INTERNATIONAL STANDARD

ISO/IEC 18367

First edition
2016-12-15

Information technology — Security techniques — Cryptographic algorithms and security mechanisms conformance testing

Technologie de l'information — Techniques de sécurité — Essais de conformité des algorithmes cryptographiques et des mécanismes de sécurité

Reference number
ISO/IEC 18367:2016(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword

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

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 27, *IT Security techniques.*

Introduction

This document describes cryptographic algorithms and security mechanisms conformance testing methods.

The purpose of this document is to address conformance testing methods of cryptographic algorithms and security mechanisms implemented in a cryptographic module. This will allow a complete security evaluation of both the cryptographic module and the implemented cryptographic algorithms and security mechanisms.

This document is related to ISO/IEC 19790 and ISO/IEC 24759. ISO/IEC 19790 specifies the security requirements for cryptographic modules. At a minimum, a cryptographic module implements at least one approved security function (i.e., cryptographic algorithm or security mechanism). ISO/IEC 24759 addresses the test requirements for each of the security requirements in ISO/IEC 19790. However, ISO/IEC 24759 does not address test methods for cryptographic algorithms and security mechanisms conformance testing.

Information technology — Security techniques — Cryptographic algorithms and security mechanisms conformance testing

1 Scope

This document gives guidelines for cryptographic algorithms and security mechanisms conformance testing methods.

Conformance testing assures that an implementation of a cryptographic algorithm or security mechanism is correct whether implemented in hardware, software or firmware. It also confirms that it runs correctly in a specific operating environment. Testing can consist of known-answer or Monte Carlo testing, or a combination of test methods. Testing can be performed on the actual implementation or modelled in a simulation environment.

This document does not include the efficiency of the algorithms or security mechanisms nor the intrinsic performance. This document focuses on the correctness of the implementation.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14888-3:2016, *Information technology — Security techniques — Digital signatures with appendix*

ISO/IEC 19790:2012, *Information technology — Security techniques — Security requirements for cryptographic modules*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 19790 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— IEC Electropedia: available at http://www.electropedia.org/

— ISO Online browsing platform: available at http://www.iso.org/obp

3.1
approval authority
any national or international organisation/authority mandated to approve and/or evaluate security functions

Note 1 to entry: An approval authority in the context of this definition evaluates and approves security functions based on their cryptographic or mathematical merits but is not the testing entity which would test for conformance to this document.

[SOURCE: ISO/IEC 19790:2012, 3.4]

3.2
approved mode of operation
set of services which includes non-security relevant services and at least one service that utilizes an approved security function or process

Note 1 to entry: Not to be confused with a specific mode of an approved security function, e.g. Cipher Block Chaining (CBC) mode.

Note 2 to entry: Non-approved security functions or processes are excluded.

[SOURCE: ISO/IEC 19790:2012, 3.7]

3.3
approved security function
security function (e.g. cryptographic algorithm) approved by an approval authority

3.4
black box
idealized mechanism that accepts inputs and produces outputs, but is designed such that an observer cannot see inside the box or determine exactly what is happening inside that box

Note 1 to entry: This term can be contrasted with *glass box* (3.12).

[SOURCE: ISO/IEC 18031:2011, 3.6]

3.5
critical security parameter
CSP
security-related information whose disclosure or modification can compromise the security of a cryptographic module

EXAMPLE Secret and private cryptographic keys, authentication data such as passwords, PINs, certificates or other trust anchors.

[SOURCE: ISO/IEC 19790:2012, 3.18, modified]

3.6
cryptographic algorithm
well-defined computational procedure that takes variable inputs, which may include cryptographic keys, and produces an output

[SOURCE: ISO/IEC 19790:2012, 3.20]

3.7
cryptographic algorithm boundary
boundary encompassing the complete cryptographic algorithm implementation

3.8
cryptographic boundary
explicitly defined continuous perimeter that establishes the physical and/or logical bounds of a cryptographic module and contains all the hardware, software, and/or firmware components of a cryptographic module

[SOURCE: ISO/IEC 19790:2012, 3.21, modified]

3.9
firmware
executable code of a cryptographic module that is stored in hardware within the cryptographic boundary and cannot be dynamically written or modified during execution while operating in a non-modifiable or limited operational environment

EXAMPLE Storage hardware can include but is not limited to PROM, EEPROM, FLASH, solid state memory, hard drives, etc.

[SOURCE: ISO/IEC 19790:2012, 3.45]

3.10
functional specification
high-level description of the ports and interfaces visible to the operator and high-level description of the behaviour of the IUT

Note 1 to entry: Here, the IUT means a cryptographic algorithm implementation (see 3.13).

[SOURCE: ISO/IEC 19790:2012, 3.47, modified]

3.11
functional testing
testing of the IUT functionality as defined by the *functional specification* (3.10)

[SOURCE: ISO/IEC 19790:2012, 3.48, modified]

3.12
glass box
idealized mechanism that accepts inputs and produces outputs and is designed such that an observer can see inside and determine exactly what is going on

Note 1 to entry: This term can be contrasted with black box (3.4).

[SOURCE: ISO/IEC 18031:2011, 3.14]

3.13
implementation under test
IUT
implementation which is tested for conformance to the selected cryptographic algorithm or security mechanism standard

3.14
independent verification test
test verifying the conformance for algorithms where their outputs are non-deterministic (or randomized) for defined input vectors in an independent way, instead of literally following the algorithm steps

Note 1 to entry: The signature generation function of DSA involves per-message secret number internally, and therefore, the resultant signature cannot be derived from the input vectors in a deterministic way without the knowledge of per-message secret number. Here, the signature verification function can be used to verify the relation between the public key, message and resultant signature, without knowing the per-message secret number.

Note 2 to entry: Independent verification tests can involve the reference implementation of the inverse function of the selected cryptographic algorithm, e.g. using the digital signature verification function to verify the corresponding signature generation function.

Note 3 to entry: In contrast to the other KATs, the independent verification test does not prepare expected values in advance.

EXAMPLE Applying Miller-Rabin primality test to verifying prime number generation functions, independent of the implementation details of the IUT.

3.15
key agreement
process of establishing a shared secret key between entities in such a way that neither of them can predetermine the value of that key

[SOURCE: ISO/IEC 11770-3:2015, 3.18, modified]

3.16
key derivation function
KDF
function that outputs one or more shared secrets, for use as keys, given shared secrets and other mutually known parameters as input

[SOURCE: ISO/IEC 11770-3:2015, 3.22]

3.17
key derivation key
key that is used as input to the key expansion function to derive other keys

3.18
key establishment
process of making available a shared secret key to one or more entities, where the process includes key agreement and key transport

[SOURCE: ISO/IEC 11770-3:2015, 3.23]

3.19
key expansion function
function which takes as input a number of parameters, at least one of which is a MAC algorithm key, and which gives as output keys appropriate for the intended algorithm and application, and which has the property that it is computationally infeasible to deduce either the output without prior knowledge of the secret input or the secret input from the output

3.20
key extraction function
function which takes as input a number of parameters, at least one of which is secret, which gives as output a MAC algorithm key for use as input to a key expansion function, and which has the property that it is computationally infeasible to deduce either the output without prior knowledge of the secret input or the secret input from the output

3.21
keying material
data necessary to establish and maintain cryptographic keying relationships

EXAMPLE Keys, initialization values.

[SOURCE: ISO/IEC 11770-1:2010, 2.27]

3.22
key management
administration and use of the generation, registration, certification, deregistration, distribution, installation, storage, archiving, revocation, derivation and destruction of keying material in accordance with a security policy

[SOURCE: ISO/IEC 19790:2012, 3.64]

3.23
key transport
process of transferring a key from one entity to another entity, suitably protected

[SOURCE: ISO/IEC 11770-3:2015, 3.25]

3.24
known-answer test
KAT
method of testing a deterministic mechanism where a given input is processed by the mechanism and the resulting output is then compared to a corresponding known value

Note 1 to entry: The known-answer tests are designed to test the conformance of the implementation under test (IUT) to the various specifications of the referenced cryptographic algorithm.

Note 2 to entry: A known-answer test is considered as a kind of black box testing.

[SOURCE: ISO/IEC 18031:2011, 3.21, modified]

3.25
Monte Carlo test
MCT
subset of known-answer test utilising randomly generated input vectors and the corresponding known output result, designed to pseudo exhaust the presence of flaws by exercising the entire IUT in a manner that cannot be detected with the controlled input vectors

Note 1 to entry: The types of implementation flaws which can be detected by Monte Carlo tests include pointer problems, insufficient allocation of space, improper error handling and incorrect behaviour of the IUT.

3.26
multi-block message test
MMT
set of tests designed to test the ability of the implementation to process multi-block messages which will require the chaining of information from one block to the next

3.27
public security parameter
PSP
security-related public information whose modification can compromise the security of a cryptographic module

EXAMPLE Public cryptographic keys, public key certificates, self-signed certificates, trust anchors, one-time passwords associated with a counter and internally held date and time.

[SOURCE: ISO/IEC 19790:2012, 3.99, modified]

3.28
random bit generator
RBG
device or algorithm that outputs a sequence of bits that appears to be statistically independent and unbiased

[SOURCE: ISO/IEC 18031:2011, 3.29]

3.29
salt
random data item produced by the signing entity during the generation of message representative

Note 1 to entry: Also known as the randomizer in ISO/IEC 14888-3.

Note 2 to entry: Also known as the per-message secret number in Reference [16].

3.30
security function
cryptographic algorithms together with modes of operation, such as block ciphers, stream ciphers, symmetric or asymmetric key algorithms, message authentication codes, hash functions or other security functions, random bit generators, entity authentication and SSP generation and establishment all approved either by ISO/IEC or an approval authority

[SOURCE: ISO/IEC 19790:2012, 3.106]

3.31
sensitive security parameter
SSP
critical security parameter (CSP) or public security parameter (PSP)

3.32
shared secret key
key which is shared with all the active entities via a key establishment mechanism for multiple entities

Note 1 to entry: Also known as "shared secret."

[SOURCE: ISO/IEC 11770-5:2011, 3.28, modified]

3.33
simulation
exercise of source code (e.g. VHDL code) prior to physical entry into the module (e.g. an FPGA or custom ASIC)

Note 1 to entry: The behaviour of the source code within the simulator can be logically identical when placed into the module or instantiated into logic gates. However, many other variables exist that can alter the actual behaviour (e.g., path delays, transformation errors, noise, environmental, etc.). It is not guaranteed that the actual behaviour of the IUT is identical, as many other variables cannot be identified with certainty.

