0*, J. Manger, In J. Kilian, editor, Advances in Cryptology – Crypto 2001, pp. 230 – 238, Springer-Verlag, 2001.

[RSA 1978] *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, R. Rivest, A. Shamir and L. Adleman, Communications of the ACM, 21(2), pp. 120 – 126, 1978.

[HN 1998] *The Security of all RSA and Discrete Log Bits*, J. Håstad and M. Näslund, Proc. of the 39th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 510 – 521, 1998.

[Boneh 1999] *Twenty Years of Attacks on the RSA Cryptosystem*, D. Boneh, Notices of the American Mathematical Society (AMS), 46(2), 203 – 213. 1999.

Appendix B: Data Conversions (Normative)

B.1 Integer-to-Byte String (I2BS) Conversion

Input: A non-negative integer X and the intended length n of the byte string satisfying

$$2^{8n} > X.$$

Output: A byte string S of length n bytes.

```
1. $Q_{n+1} = X$.
2. For $i = n$ to 1 by −1
   2.1 $Q_i = \lfloor (Q_{i+1})/256 \rfloor$.
   2.2 $X_i = Q_{i+1} - (Q_i \cdot 256)$.
   2.3 $S_i$ = the 8-bit binary representation of the non-negative integer $X_i$.
3. Let $S_1, S_2, \ldots, S_n$ be the bytes of $S$ from leftmost to rightmost.
4. Output $S$.
```

B.2 Byte String to Integer (BS2I) Conversion

Input: A non-empty byte string S (*SLen* is used to denote the length of the byte string).

Output: A non-negative integer X.

```
1. Let $S_1, S_2, \ldots S_{SLen}$ be the bytes of $S$ from first to last (i.e., from leftmost to rightmost).
2. Let $X = 0$.
3. For $i = 1$ to $SLen$ by 1
   3.1 Let $Xi$ be the non-negative integer whose 8-bit binary representation is $Si$.
   3.2 $X = X + (X_i \cdot 256^{SLen-i})$.
4. Output $X$.
```

Appendix C: Prime-Factor Recovery (Normative)

The following algorithm recovers the prime factors of a modulus, given the public and private exponents. The algorithm is based on Fact 1 in [Boneh 1999].

Function call: RecoverPrimeFactors(n, e, d)

Input:

1. n: modulus
2. e: public exponent
3. d: private exponent

Output:

1. (p, q): prime factors of modulus

Errors: “prime factors not found”

Assumptions: The modulus n is the product of two prime factors p and q; the public and private exponents satisfy $de \equiv 1 \pmod{\lambda(n)}$ where $\lambda(n) = \text{LCM}(p - 1, q - 1)$

Process:

```
1. Let $k = de - 1$. If $k$ is odd, then go to Step 4.

2. Write $k$ as $k = 2^t r$, where $r$ is the largest odd integer dividing $k$, and $t \geq 1$.

3. For $i = 1$ to 100 do:

        a. Generate a random integer $g$ in the range $[0, n-1]$.

        b. Let $y = g^r \bmod n$.

        c. If $y = 1$ or $y = n - 1$, then go to Step g.

        d. For $j = 1$ to $t - 1$ do:

              i.    Let $x = y^2 \bmod n$.

              ii.   If $x = 1$, go to Step 5.

              iii.  If $x = n - 1$, go to Step g.

              iv.   Let $y = x$.

        e. Let $x = y^2 \bmod n$.

        f. If $x = 1$, go to Step 5.
```

g. Continue.

4. Output “prime factors not found,” and exit without further processing.
5. Let p = GCD(y – 1, n) and let q = n/p.
6. Output (p, q) as the prime factors.

Any local copies of d, p, q , k, t, r, x, y, g and any intermediate values used during the execution of the RecoverPrimeFactors function **shall** be destroyed prior to or during steps 4 and 6. Note that this includes the values for p and q that are output in step 6.

Notes:

1. According to Fact 1 in [Boneh 1999], the probability that one of the values of y in an iteration of Step 3 reveals the factors of the modulus is at least 1/2, so on average, at most two iterations of that step will be required. If the prime factors are not revealed after 100 iterations, then the probability is overwhelming that the modulus is not the product of two prime factors, or that the public and private exponents are not consistent with each other.
2. The algorithm bears some resemblance to the Miller-Rabin primality-testing algorithm (see, e.g., FIPS 186).
3. The order of the recovered prime factors p and q may be the reverse of the order in which the factors were generated originally.

Appendix D: Revisions (Informative)

In the 2014 revision, the following revisions were made:

- Section 3.1 – Added definitions of assumptions, binding, destroy, fresh, key-derivation function, key-derivation method, key-wrapping key, MAC tag, and trusted association; removed algorithm identifier, digital signature, initiator, responder.
- Section 4 – Used party U and party V to name the parties, rather than using the initiator and responder as the parties. In Sections 8 and 9, the schemes have been accordingly renamed: KAS1-responder-confirmation is now KAS1-Party_V-confirmation, KAS2-responder-confirmation is now KAS2-Party_V-confirmation, KAS2-initiator-confirmation is now KAS2-Party_U-confirmation, KTS-OAEP-receiver-confirmation is not KTS-OAEP-Party_V-confirmation, and KTS-KEM-KWS-receiver-confirmation is now KTS-KEM-KWS-Party_V-confirmation.
- Section 4 – Added requirements to destroy the local copies of secret and private values and all intermediate calculations before terminating a routine normally or in response to an error. Instructions to this effect have been inserted throughout the document.
- The discussion about identifiers vs. identity and binding have been moved to Section 4.1.
- Section 4.3 – The phrase "IFC-based" has been removed throughout the document.
- Section 5.4 – More discussion has been added about the use of nonces, including new requirements and recommendations.
- Section 5.5 – Key derivation has been divided into single-step key derivation methods (Section 5.5.1), an extract-then-expand key derivation procedure (Section 5.5.2) and application-specific key-derivation methods (Section 5.5.3).
- Section 5.5.1.2 – The use of *OtherInfo* (including identifiers) during the derivation of keys is recommended, but no longer required (Section 5.5.1.2).
- Moved the general introduction of key-confirmation to Section 5.9 – The discussion now incorporates the material from Section 6.6 of the previous version of the document.
- Section 6.4 – There is now a longer, and more thorough discussion of validity in Section 6.4. The concept of trusted associations has been introduced.
- Section 6.4.1.1 – Removed "or TTP" from the following: "The key pair can be revalidated at any time by the owner as follows…."
- Section 7.2.3.2 – Moved discussion of symmetric key-wrapping methods from Section 5.7 to Section 7.2.3.2; much more information is now provided.
- Section 10 – The rationale for choosing each scheme type has been combined in this new section, along with a discussion of their security properties.
- The old Appendix A, Summary of Differences between this Recommendation and ANS X9.44 (Informative), was removed.

- The old Appendix E becomes Appendix D, and the changes introduced in this Revision are listed here.
- All figures are replaced to reflect the content, text, and terminology changes.
- Security requirements have been updated; in particular, the 80-bit security strength is no longer permitted in this Recommendation.
- Changes to handle the destruction of local keys and intermediate values have been introduced.
- General changes have been made to make this Recommendation more similar to [SP 800 56A].