3.34
zeroisation
method of destruction of stored data and unprotected SSPs to prevent retrieval and reuse

[SOURCE: ISO/IEC 19790:2012, 3.134]

4 Symbols and abbreviated terms

AES	advanced encryption standard
ASN.1	abstract syntax notation one
CBC	cipher block chaining mode of operation
CCM	counter mode with cipher block chaining-message authentication code
CFB	cipher feedback mode of operation
CMAC	cipher-based message authentication code
CSP	critical security parameter
DER	distinguished encoding rules
DRBG	deterministic random bit generator
DSA	digital signature algorithm

EAL	evaluation assurance level
ECB	electronic codebook mode of operation
ECDSA	elliptic-curve digital signature algorithm
FFC	finite field cryptography
GCM	galois/counter mode
GMAC	GCM message authentication code
HMAC	keyed-hash message authentication code
IUT	implementation under test
KAS	key agreement scheme
KAT	known-answer test
KC	key confirmation
KDF	key derivation function
MAC	message authentication code
MCT	Monte Carlo test
MMT	multi-block message test
OFB	output feedback mode of operation
PKCS	public key cryptography standards
PSS	probabilistic signature scheme
RBG	random bit generator
RSA	algorithm developed by Rivest, Shamir and Adleman
SHA	secure hash algorithm
TDEA	triple data encryption algorithm

$\lceil X \rceil$	ceiling: the smallest integer greater than or equal to X. For example, $\lceil 5 \rceil = 5$ and $\lceil 5.3 \rceil = 6$
$X \oplus Y$	bitwise exclusive-or (also bitwise addition mod 2) of two bit strings X and Y of the same length
X || Y	concatenation of two bit strings X and Y in that order
x mod n	unique remainder r, $0 \le r \le n-1$, when integer x is divided by n. For example, 23 mod 7 = 2

5 Objectives

The requirements specified in this document are derived from the following objectives for cryptographic algorithm implementations to

— provide assurance that the cryptographic algorithm implementation adheres to the specifications detailed in the associated cryptographic standard, and

— detect implementation non-conformities made by implementers by testing the algorithm's specifications, components, features and/or functionality for correctness and completeness.

6 Types of cryptographic algorithms and security mechanisms from a conformance testing perspective

6.1 General

This document will address approved security functions from a conformance testing point perspective. In particular, this document will address those defined in Clause 2. It will include within its scope the associated security mechanisms of the cryptographic algorithms or the security mechanisms. The considered implementations can be software, firmware, hardware or a combination thereof.

6.2 Asymmetric key algorithms

This subclause describes the different types of asymmetric key algorithms.

Asymmetric key algorithms consist of asymmetric key cryptographic primitive(s) and supporting functions. Some of asymmetric key algorithms are non-deterministic, due to salt or internally generated random numbers. The implementation of asymmetric key algorithms would contain more conditional branches than symmetric key algorithms. In considering these aspects of asymmetric key algorithms, the known-answer test (see 7.2.2) and/or independent verification test (see 3.14) are applicable.

In addition to these testing methodologies, other conformance testing methodologies (e.g. source code inspection) are also applicable.

6.3 Digital signature

This subclause describes the different types of digital signature algorithms.

The same perspective as asymmetric key algorithms is still applicable to digital signature algorithms.

6.4 Digital signature with message recovery

This subclause describes the different types of digital signature with message recovery algorithms.

The same perspective as asymmetric key algorithms is still applicable to digital signature with message recovery.

6.5 Hashing algorithms

This subclause describes the different types of hashing algorithms.

Hashing algorithms will be dedicated hash functions, functions based on block cipher algorithms or functions based on modular arithmetic. In considering the nature of hashing algorithms, the known-answer test (see 7.2.2) and Monte Carlo test (see 7.2.4) are applicable. Other conformance testing methodologies (e.g. source code inspection) are also applicable.

6.6 Key establishment mechanisms

This subclause describes the different types of key establishment mechanisms.

Key establishment mechanisms consist of asymmetric/symmetric key cryptographic primitive(s) and supporting functions. Supporting functions can be hashing algorithms, random bit generation, asymmetric key pair generation function, and public key validation function.

In considering the complex nature of key establishment mechanisms, the known-answer test (see 7.2.2) and independent verification test (see 3.14) are applicable. As a prerequisite for this conformance

testing, underlying algorithm implementations have passed the conformance testing elsewhere in this document.

Note that some input parameters are transmitted through a communication channel. It should be tested through the conformance testing that an IUT has an ability to distinguish valid parameters with invalid parameters.

In addition to these testing methodologies, other conformance testing methodologies (e.g. source code inspection) are also applicable.

6.7 Lightweight cryptography

This subclause describes the different types of lightweight cryptography algorithms.

Lightweight cryptography includes asymmetric key algorithms, block ciphers and stream ciphers. So, the applicable perspective is the same as one for asymmetric key algorithms, block ciphers and stream ciphers.

6.8 Message authentication algorithms

This subclause describes the different types of message authentication algorithms.

Message authentication algorithms can consist of underlying algorithms (e.g. block cipher algorithms, hash algorithms).

In considering this aspect of message authentication algorithms, the known-answer test (see 7.2.2) is applicable. As a prerequisite for this conformance testing, underlying algorithm implementations have passed the conformance testing elsewhere in this document.

In addition to this testing methodology, other conformance testing methodologies (e.g. source code inspection) are applicable.

6.9 Random bit generator algorithms

6.9.1 Deterministic random bit generator algorithms

This subclause describes the different types of deterministic random bit generator algorithms.

Deterministic random bit generators can consist of underlying algorithms (e.g. block cipher algorithms, hash algorithms).

In considering this aspect of deterministic random bit generator, the known-answer test (see 7.2.2) is applicable. As a prerequisite for this conformance testing, underlying algorithm implementations have passed the conformance testing elsewhere in this document.

In addition to this testing methodology, other conformance testing methodologies (e.g. source code inspection) are also applicable.

6.9.2 Non-deterministic random bit generator algorithms

This subclause describes the different types of non-deterministic random bit generator algorithms.

Currently, there is no standard specification of NRBG in ISO/IEC 18031, so the associated conformance testing methodologies are not specified in this document.

6.10 Symmetric key algorithms

6.10.1 Block cipher symmetric key algorithms

This subclause describes the different types of block cipher symmetric key algorithms.

Block cipher symmetric key algorithms support fixed input block and multiple input blocks with block cipher modes of operation. Block cipher symmetric key algorithms are used for various purposes and, therefore, the correctness of the algorithm implementations is important. In considering these aspects, known-answer tests (see 7.2.2), multi-block message test (see 7.2.3) and Monte Carlo test (see 7.2.4) are applicable. In addition to these testing methodologies, other conformance testing methodologies are also applicable.

6.10.2 Stream cipher symmetric key algorithms

This subclause describes the different types of stream cipher symmetric key algorithms.

Stream cipher symmetric key algorithms are similar to block cipher symmetric key algorithms in their structures.

In considering this aspect of stream cipher algorithms, known-answer tests (see 7.2.2) and Monte Carlo test (see 7.2.4) are applicable.

In addition to these testing methodologies, other conformance testing methodologies are also applicable.

7 Conformance testing methodologies

7.1 Overview

For cryptographic algorithms which do not introduce randomness, known-answer test(s), Monte Carlo and multi-block message tests shall be applied. For cryptographic algorithms which utilize internally generated random numbers to generate output, independent verification test(s) shall be applied.

EXAMPLE Digital signature scheme 2 in ISO/IEC 9796-2.

Note that neither the known-answer test, Monte Carlo test, multi-block message test, nor the independent verification test is universal. In order to complement these tests, source code review may be applied.

Conformance testing provides tests to determine the correctness of the algorithm implementation and implementation errors including pointer problems, insufficient allocation of space, improper error handling and incorrect behaviour of the algorithm implementation. The validation tests are designed to assist in the detection of accidental implementation errors and are not designed to detect intentional attempts to misrepresent conformance. Conformance testing should not be interpreted as an evaluation or endorsement of overall product security. Conformance testing shall utilize statistical sampling (i.e. only a small number of the possible cases are tested); hence, the successful conformance testing of a device does not imply 100 % correctness of conformance.

NOTE 1 Implementation errors in hardware implementations include timing delay.

NOTE 2 Implementation errors in software/firmware implementations include pointer problems, insufficient allocation of space, improper error handling and incorrect behaviour of the algorithm implementation.

The different types of testing methodologies are described in 7.2 and 7.3.

7.2 Black box testing

7.2.1 General

Black box validation testing is where the functionality of an implementation is examined without peering into the implementation's actual code or internal workings. The given inputs exercise the implementation to assure the specifications of the standard are implemented correctly. This is determined by comparing the output produced by the implementation under test (IUT) with the expected outputs. Black box testing can be used when a reference implementation or simulator is available.

7.2.2 Known-answer test vectors

The known-answer tests are designed to verify the individual components of the specifications of the cryptographic algorithm. The tests exercise each bit of every component of the algorithm implementation. Known-answer tests use specific pre-determined input test vectors and the computation of the expected results to target the specific functionality of a security function.

7.2.3 Multi-block message testing

The Multi-block message test (MMT) is designed to test the ability of an implementation to process multi-block messages, which may require chaining of information from one block to the next.

7.2.4 Monte Carlo or statistical testing

Monte Carlo testing is a statistical method which is designed to exercise the entire implementation of the cryptographic algorithm being implemented. The purpose is to detect the presence of flaws in the IUT that were not detected with the controlled input of the known-answer test. The Monte Carlo test does not guarantee ultimate reliability of the IUT that implements the algorithm (e.g. hardware failure, software corruption, etc.). A predetermined number of pseudorandom values are used as input values to test the algorithmic implementation. Using these values, the IUT is exercised through the complete implementation. The results are then compared with the expected values.

7.3 Glass box or white box testing

7.3.1 Source code inspection

7.3.1.1 Overview

In contrast to black box testing such as KAT, source code inspection is capable of wide application, which will be useful in verifying the internal behaviour and rare case handling. However, it needs knowledge of description language and reviewing time. So, the source code inspection should be complementary to other testing approaches.

7.3.2 Binary analysis

Even if the source code itself is consistent with the specification of a selected algorithm, the behaviour of resultant binary (or executable code) might be different from the original source code, e.g. due to the software building tools. In such a case, binary analysis should be performed in order to ensure the conformance. In the case of software or firmware implementations, this can be performed by inspecting the generated assembly output to assure it is consistent with the original source code (e.g. in Mixed Mode).

EXAMPLE Compiler, code interpreter.

NOTE Some cryptographic algorithm specifications include a step to zeroise secret information. However, such zeroisation steps might be deleted from the resultant binary due to the optimization or decision made by a compiler.

8 Levels of conformance testing

8.1 Introduction

The overview of two levels of conformance testing is described in 8.2 and 8.3.

8.2 Level of basic conformance testing

In this level, the conformity is tested by exercising IUTs from outside the cryptographic algorithm boundary. This level will commensurate with security level 1 in ISO/IEC 19790 or EAL3 in ISO/IEC 15408.

The tester is required to test the IUT by considering implementation errors listed in 7.1.

8.3 Level of moderate conformance

This level enhances the level of basic conformance by enforcing informal security policy. In this level, the conformity is tested by inspecting the IUTs also from inside the cryptographic algorithm boundary with focus on the security policy enforcement. This level will commensurate with security levels 2 and 3 in ISO/IEC 19790 or EAL4 in ISO/IEC 15408.

NOTE There can exist a level of conformance in which the formal security policy or formal verification is applied. However, this document does not define conformance tests covering this level.

9 Conformance testing guidelines

9.1 General guidelines

9.1.1 Identification

The tester shall identify the following information on the IUT:

a) tested platform or configuration;

EXAMPLE 1 Operational environment for software, including test harness.

EXAMPLE 2 When the virtualization technology is used for the operational environment, configuration parameters for the virtual machine.

The vendor shall identify the following information on the IUT:

b) reference to the specification of cryptographic algorithm implemented;

c) security function(s);

EXAMPLE 3 Digital signature generation function.

EXAMPLE 4 Encryption function of block ciphers.

EXAMPLE 5 Two-step key derivation function.

d) supporting functions;

EXAMPLE 6 Hash functions in digital signatures.

EXAMPLE 7 Entropy conditioning functions in random bit generators.

EXAMPLE 8 Public key validation function in key establishment scheme.

EXAMPLE 9 Key derivation functions in key establishment scheme.

1) If the conformance testing for the supporting function is available, the tester shall verify that the supporting function implementation has passed the conformance testing.

e) supported parameters range;

EXAMPLE 10 Bit length of key of block cipher.

EXAMPLE 11 Bit length of key of MAC.

EXAMPLE 12 Bit length of input message for hash functions.

EXAMPLE 13 Bit length of input message to be encrypted using asymmetric ciphers.

f) data types and representation of parameters;

EXAMPLE 14 Most significant octet first.

EXAMPLE 15 Point compression of elliptic curve points.

1) The data types and format of input and output parameters shall be precise.

g) architecture or functional decomposition.

EXAMPLE 16 A digital signature algorithm implementation using a hashing algorithm implemented in firmware and a modular arithmetic coprocessor implemented in hardware.

EXAMPLE 17 An AES implementation using cryptographic supporting instructions.

9.1.2 Guidelines for black box testing

Some cryptographic algorithms use multiple-precision integer. Assuming that the multiple-precision integer is represented by octets, the carry bit handling will occur normally once in 256 test vectors. This also applies to the implementation of a counter. For such an implementation, at least 512 test vectors should be used to perform black box testing.

9.1.3 Guidelines for white box testing

9.1.3.1 Guidelines for source code inspection

9.1.3.1.1 General

Under the condition that the IUT is tested by black box testing and found to be functionally conformant, source code inspection may be applied to claim conformance of one level higher.

Source code inspection should be used to check the following aspects:

a) security feature which cannot be tested easily from external;

1) zeroisation,

EXAMPLE Zeroisation of shared secret key, ephemeral key, or subkeys

2) call of underlying cryptographic algorithm(s),

NOTE 1 SHA can be an underlying algorithm of HMAC.

NOTE 2 Random number generation becomes an underlying algorithm for DSA, ECDSA, and RSASSA-PSS signature generation function.

NOTE 3 If the IUT includes two or more implementations of the same cryptographic algorithm, it would be difficult to identify which implementation is used by black box testing from external. If such implementations are used as underlying algorithms, source code inspection enables us to identify which implementation is used. This also assures the chain of tested algorithm implementations.

3) health tests,

NOTE 4 Health tests are also called self-tests (see ISO/IEC 19790:2012, 3.108).

NOTE 5 There are health test requirements in ISO/IEC 18031.

b) branch condition;

c) status output;

d) source code which is rarely executed, but security relevant.

The tester should review the selected cryptographic algorithm specification to check whether the above examples are applicable to the IUT or not. If one or more examples are found to be applicable, the tester should apply source code inspection to verify that the IUT is conformant to the selected cryptographic algorithm specification. The tester shall not be misguided by source code annotation, if any.

Some IUTs support only limited input parameters range and/or limited functions, which are narrower than that of the original cryptographic algorithm specification. Such limited input parameters range and/or functions shall be recognised by the validation authority. The tester shall perform source code inspection by considering the claimed input parameters range, and verify that the source code is the correct embodiment of the cryptographic algorithm.

In the source code, many variables will be introduced. The tester should identify conversion between variables in the cryptographic algorithm specification and variables in the source code.

9.1.3.1.2 Security features which cannot be tested easily

9.1.3.1.2.1 Zeroisation

In general, there are several types of memory to which IUTs access, and IUTs can generate some copies of critical security parameters and/or their derivatives in the course of their operations.

The tester shall verify that all of the instances of CSPs and their derivatives are zeroised in both cases

a) when the cryptographic algorithm is completed successfully if they are no longer used, and

b) when the cryptographic algorithm terminates with an error indicator.

In order to perform source code inspection for zeroisation, the tester should identify types of memory on which any copies of critical security parameters or its derivatives reside. The tester should identify registers, variables, data structures, objects and/or pieces of memory which store any CSP.

EXAMPLE 1 Registers, cache and RAM.

EXAMPLE 2 The internal state of RBG.

NOTE 1 In ISO/IEC 19790:2012, 7.9.7, it is required to zeroise temporary SSPs when they are no longer needed, for security level 2 and higher.

The tester shall identify applied zeroisation techniques to the IUT. If the IUT is to be used in a cryptographic module, the tester shall verify that the applied zeroisation techniques meet sensitive security parameter zeroisation requirements in ISO/IEC 19790:2012, 7.9.7.

NOTE 2 The zeroisation requirements differ by security levels in ISO/IEC 19790.

9.1.3.1.2.2 Call of underlying cryptographic algorithms

Some cryptographic algorithm specifications need underlying algorithms. The tester shall identify the underlying cryptographic algorithms by reviewing the cryptographic algorithm specification. The tester shall also identify the underlying cryptographic algorithm implementations by reviewing the source code. The tester shall verify, by performing source code inspection, that the underlying cryptographic algorithms implementation utilized meets the requirements of the cryptographic algorithm specification.

9.1.3.1.2.3 Health tests

Some cryptographic algorithm specifications request IUTs to implement and perform health tests.

The tester shall identify health test requirements from the selected specification and perform the source code inspection, in order to verify that the required health tests are really implemented. Some of the health test requirements can be purposely designed to test a) a specific part of cryptographic algorithm and/or b) a specific ability. Through the identification of selected cryptographic algorithms using guidelines in Clause 9, the tester shall verify that the heath test requirements are met in contrast to those selected cryptographic algorithm configurations.

However, the invocation of heath tests might not be a matter of IUTs, but the matter of cryptographic modules (see ISO/IEC 19790:2012, 7.10.3.2) or of security architecture under ISO/IEC 15408 (see ADV_ARC.1-5 in ISO/IEC 18045). For the latter case, through the cryptographic module testing, the tester should confirm that those heath tests are really invoked as specified in the selected cryptographic algorithm specification.

9.1.3.1.3 Branch condition

There will be branch conditions in cryptographic algorithm specifications. The tester shall identify targeted branch conditions by reviewing not only the cryptographic algorithm specification but also the source code.

NOTE 1 IUTs can include implementation-dependent branch conditions which are not identified in the cryptographic algorithm specification.

EXAMPLE Carry bit handling.

The branch condition will be expressed using a comparison operator. Note that the branch condition is not always directly translated into source code. The tester shall verify that the source code is arithmetically equivalent to the identified branch condition.

NOTE 2 Branch conditions are not always directly translated into source code. For example, if the statement is expressed as (A > 127) by integer A, another statement (A ≥ 128) is arithmetically equivalent.

9.1.3.1.4 Status output

Some cryptographic algorithms define one or more status output values, e.g. composite or probably prime for primality test. However, such status output might not be observed in the black box testing. The tester should perform source code inspection to verify that each distinct status output in the specification is mapped to the different value in the IUT.

NOTE 1 ADV_FSP.4-8 in ISO/IEC 18045.

NOTE 2 ADV_TDS.3-10 in ISO/IEC 18045.

NOTE 3 ADV_IMP.1-3 in ISO/IEC 18045.

9.1.3.1.5 Source code rarely executed

Currently, there are no dedicated guidelines for source code that is rarely executed. However, guidelines in 9.1.3.1.4 should be followed if a status output is included in the source code that is rarely executed.

9.1.3.2 Guidelines for binary analysis

Under the condition that the source code of the IUT is inspected to be conformant, binary analysis should be performed to confirm that the final implementation representation corresponds to the original source code.

9.2 Guidelines specific to encryption algorithms

9.2.1 Identification of encryption algorithms

9.2.1.1 Identification of block ciphers

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) bit length of key supported;

b) bit length of input block supported.

In many cases, implementations of symmetric key algorithms support a block cipher mode of operation in order to process multiple blocks.

If the IUT supports a block cipher mode of operation, the vendor shall identify the following information on the IUT:

a) block cipher mode of operation;

b) bit length of initialization vector;

c) maximum number of blocks supported.

If a counter mode is supported by the IUT, the vendor shall identify the following information on the IUT:

d) how the counter is constructed, including its format

9.2.1.2 Identification of stream ciphers

The vendor shall identify the following information on the implementation of stream cipher:

a) reference to the specification of stream cipher;

b) bit length of key supported;

c) bit length of initialization vector, if any.

9.2.1.3 Identification of asymmetric ciphers

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) consistency between security function, supporting functions and supported parameters;

b) bit length of each public key component;

c) bit length of each private key component.

9.2.2 Selecting a set of conformance test items

9.2.2.1 Block ciphers

A block cipher sometimes becomes the underlying algorithm of higher-level algorithms, e.g. MAC and RBG. In this sense, implementations of symmetric key algorithms should be extensively tested.

Taking into account the above situation, the following five basic black box test items can be applicable to block ciphers:

a) KAT-Text, which supplies various plaintexts (or ciphertexts) to the IUT and tests if the resultant ciphertexts (or plaintexts) match the expected values, while using a fixed key;

b) KAT-Key, which uses various keys in the IUT and tests if the resultant ciphertexts (or plaintexts) match the expected values, while using fixed plaintexts (or ciphertexts);

c) KAT-Sbox, which utilizes all of the entries in the Sbox and tests if the resultant ciphertexts (or plaintexts) match the expected values;

d) MMT;

e) MCT.

In addition to the above test items, the following optional white box tests can be applicable to symmetric key algorithms:

f) source code inspection;

g) binary analysis.

9.2.2.2 Stream ciphers

The construction of stream cipher algorithms will be different from that of block ciphers; therefore, the same test items as block ciphers cannot always be applicable to stream ciphers.

Taking into account the above situation, the following two basic black box test items can be applicable to stream ciphers:

a) KAT-Key, which uses various keys in the IUT and tests if the resultant ciphertexts (or plaintexts) match the expected values, while using fixed plaintexts (or ciphertexts);

b) MCT.

In addition to the above test items, the following optional white box tests can be applicable to symmetric key algorithms:

c) source code inspection;

d) binary analysis.

9.2.2.3 Asymmetric ciphers

Before selecting a set of conformance test items, the tester shall verify that the conformance to the underlying algorithms in asymmetric ciphers is already met. Therefore, it is assumed that the underlying algorithms are extensively tested. The following KAT or independent verification test should be selected:

a) Random test, which uses various information needed by the IUT and tests:

   1) if the resultant ciphertext matches the expected value for the encryption function, or that the resultant ciphertext can be decrypted by a reference implementation,

2) if the resultant plaintext matches the expected plaintext for the decryption function.

In addition to the above test items, the following optional white box tests can be applicable to asymmetric ciphers:

a) source code inspection;

b) binary analysis.

9.2.3 Guidelines for each conformance test item

9.2.3.1 Guidelines for black box testing

9.2.3.1.1 Guidelines for KAT-Text

In the KAT-Text, the tester shall supply various plaintexts (or ciphertexts) to the IUT and test if the resultant ciphertexts (or plaintexts) match the expected values, while using a fixed key.

Let n be the number of bits of plaintext (or ciphertext) block; the tester shall supply at least n different plaintexts (or ciphertexts) to the IUT by changing every single bit in the one block.

EXAMPLE 128 plaintexts for 128-bit block ciphers.

For example, let PT_j be the jth entry in plaintexts supplied to the IUT, and let (b_1, b_2, ..., b_n) be the bits of PT_j from leftmost to rightmost. The set of plaintexts constructed based on the following rules will meet the guidelines:

$$b_i = \begin{cases} 1(i \in \{1, 2, \ldots, j\}) \\ 0(i \in \{j+1, \ldots, n\}) \end{cases}$$

9.2.3.1.2 Guidelines for KAT-Key

In the KAT-Key, the tester shall send various keys to the IUT and test if the resultant ciphertexts (or plaintexts) match the expected values, while using fixed plaintext (or ciphertext) values.

Let k be the number of significant bits of key. The tester shall supply different keys to the IUT at least k times, by changing every single bit in the key.

NOTE For TDEA, the number of bits of key is not equal to the number of significant bits of key, due to parity bits in the key.

For example, let K_j be the jth entry in keys sent to the IUT, and let (b_1, b_2, ..., b_k) be the bits of K_j from leftmost to rightmost. The set of keys constructed based on the following rules will meet the guidelines:

$$b_i = \begin{cases} 1(i \in \{1, 2, \ldots, j\}) \\ 0(i \in \{j+1, \ldots, k\}) \end{cases}$$

9.2.3.1.3 Guidelines for KAT-Sbox

In the KAT-Sbox, the tester shall supply the sets of test vectors and demonstrate that all of the entries in Sbox are used in processing these test vectors.

Let m be the number of entries in Sbox. The tester shall supply m sets of test vectors for KAT-Sbox.

9.2.3.1.4 Guidelines for MMT

In the MMT, the tester shall supply various plaintexts (or ciphertexts) to the IUT and test if the resultant ciphertexts (or plaintexts) match the expected values.

The variable and function used in the description of MMT are:

nb The maximum number of blocks supported by the IUT.

min(*A*, *B*) A function that returns either *A* or *B*, whichever is less.

Except when the IUT supports only one block processing or the block cipher mode of operation is ECB, or CTR, the tester shall supply up to **min**(10, *nb*) blocks of plaintexts (or ciphertexts) to the IUT and test if the resultant ciphertexts (or plaintexts) match the expected values using a reference implementation.

In ISO/IEC 10116, the generation of counter block is based on an incremented counter [i.e. $x \leftarrow (x + 1) \bmod 2^n$, where n is the number of bits of plaintext (or ciphertext) per block]. Therefore, the above MMT can be applied to the IUT which supports ISO/IEC 10116 compliant CTR mode.

Modes of operation for an n-bit block cipher specified in ISO/IEC 10116 only use exclusive-or and is not as complex; therefore, extensive testing is not needed.

9.2.3.1.5 Guidelines for MCT

9.2.3.1.5.1 General

In the MCT, the tester shall choose one set of initial values for keys, plaintexts (or ciphertexts) and initialization vectors, randomly. The variables and functions used in the description of MCT are:

CT, CT$_j$ A ciphertext at each iteration *j*.

CTR$_j$ A counter used in CTR mode.

$\boldsymbol{D}_K$ The decryption function with key *K*.

$\boldsymbol{E}_K$ The encryption function with key *K*.

f, g Temporary integers.

i A temporary value used as a loop counter.

INIT_CT An intermediate value for OFB mode.

INIT_PT An intermediate value for OFB mode.

inner_loops The number of inner loops.

IV An initialization vector.

NOTE In ISO/IEC 10116, this variable is called the “starting variable.”

IV$_i$ An initialization vector at each iteration *i* in CFB mode.

j A temporary value used as a loop counter.

K The key used to encrypt or decrypt.

K$_1$, *K*$_2$, *K*$_3$ The three keys for TDEA.

Len(*x*) A function that returns the number of bits in input string *x*.

LSB$_s$(*x*) The bit string consisting of the *s* least significant bits of the bit string *x*.

MSB$_s$(*x*) The bit string consisting of the *s* most significant bits of the bit string *x*.

n The number of bits of plaintext (or ciphertext) per block of block cipher.

outer_loops	The number of outer loops. This is identical to the number of resultant ciphertexts or plaintexts.
PT, PT_j	A plaintext at each iteration *j*.
PT_0	The initial plaintext for *j*-loop.
s	The number of bits of plaintext (or ciphertext) block for CFB mode.
Truncate(bits, in_len, out_len**)**	A function that inputs a bit string *bits* of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.
x, y	Intermediate values.

9.2.3.1.5.2 MCT for encryption function

The general framework of MCT for encryption function is described in the following steps.

a) Assign an initial value for the initial key, plaintext and initialization vector *IV*.

NOTE 1 For CTR mode, the initialization vector will be replaced with counter.

b) For *i* = 1 to *outer_loops*, do:

1) Prepare intermediate values for next *i*-value.

NOTE 2 This step will be tailored based on the block cipher mode of operation.

2) For *j* = 1 to *inner_loops,* do:

i) Prepare intermediate values for next *j*-value.

ii) Generate ciphertext CT_j using key *K*, plaintext *PT* and initialization vector *IV*.

iii) Update plaintext *PT*, and initialization vector *IV*, by using the ciphertext CT_j obtained.

NOTE 3 This step will be tailored based on the nature of the block cipher algorithm and the mode of operation tested.

NOTE 4 Updating *PT* and *IV* can be done by applying bitwise exclusive-or and/or concatenation.

3) Output ciphertext CT_j.

4) Update key *K* by using ciphertexts $\{CT_j\}$.

NOTE 5 This step will be tailored based on the nature of block cipher algorithm.

NOTE 6 Updating the key can be done by applying bitwise exclusive-or and/or concatenation.

From the above steps, *outer_loops* sets of ciphertexts are obtained.

9.2.3.1.5.3 Tailoring MCT for encryption function

The tailored steps in 9.2.3.1.5.2 are listed in Tables 1 to 5, for each block cipher mode of operation.

NOTE 1 In the tailoring, Reference [17] is considered for TDEA, and Reference [18] is considered for the other block ciphers listed in ISO/IEC 18033-3.

NOTE 2 In the tailoring, it is considered that only 128-bit key is used in 64-bit block ciphers listed in ISO/IEC 18033-3, except for TDEA.

Table 1 — Tailored steps in MCT for ECB mode encryption

Step number	Tailored steps for ECB mode encryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	$CT_j = \boldsymbol{E}_K(PT)$. [a]	←	←
b) - 2) - iii)	$PT = CT_j$.	←	←
b) - 4)	If ($\boldsymbol{Len}(K) = 128$), then $K = K \oplus CT_{inner\_loops}$ Else if ($\boldsymbol{Len}(K) = 192$), then $K = K \oplus (\boldsymbol{LSB}_{64}(CT_{(inner\_loops-1)}) \| CT_{inner\_loops})$ Else if ($\boldsymbol{Len}(K) = 256$), then $K = K \oplus (CT_{(inner\_loops-1)} \| CT_{inner\_loops})$ Else terminate MCT.	$K = K \oplus (CT_{(inner\_loops-1)} \| CT_{inner\_loops})$.	i) $K_1 = K_1 \oplus CT_{(inner\_loops-2)}$, ii) $K_2 = K_2 \oplus CT_{(inner\_loops-1)}$, iii) $K_3 = K_3 \oplus CT_{inner\_loops}$.

[a] This step corresponds to ECB mode encryption.

Table 2 — Tailored steps in MCT for CBC mode encryption

Step number	Tailored steps for CBC mode encryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	$CT_j = \boldsymbol{E}_K(PT \oplus IV)$. [a]	←	←
b) - 2) - iii)	I) If ($j = 1$), then $PT = IV$ Else $PT = CT_{(j-1)}$ II) $IV = CT_j$.	←	←
b) - 4)	If ($\boldsymbol{Len}(K) = 128$), then $K = K \oplus CT_{inner\_loops}$ Else if ($\boldsymbol{Len}(K) = 192$), then $K = K \oplus \boldsymbol{LSB}_{64}(CT_{(inner\_loops-1)}) \| CT_{inner\_loops}$ Else if ($\boldsymbol{Len}(K) = 256$), then $K = K \oplus (CT_{(inner\_loops-1)} \| CT_{inner\_loops})$ Else terminate MCT.	$K = K \oplus (CT_{(inner\_loops-1)} \| CT_{inner\_loops})$.	i) $K_1 = K_1 \oplus CT_{(inner\_loops-2)}$, ii) $K_2 = K_2 \oplus CT_{(inner\_loops-1)}$, iii) $K_3 = K_3 \oplus CT_{inner\_loops}$.

[a] This step corresponds to CBC mode encryption.

Table 3 — Tailored steps in MCT for CFB mode encryption

Step number	Tailored steps for CFB mode encryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	$IV = IV_i$.	←	no operation
b) - 2) - ii)	I) $y = \boldsymbol{E}_K(IV)$. [a] [b] II) $CT_j = PT \oplus \boldsymbol{MSB}_s(y)$. [a] [b] [c]	←	←

b) - 2) - iii)	If ($s = n$), then
	{
	If ($j = 1$), then
	$PT = IV$,
	Else
	$PT = CT_{(j-1)}$.
	$IV = CT_j$. [a]
	}
	Else
	{
	$IV = \boldsymbol{LSB}_{n-s}(IV) \| CT_j$. [a]
	If $\left(j \le \lceil n / s \rceil\right)$, then
	$PT = \boldsymbol{MSB}_s(IV_i)$,
	Else
	$PT = CT_{\left(j-\lceil n/s \rceil\right)}$.
	}
		←	I) $PT = \boldsymbol{MSB}_s(IV)$.
			II) $IV = \boldsymbol{LSB}_{n-s}(IV) \| CT_j$. [a]

b) - 4)	i) $f = \lceil \boldsymbol{Len}(K) / s \rceil$. ii) $C = CT_{(inner\_loops-f+1)} \| \ldots \| CT_{inner\_loops}$. iii) $K = K \oplus \boldsymbol{LSB}_{\boldsymbol{Len}(K)}(C)$. iv) $g = \lceil \boldsymbol{Len}(IV_i) / s \rceil$ v) $IV_{(i+1)} =$ $\boldsymbol{LSB}_n(CT_{(inner\_loops-g+1)} \| \ldots \| CT_{inner\_loops})$.	←	i) $f = \lceil 192 / s \rceil$. ii) $C = CT_{(inner\_loops-f+1)} \| \ldots \| CT_{inner\_loops}$. ii) $K_1 = K_1 \oplus \boldsymbol{LSB}_{64}(C)$. ii) $K_2 = K_2 \oplus \textbf{Truncate}(\boldsymbol{LSB}_{128}(C), 128, 64)$ iii) $K_3 = K_3 \oplus \boldsymbol{MSB}_{64}(C)$.

[a] This step constitutes a part of CFB mode encryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

[c] PT and CT_j are assumed to be s bits in length.

Table 4 — Tailored steps in MCT for OFB mode encryption

Step number	Tailored steps for OFB mode encryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	$INIT\_PT = PT_0$
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	I) $y = \boldsymbol{E}_K(IV)$. [a] [b] II) $CT_j = PT \oplus y$. [a] [b]	←	←
b) - 2) - iii)	I) If (j = 1), then $PT = IV$ Else $PT = CT_{(j-1)}$. II) $IV = y$. [a] [b]	←	I) $PT = IV$. II) $IV = y$. [a] [b]
b) - 4)	i) If ($\boldsymbol{Len}$ (K) = 128), then $K = K \oplus CT_{inner\_loops}$ Else if ($\boldsymbol{Len}$(K) = 192), then $K = K \oplus \boldsymbol{LSB}_{64}(CT_{(inner\_loops-1)}) \,\|\, CT_{inner\_loops}$ Else if ($\boldsymbol{Len}$(K) = 256), then $K = K \oplus (CT_{(inner\_loops-1)} \,\|\, CT_{inner\_loops})$ Else terminate MCT. ii) $IV = CT_{inner\_loops}$.	i) $K = K \oplus (CT_{(inner\_loops-1)} \,\|\, CT_{inner\_loops})$. ii) $IV = CT_{inner\_loops}$.	i) $K_1 = K_1 \oplus CT_{(inner\_loops-2)}$, ii) $K_2 = K_2 \oplus CT_{(inner\_loops-1)}$, iii) $K_3 = K_3 \oplus CT_{inner\_loops}$. iv) $PT_0 = PT_0 \oplus INIT\_PT$.

[a] This step constitutes a part of OFB mode encryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

Table 5 — Tailored steps in MCT for CTR mode encryption

Step number	Tailored steps for CTR mode encryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	I) $y = \boldsymbol{E}_K(CTR_j)$. [a] [b] II) $CT_j = PT \oplus y$. [a] [b]	←	←
b) - 2) - iii)	I) $CTR_j = (CTR_j + 1) \bmod 2^n$. [c] II) $PT = CT_j$.	←	←
b) - 4)	If ($\boldsymbol{Len}$ (K) = 128), then $K = K \oplus CT_{inner\_loops}$ Else if ($\boldsymbol{Len}$(K) = 192), then $K = K \oplus \boldsymbol{LSB}_{64}(CT_{(inner\_loops-1)}) \,\|\, CT_{inner\_loops}$ Else if ($\boldsymbol{Len}$(K) = 256), then $K = K \oplus (CT_{(inner\_loops-1)} \,\|\, CT_{inner\_loops})$ Else terminate MCT.	$K = K \oplus (CT_{(inner\_loops-1)} \,\|\, CT_{inner\_loops})$.	i) $K_1 = K_1 \oplus CT_{(inner\_loops-2)}$, ii) $K_2 = K_2 \oplus CT_{(inner\_loops-1)}$, iii) $K_3 = K_3 \oplus CT_{inner\_loops}$.

[a] This step constitutes a part of CTR mode encryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

[c] This step corresponds to the counter generation.

9.2.3.1.5.4 MCT for decryption function

The general framework of MCT for the decryption function is described by the following steps:

a) Assign initial value for initial key, ciphertext and initialization vector *IV*.

NOTE 1 For CTR mode, the initialization vector will be replaced with counter.

b) For $i = 1$ to *outer_loops*, do:

1) Prepare intermediate values for next *i*-value.

NOTE 2 This step will be tailored based on the block cipher mode of operation.

2) For $j = 1$ to *inner_loops*, do:

i) Prepare intermediate values for the next *j*-value.

ii) Generate plaintext PT_j using key *K*, ciphertext *CT*, and initialization vector *IV*.

iii) Update ciphertext *CT*, and initialization vector *IV*, by using the plaintext PT_j obtained.

NOTE 3 This step will be tailored based on the nature of block cipher algorithm and mode of operation tested.

NOTE 4 Updating PT_j and *IV* can be done by applying bitwise exclusive-or and/or concatenation.

3) Output plaintext PT_j.

4) Update key *K* by using plaintexts $\{PT_j\}$.

NOTE 5 This step will be tailored based on the nature of block cipher algorithm.

NOTE 6 Updating the key can be done by applying bitwise exclusive-or and/or concatenation.

From the above steps, the output consists of *outer_loops* plaintexts.

9.2.3.1.5.5 Tailoring MCT for decryption function

The tailored steps in 9.2.3.1.5.4 are listed in Tables 6 to 10, for each block cipher mode of operation.

NOTE 1 In the tailoring, Reference [17] is considered for TDEA, and Reference [18] is considered for the other block ciphers listed in ISO/IEC 18033-3.

NOTE 2 In the tailoring, it is considered that only 128-bit key is used in 64-bit block ciphers listed in ISO/IEC 18033-3, except for TDEA.

Table 6 — Tailored steps in MCT for ECB mode decryption

Step number	Tailored steps for ECB mode decryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation		
b) - 2) - i)	no operation		
b) - 2) - ii)	$PT_j = \boldsymbol{D}_K(CT)$. [a]		
b) - 2) - iii)	$CT = PT_j$.		
b) - 4)	If (***Len*** $(K) = 128$), then $K = K \oplus PT_{inner\_loops}$ Else if (***Len***$(K) = 192$), then $K = K \oplus \boldsymbol{LSB}_{64}(PT_{(inner\_loops\text{-}1)}) \,\|\| PT_{inner\_loops}$ Else if (***Len***$(K) = 256$), then $K = K \oplus (PT_{(inner\_loops\text{-}1)} \,\|\| PT_{inner\_loops})$ Else terminate MCT.	$K = K \oplus (PT_{(inner\_loops\text{-}1)} \,\|\| PT_{inner\_loops})$.	i) $K_1 = K_1 \oplus PT_{(inner\_loops\text{-}2)}$, ii) $K_2 = K_2 \oplus PT_{(inner\_loops\text{-}1)}$, iii) $K_3 = K_3 \oplus PT_{inner\_loops}$.
[a] This step corresponds to ECB mode decryption.			

Table 7 — Tailored steps in MCT for CBC mode decryption

Step number	Tailored steps for CBC mode decryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	$PT_j = \boldsymbol{D}_K(CT) \oplus IV$. [a]	←	←

b) - 2) - iii)	If ($j = 1$), then
	$y = CT$, [a]
	$CT = IV$,
	$IV = y$, [a]
	Else
	$IV = CT$, [a]
	$CT = PT_{(j-1)}$.
		←	$IV = CT$, [a]
			$CT = PT_j$.

b) - 4)	If (***Len*** $(K) = 128$), then $K = K \oplus PT_{inner\_loops}$ Else if (***Len***$(K) = 192$), then $K = K \oplus \boldsymbol{LSB}_{64}(PT_{(inner\_loops\text{-}1)}) \,\|\| PT_{inner\_loops}$ Else if (***Len***$(K) = 256$), then $K = K \oplus (PT_{(inner\_loops\text{-}1)} \,\|\| PT_{inner\_loops})$ Else terminate MCT.	$K = K \oplus (PT_{(inner\_loops\text{-}1)} \,\|\| PT_{inner\_loops})$.	i) $K_1 = K_1 \oplus PT_{(inner\_loops\text{-}2)}$, ii) $K_2 = K_2 \oplus PT_{(inner\_loops\text{-}1)}$, iii) $K_3 = K_3 \oplus PT_{inner\_loops}$.
[a] This step corresponds to CBC mode decryption.			

Table 8 — Tailored steps in MCT for CFB mode decryption

Step number	Tailored steps for CFB mode decryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	no operation
b) - 2) - ii)	I) $y = E_K(IV)$. [a] [b] II) $PT_j = CT \oplus \boldsymbol{MSB}_s(y)$. [a] [b] [c]	←	←

b) - 2) - iii)	If ($s = n$), then
	{
	If ($j = 1$), then
	$x = IV$,
	$IV = CT$,
	$CT = x$,
	Else
	$IV = CT$. [a]
	$CT = PT_{(j-1)}$.
	}
	Else
	{
	$IV = \boldsymbol{LSB}_{n-s}(IV) \| CT$. [a]
	If , then
	$CT = \boldsymbol{MSB}_s(IV_i)$,
	Else
	$C = PT_{\left(j-\lceil n/s \rceil\right)}$
	}
		←	I) $IV = \boldsymbol{LSB}_{n-s}(IV) \| CT$. [a]
			II) $CT = \boldsymbol{MSB}_s(y)$.

b) - 4)	i) $f = \left\lceil \boldsymbol{Len}\,(K) / s \right\rceil$. ii) $P = PT_{(inner\_loops-f+1)} \| \ldots \| PT_{inner\_loops}$. iii) $K = K \oplus \boldsymbol{LSB}_{\boldsymbol{Len}(K)}(P)$. iv) $g = \left\lceil \boldsymbol{Len}\,(IV_i) / s \right\rceil$. v) $IV_{(i+1)} =$ $\boldsymbol{LSB}_n(PT_{(inner\_loops-g+1)} \| \ldots \| PT_{inner\_loops})$.	←	i) $f = \left\lceil 192 / s \right\rceil$. ii) $P = PT_{(inner\_loops-f+1)} \| \ldots \| PT_{inner\_loops}$. ii) $K_1 = K_1 \oplus \boldsymbol{LSB}_{64}(P)$. ii) $K_2 = K_2 \oplus$ $\boldsymbol{Truncate}(\boldsymbol{LSB}_{128}(C), 128, 64)$ iii) $K_3 = K_3 \oplus \boldsymbol{MSB}_{64}(C)$.

[a] This step constitutes a part of CFB mode decryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

[c] PT_j and CT are assumed to be s bits in length.

Table 9 — Tailored steps in MCT for OFB mode decryption

Step number	Tailored steps for OFB mode decryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	$INIT\_CT = CT_0$
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	I) $y = \boldsymbol{E}_K(IV)$. [a] [b] II) $PT_j = CT \oplus y$. [a] [b]	←	←
b) - 2) - iii)	I) If ($j = 1$), then $CT = IV$ Else $CT = PT_{(j-1)}$. II) $IV = y$. [a] [b]	←	I) $CT = IV$. II) $IV = y$. [a] [b]
b) - 4)	i) If ($\boldsymbol{Len}(K) = 128$), then $K = K \oplus PT_{inner\_loops}$ Else if ($\boldsymbol{Len}(K) = 192$), then $K = K \oplus \boldsymbol{LSB}_{64}(PT_{(inner\_loops-1)}) \,\|\, PT_{inner\_loops}$ Else if ($\boldsymbol{Len}(K) = 256$), then $K = K \oplus (PT_{(inner\_loops-1)} \,\|\, PT_{inner\_loops})$ Else terminate MCT. ii) $IV = PT_{inner\_loops}$.	i) $K = K \oplus (PT_{(inner\_loops-1)} \,\|\, PT_{inner\_loops})$. ii) $IV = PT_{inner\_loops}$.	i) $K_1 = K_1 \oplus PT_{(inner\_loops-2)}$, ii) $K_2 = K_2 \oplus PT_{(inner\_loops-1)}$, iii) $K_3 = K_3 \oplus PT_{inner\_loops}$. iv) $CT_0 = CT_0 \oplus INIT\_CT$.

[a] This step constitutes a part of OFB mode decryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

Table 10 — Tailored steps in MCT for CTR mode decryption

Step number	Tailored steps for CTR mode decryption		
	For 128-bit block ciphers	For 64-bit block ciphers other than TDEA	For TDEA
b) - 1)	no operation	←	←
b) - 2) - i)	no operation	←	←
b) - 2) - ii)	I) $y = \boldsymbol{E}_K(CTR_j)$. [a] [b] II) $PT_j = CT \oplus y$. [a] [b]	←	←
b) - 2) - iii)	I) $CTR_j = (CTR_j + 1) \bmod 2^n$. [c] II) $CT = PT_j$.	←	←
b) - 4)	If ($\boldsymbol{Len}(K) = 128$), then $K = K \oplus PT_{inner\_loops}$, Else if ($\boldsymbol{Len}(K) = 192$), then $K = K \oplus \boldsymbol{LSB}_{64}(PT_{(inner\_loops-1)}) \,\|\, PT_{inner\_loops}$, Else if ($\boldsymbol{Len}(K) = 256$), then $K = K \oplus (PT_{(inner\_loops-1)} \,\|\, PT_{inner\_loops})$, Else terminate MCT.	$K = K \oplus (PT_{(inner\_loops-1)} \,\|\, PT_{inner\_loops})$.	i) $K_1 = K_1 \oplus PT_{(inner\_loops-2)}$. ii) $K_2 = K_2 \oplus PT_{(inner\_loops-1)}$. iii) $K_3 = K_3 \oplus PT_{inner\_loops}$.

[a] This step constitutes a part of CTR mode decryption.

[b] NIST/SP 800-38A[19] uses the different variable name, O_j, instead of y.

[c] This step corresponds to the counter generation.

9.2.3.1.6 Guidelines for random test

For asymmetric ciphers specified in ISO/IEC 18033-2, an encryption algorithm takes as input a) a public key, b) a label, c) a plaintext and d) an encryption option, as well as outputs a ciphertext. Also, a decryption algorithm takes as input a) a private key, b) a label and c) a ciphertext, as well as outputs a plaintext.

In the random test, the tester shall supply random data to the IUT and test if the resultant data match the expected data. The tester should select plaintexts, by considering the upper limit of plaintext length.

Some asymmetric ciphers involve the random value or salt. If the salt length is not zero, the tester shall request the IUT to generate ciphertexts using only a single public key pair, and a single message, and then shall verify that the resultant ciphertexts match the rest of the expected ciphertexts. If the insertion of the random value is allowed by the IUT's implementation, the tester may insert the random number from outside the cryptographic algorithm boundary, and the tester shall verify that the resultant ciphertext matches the expected ciphertext.

9.2.3.1.7 Representation of test vectors

For block cipher algorithms, test vectors should be represented in natural order, from left to right. For some stream cipher algorithms, octet/bit order or endianness has to be considered in the representation of test vectors. For such algorithms, the representation of test vectors should either be represented big-endian or follow the endianness in the specification.

EXAMPLE 1 Big-endian for SNOW 2.0 (see ISO/IEC 18033-4).

EXAMPLE 2 Little-endian for Rabbit (see ISO/IEC 18033-4).

For asymmetric ciphers, bit strings should be represented in natural order, from left to right, and multiple-precision integers should be represented with the most significant bit first.

An elliptic curve point should be represented as an octet strings using the uncompressed form (see ISO/IEC 18033-2:2006, 5.4.2 and 5.4.3).

9.2.3.2 Guidelines for white box testing

9.2.3.2.1 Guidelines for source code inspection

9.2.3.2.1.1 General guidelines

Beyond the level of functional conformance, vendors may claim additional security features which are imposed by security policy or designed to mitigate attacks. For symmetric key algorithms, subkeys can reside on RAM or cache. In order to minimise the risk to disclose subkeys, vendors may claim that the IUT zeroises subkeys. If such a security feature is claimed, the tester shall perform source code inspection to verify that the IUT zeroises subkeys.

9.2.3.2.1.2 Source code inspection for CTR mode in symmetric algorithms

It is a requirement of the Counter Mode that only a unique counter block can be used for each plaintext block that is ever encrypted with a given key, across all messages. If the IUT derives counter values internally, a source code inspection of the implementation of the counter mechanism shall be conducted to ensure that it provides unique counter block values.

9.2.3.2.2 Guidelines for binary analysis

Beyond the level of functional conformance, vendors may claim additional security features which are imposed by security policy or in order to mitigate attack. Even if the source code itself implements security features, it is not clear that the final implementation representation really implements the security feature, as a result of optimisation of building tools.

9.3 Guidelines specific to digital signature algorithms

9.3.1 Identification of digital signature algorithms

In addition to the identified information in 9.1.1, the vendor shall identify the following information on the IUT:

a) consistency between security function, supporting functions and supported parameters;

b) bit length of each public key component;

c) bit length of each private key component.

9.3.2 Selecting a set of conformance test items

Digital signature algorithms are considered as high-level mechanisms compared with other cryptographic algorithms, such as hashing algorithms. Digital signature algorithms involve several components (e.g. modular arithmetic, hashing algorithm and random number generator); therefore, it will be difficult to apply systematic conformance tests. So, the random test should be selected.

The random test for the signature generation function becomes either a) an independent verification test when salt is used and cannot be inserted, or b) a KAT when no salt is used or when salt can be inserted for conformance testing purpose.

The random test for the signature verification function becomes a KAT.

9.3.3 Guidelines for each conformance test item

9.3.3.1 Guidelines for black box testing

9.3.3.1.1 Guidelines for KAT for signature generation function

Signature generation functions involve parameters such as private key, message and resultant signature. For the non-randomized (or deterministic) signature scheme, the known-answer test can be applied. For the randomized signature scheme, the known-answer test may be applied by forcing the IUT to use tool supplied values for salt.

EXAMPLE Digital signature schemes 1 and 3 in ISO/IEC 9796-2.

In the KAT, the tester shall supply various messages to the IUT and test if each of the resultant signatures matches the expected values, while using a fixed private key. The tester should run this test several times by changing the private keys.

Some of the messages should be purposely selected so that the most significant octet of the expected signature becomes zero, in order to verify that the integer to octet strings conversion is implemented as specified (see A.1).

9.3.3.1.2 Guidelines for the independent verification test for the signature generation function

For the randomized signature scheme, the independent verification test should be applied.

EXAMPLE Digital signature scheme 2 in ISO/IEC 9796-2.

In the independent verification test, the tester shall supply various messages and private keys to the IUT and test if each of the resultant signatures is successfully verified by the testing tool using the corresponding public key.

9.3.3.1.3 Guidelines for KAT for signature verification function

The signature verification functions involve parameters such as public key, message, signature and the result of verification.

In the KAT, the tester shall supply various public keys, messages, signatures, and test if each of the results of the verification matches the expected result.

This KAT should include signatures whose length is not conforming to the standard, in order to verify that the step is implemented to reject non-conforming signatures as specified (see A.1).

9.3.3.1.4 Representation of test vectors

For digital signature algorithms, test vectors should be represented in the order where the most significant octet is on the left and the least significant octet is on the right.

For some digital signature algorithms, the signature can be divided into two parts, e.g. r and s for DSA and ECDSA. For such algorithms, each part should be separately represented or the representation (r || s) should be used.

9.3.3.2 Guidelines for white box testing

9.3.3.2.1 Guidelines for source code inspection

For the signature generation function, the tester should perform source code inspection to verify that the integer to octet strings conversion is implemented as specified (see A.1). For the signature generation function, the tester should perform source code inspection to verify that the non-conforming signatures are rejected as specified (see A.1).

The randomized signature generation function involves the random value or salt. Therefore, the tester should perform source code inspection so that the salt is zeroised (see 9.1.3.1.2.1).

In some standards, it is required that the randomizer or salt is generated in a conforming way to the selected standard. The tester should perform source code inspection following the guidelines in 9.1.3.1.2.2.

In some standards, it is required to avoid the signature value zero by generating a new value for the salt. The tester should perform a source code inspection so that the branch condition is implemented as specified (see 9.1.3.1.3).

9.3.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.4 Guidelines specific to hashing algorithms

9.4.1 Identification of hashing algorithms

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) bit length of message digest;

b) bit length of input block;

c) maximum bit length of input message;

d) whether the IUT is an octet-oriented implementation, or a bit-oriented implementation.

Hashing algorithms do not involve any cryptographic keys themselves. However, if the IUT is to be used in processing any CSPs, the vendor should identify the following information on the IUT.

e) policy to zeroise messages, which can be considered as CSPs.

9.4.2 Selecting a set of conformance test items

A hashing algorithm can be an underlying algorithm of higher-level algorithms, e.g. MAC, RBG, key derivation function and prime number generation. In this sense, implementations of hashing algorithms should be extensively tested.

Taking into account the above situation, the following three basic black box test items can be applicable to hashing algorithms:

a) short message test, which supplies short messages to the IUT and tests if each resultant message digest matches the expected value;

NOTE 1 The word "short" means that the length of message is shorter than the input block size after splitting (see ISO/IEC 10118-1:2000, 6.1.2).

b) selected long message test, which supplies long messages to the IUT and tests if each resultant message digest matches the expected value;

NOTE 2 The word "long" means that the length of message is longer than the input block size after splitting (see ISO/IEC 10118-1:2000, 6.1.2).

c) MCT.

NOTE 3 The MCT is sometimes called the "pseudorandomly generated messages test."

In addition to the above test items, the following optional white box tests can be applicable to hashing algorithms:

a) source code inspection;

b) binary analysis.

9.4.3 Guidelines for each conformance test item

9.4.3.1 Guidelines for black box testing

9.4.3.1.1 Guidelines for short message test

In the short message test, the tester shall supply short messages to the IUT and test if the resultant message digests match the expected values.

Let m be the bit length of the input block of the hashing algorithm.

For octet-oriented implementations, at least (m / 8 + 1) messages should be supplied to the IUT, whose lengths are {0, 8, 16, ..., m }. Their values, except for length, should be random.

For bit-oriented implementations, at least (m + 1) messages should be supplied to the IUT, whose lengths range from 0 to m. Their values, except for length, should be random.

NOTE m is identical to L_1 in ISO/IEC 10118-1:2000, 4.1.

9.4.3.1.2 Guidelines for selected long message test

In the selected long message test, the tester shall supply long messages to the IUT and test if the resultant message digests match the expected values.

The variables and function used in the description of the selected long message test are:

m	The bit length of input block of the hashing algorithm.
D	The message.
$\|D\|_{max}$	The maximum bit length of message supported by the IUT.
min(A, B)	A function that returns either A or B, whichever is less.

Bit lengths of selected long messages should cover the range of [2* m, **min**(100*m, $|D|_{max}$)].

For octet-oriented implementations, at least (m/8) messages should be supplied to the IUT, whose lengths are typically selected according to Formula (1):

$$m + 8 * 99 * i, \qquad [1 \le i \le (m/8)] \tag{1}$$

For bit-oriented implementations, at least m messages should be supplied to the IUT, whose lengths are typically selected according to Formula (2):

$$m + 99 * i, \qquad (1 \le i \le m) \tag{2}$$

The values of messages, except for their lengths, should be random.

9.4.3.1.3 Guidelines for selected MCT

In the MCT, the tester shall choose one initial message of message digest length, randomly. The variables and the function used in the description of the MCT are:

D	The message.
h	The hashing algorithm.
i	A temporary value used as a loop counter.
inner_loops	The number of inner loops.
j	A temporary value used as a loop counter.
k	A temporary value used as a loop counter.
MD_i	The ith resultant message digest.
N	The number of concatenated message digests to create each input message in the MCT.
outer_loops	The number of outer loops. This is identical to the number of resultant message digests.
seed	The initial message whose bit length is identical to the bit length of a message digest.
TMD_i	The ith temporary message digest.

Here, N should be selected so that the bit length of message D becomes greater than or equal to the bit length of input block.

NOTE N = 3 is used in Reference [20].

The general framework of MCT for encryption function is described by the following steps:

1) For $i = 1$ to *outer_loops* do:

   1.1 For $k = 0$ to $(N - 1)$, do:

      1.1.1 $TMD_k = seed$.

   1.2 For $j = N$ to $(inner\_loops + N - 1)$, do:

      1.2.1 $D_i = TMD_{j-N} \,||\, \ldots \,||\, TMD_{j-2} \,||\, TMD_{j-1}$.

      1.2.2 $TMD_j = h(D_i)$.

   1.3 $MD_i = TMD_{(inner\_loops + N - 1)}$.

   1.4 $seed = MD_i$.

   1.5 Output MD_i.

9.4.3.1.4 Representation of test vectors

For hashing algorithms, test vectors should be represented in natural order, from left to right.

9.4.3.2 Guidelines for white box testing

9.4.3.2.1 Guidelines for source code inspection

If the maximum bit length of the input message is enforced in the IUT, the guidelines in 9.1.3.1.3 should be applied. If the IUT is to be used in the processing of any CSPs, the guidelines in 9.1.3.1.2.1 should be applied.

9.4.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.5 Guidelines specific to MAC algorithms

9.5.1 Identification of MAC algorithms

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) underlying algorithm;

b) bit length of MAC;

c) maximum bit length of input message;

d) supported range of bit length of key.

NOTE The defined range of bit length of key in MAC Algorithm 2, known as HMAC, is different from that in Reference [21].

9.5.2 Selecting a set of conformance test items

A MAC algorithm can become the underlying algorithm of higher-level algorithms, e.g. RBG, and key derivation function. However, if the complexity introduced is considered very little, then the following KAT should be selected.

a) Random test, which uses various keys and supplies random keys, messages and any additional information needed to the IUT and tests if each resultant MAC matches the expected value.

If a more extensive test is needed, the tester may select conformance test items listed in 9.4.2.

9.5.3 Guidelines for each conformance test item

9.5.3.1 Guidelines for black box testing

9.5.3.1.1 Guidelines for random test

In the random test, the tester shall supply random messages and any additional information needed to the IUT and test if each resultant MAC matches the expected value, while changing the key value randomly.

Here, the tester shall perform the random test by carefully selecting the following, so that they fit in the defined range of MAC algorithm:

a) supported range of bit length of key;

b) supported range of bit length of additional information.

If there is any branch condition in the cryptographic algorithm specification, above parameters should be selected to cover the branched source codes.

9.5.3.1.2 Representation of test vectors

For MAC algorithms, test vectors should be represented in natural order, from left to right.

9.5.3.2 Guidelines for white box testing

9.5.3.2.1 Guidelines for source code inspection

Beyond the level of functional conformance, vendors may claim additional security features which are imposed by security policy or in order to mitigate attacks. For MAC algorithms, there can be intermediate secret values related to subkeys or derived keys. In order to minimize the risk to disclose subkeys, vendors may claim that the IUT zeroises these secret values. If such a security feature is claimed, the tester shall perform source code inspection to verify that the IUT zeroises secret values.

NOTE For CMAC, see the secret string *S* in ISO/IEC 9797-1:2011, 6.2.3.

Some cryptographic algorithm standards allow the truncation of MAC. In response to the truncation, only truncated MAC values might be used in the health tests as known-answers. If the IUT is able to output full MAC values, the health tests cannot detect errors in bits truncated (i.e. bits not compared). The tester should perform source code inspection of the health tests so that there is no such problem.

9.5.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.6 Guidelines specific to RBG algorithms

9.6.1 Identification of RBG algorithms

An RBG can be categorized into either non-deterministic RBG (NRBG) or deterministic RBG (DRBG) (see ISO/IEC 18031:2011, Figure 2). Currently, there is no standard specification of NRBG in ISO/IEC 18031. So, guidelines in 9.6 focus only on the DRBG algorithms.

NOTE For example, the AIS 20/31[22] defines comprehensive testing for non-deterministic as well as for deterministic RBGs; however, it is open for debate, how much of this belongs to correctness testing as opposed to vulnerability analysis. Of course, statistical tests can be made in the same way for all types of RBGs but statistical tests do not prove unpredictability of random numbers. The latter can only be concluded from analysing the design of the RBG, which is probably beyond the scope of correctness testing.

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) underlying algorithm;

b) bit length of output block;

EXAMPLE Bit length of hashing algorithm for hash-based RBGs.

c) maximum bit length of random bits;

d) supported range of entropy input;

e) supported range of nonce;

f) supported range of personalization string;

g) supported range of additional input;

h) reseed interval;

i) whether the IUT supports reseed capability;

j) whether the IUT enables prediction resistance;

k) whether the IUT uses a derivation function.

9.6.2 Selecting a set of conformance test items

Before selecting a set of conformance test items, the tester shall verify that the conformance to the underlying algorithms in RBG is already met. Therefore, it is assumed that the underlying algorithms are extensively tested. The following KAT should be selected.

Random test, which uses various information needed to the IUT and tests if each random bit matches the expected value.

9.6.3 Guidelines for each conformance test item

9.6.3.1 Guidelines for black box testing

9.6.3.1.1 Guidelines for random test

In the random test, the tester shall supply random data needed to the IUT and test if each resultant pseudorandom bit matches the expected value.

In order to verify the output generation function generating two or more blocks of pseudorandom bits correctly, the tester shall perform the random test by carefully selecting the following:

a) requested number of bits.

If a counter is introduced in the specification for selected RBG, at least 512 test vectors should be used.

EXAMPLE Hash_DRBG, and CTR_DRBG.

In ISO/IEC 19790:2012, 7.8.2.3 and in ISO/IEC 18031:2011, 9.8.8, the continuous random bit generator test is required, and the first block or the first 80 bits cannot be output. Considering this requirement, the tester may omit the first one or several blocks from the expected value.

9.6.3.1.2 Representation of test vectors

For RBG algorithms, test vectors should be represented in natural order, from left to right.

9.6.3.2 Guidelines for white box testing

9.6.3.2.1 Guidelines for source code inspection

The tester shall perform source code inspection to verify that the documented reseed interval is really used in the IUT. Guidelines in 9.1.3.1.3 should be followed.

There are health test requirements in ISO/IEC 18031 or its comparable standards. Guidelines in 9.1.3.1.2.3 should be followed.

ISO/IEC 18031 and ISO/IEC 19790 request IUTs to inhibit outputting random bits while health tests are being performed or the IUT is in an error state. However, the output inhibition might not be a matter of IUTs, but the matter of cryptographic modules (see ISO/IEC 19790:2012, 7.10.3.2) or of security architecture under ISO/IEC 15408 (see ADV_ARC.1-5 in ISO/IEC 18045). For the latter case, through the cryptographic module testing, the tester should verify that output inhibition is enforced as specified in the selected standard.

9.6.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.7 Guidelines specific to key establishment mechanisms

9.7.1 Identification of key establishment mechanisms

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) supported scheme;

b) consistency between security function, supporting functions, and supported parameters.

9.7.2 Selecting a set of conformance test items

Key establishment mechanisms are considered high-level mechanisms compared with other cryptographic algorithms, such as hashing algorithms. Key establishment mechanisms involve at least two entities and can include an error condition which results in not sharing any key. In order to test such

error handling ability, the following two black box test items can be applicable to key establishment mechanisms:

a) independent verification test, where the IUT plays the role of one entity and the testing tool plays the role of another entity, which verifies the shared secret key generated by the IUT with the testing tool independently;

b) validity test, which tests the ability of the IUT to recognize valid and invalid results received from the testing tool.

In addition to the above test items, the following optional white box tests can be applicable to key establishment mechanisms:

a) source code inspection;

b) binary analysis.

9.7.3 Guidelines for each conformance test item

9.7.3.1 Guidelines for black box testing

9.7.3.1.1 Guidelines for independent verification test

Key establishment mechanisms involve at least two entities. In order to test key establishment mechanisms, it is assumed that the IUT plays the role of one (or supported) entity, and that the testing tool plays the role of another entity.

Also, key establishment mechanisms can involve generating either ephemeral key pair(s) or nonce, and verifying public key(s) sent by another entity. Therefore, the resultant secret key or shared secret key cannot always be determined in advance. In this sense, the independent verification test verifies the secret key or shared secret key by the testing tool independent of the IUT. Also, the independent verification test enables the conformance test for key establishment mechanism as a whole, i.e. including key pair generation and public key validation.

The independent verification test shall supply several sets of the following and test if the IUT generates correct shared secret keys:

a) domain parameter which passes domain parameter validation, if any;

b) (ephemeral or static) public key(s), which passes public key validation in conjunction with domain parameter a), sent by the other entity;

c) OtherInfo or its components, if any;

d) nonce sent by the other entity, if any.

The independent verification test should verify that resultant shared secret keys are mutually different if the IUT is capable of generating ephemeral key(s) or nonce.

The independent verification test should supply several sets of the following and test if the IUT responds as specified in the selected standard:

a) domain parameter which passes domain parameter validation, if any;

b) static public key(s), which passes public key validation in conjunction with domain parameter a), sent by the other entity, if any;

c) ephemeral public key(s), which fails public key validation in conjunction with domain parameter a), sent by the other entity, if any;

d) OtherInfo or its components, if any;

e) nonce sent by the other entity, if any.

9.7.3.1.2 Guidelines for validity test

The purpose of the validity test is to test the ability of the IUT to recognize valid and invalid results received from the testing tool. The validity test involves the following parameters:

a) domain parameter which passes domain parameter validation, if any;

b) (ephemeral or static) private key(s);

c) (ephemeral or static) public key(s);

d) shared secret key;

e) OtherInfo or its components, if any;

f) nonce sent by the other entity, if any;

g) secret key (or derived keying material), if any.

Incorrect values are generated by the testing tool by inserting errors in parameters except for the domain parameter. In order for the IUT not to include the common mistake in A.1, the validity test should include the invalid octet representation of shared secret key *Z*; otherwise, this aspect should be verified by source code inspection.

9.7.3.1.3 Representation of test vectors

For key establishment mechanisms, octet strings which are converted from integers should be represented in the order that the most significant octet on the left and the least significant octet on the right.

The other parameters should be represented in natural order, from left to right.

9.7.3.2 Guidelines for white box testing

9.7.3.2.1 Guidelines for source code inspection

The tester should perform source code inspection to verify that the integer to octet strings conversion is implemented as specified (see A.1).

In some key establishment mechanisms, weak values, e.g. shared secret key $Z = 1$, are handled specially in the specifications. For such key establishment mechanisms, guidelines in 9.1.3.1.3 to 9.1.3.1.4 should be followed to verify that the weak values are handled as specified.

In some key establishment mechanisms, ephemeral keys are used. The key establishment mechanisms only using static keys sometimes involve random value or salt. Therefore, the tester should perform source code inspection so that ephemeral keys, salt and their derivatives including shared secret keys are zeroised.

In some standards, it is required that ephemeral keys are generated in a conformable way to the selected standard. It is required to generate the salt in the same way. The tester should perform source code inspection following guidelines in 9.1.3.1.2.2.

9.7.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.8 Guidelines specific to key derivation function

9.8.1 Identification of key derivation function

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) underlying algorithm(s)

EXAMPLE 1 Hashing algorithms.

EXAMPLE 2 MAC algorithms.

9.8.2 Selecting a set of conformance test items

A key derivation function becomes the underlying algorithm of key establishment mechanism. However, if the complexity introduced is considered little, then the following KAT should be selected:

a) Random test, which uses various key derivation key, and any additional information needed to the IUT and tests if each key matches the expected value.

   If a more extensive test is needed, the tester may select conformance test items listed in 9.4.2.

b) Source code inspection.

c) Binary analysis.

9.8.3 Guidelines for each conformance test item

9.8.3.1 Guidelines for black box testing

9.8.3.1.1 Guidelines for random test

In the random test, the tester shall supply a random key derivation key and any additional information needed to the IUT and test if each resultant key matches the expected value.

Some key derivation functions use ASN.1 DER encoding. Correct encoding of the length field should be verified by inputting various lengths of key derivation keys or any additional information.

9.8.3.1.2 Representation of test vectors

For key derivation functions, test vectors should be represented in natural order, from left to right.

9.8.3.2 Guidelines for white box testing

9.8.3.2.1 Guidelines for source code inspection

In order to complement the black box test, the tester should perform source code inspection that ASN.1 DER encoding is performed correctly. As the inputs are considered CSPs, guidelines in 9.1.3.1.2.1 should be applied.

Key derivation functions are generally iterating underlying hashing algorithms or pseudorandom functions. Each call to underlying algorithm results in partial or intermediate key value. Some cryptographic algorithm specifications request IUTs to inhibit the outputting of the intermediate key value before the generation of the entire key value (see Reference [23]). The tester should perform source code inspection to verify that outputting the intermediate key values is inhibited.

However, the output inhibition might not be a matter of IUTs, but a matter of cryptographic modules (see ISO/IEC 19790:2012, 7.10.3.2) or of security architecture under ISO/IEC 15408 (see ADV_ARC.1-5 in

ISO/IEC 18045). For the latter case, through the cryptographic module testing, the tester should verify that output inhibition is enforced as specified in the selected cryptographic algorithm specification.

9.8.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

9.9 Guidelines specific to prime number generation

9.9.1 Identification of prime number generation

In addition to the identification information in 9.1.1, the vendor shall identify the following information on the IUT:

a) prime number generation method;

EXAMPLE 1 Using probabilistic primality test, such as Miller-Rabin primality test.

EXAMPLE 2 Using deterministic methods, such as Shawe-Taylor's algorithm.

b) underlying algorithm(s);

EXAMPLE 3 Random bit generators.

EXAMPLE 4 Hashing algorithms.

c) supported bit length of prime;

d) error probability for probabilistic primality test.

NOTE Required error probability or resultant risk might vary depending on the context of prime usage (see Annex F of Reference [16]).

9.9.2 Selecting a set of conformance test items

Prime number generation methods are considered as high-level mechanisms compared with other cryptographic algorithms, such as hashing algorithms. In ISO/IEC 18032, there are two categories of prime number generation methods, either using probabilistic primality test or using deterministic test. In order to test two categories, following two black box test items can be applicable to prime number generation methods:

a) independent verification test, where the testing tool uses probabilistic primality testing independently from the IUT to verify that the candidate number is probable prime;

NOTE 1 This independent verification test can be applied to prime number generation using deterministic methods.

b) KAT, where the testing tool supplies intermediate values to the IUT which will lead to the resultant prime, which tests if the resultant number matches the expected value.

NOTE 2 Some of the deterministic methods continue a set of steps until the prime is constructed. For such an implementation, a testing interface would be needed to insert intermediate values supplied by the testing tool.

Prime number generation methods are complex compared with other algorithms. In addition to the above test items, the following optional white box tests can be applicable to prime number generation:

a) source code inspection;

b) binary analysis.

9.9.3 Guidelines for each conformance test item

9.9.3.1 Guidelines for black box testing

9.9.3.1.1 Guidelines for independent verification test for prime number generation

In ISO/IEC 18032:2005, 8.1, the error probability has to be at most 2^{-100}. Based on the worst-case error estimate, the tester shall apply the Miller-Rabin test with 50 iterations of the generated candidate numbers to verify that they are probably prime.

NOTE 50 is based on the worst case error estimate for the Miller-Rabin Primality test in ISO/IEC 18032:2005, Annex A.

In order to verify that generated candidate numbers are random, the tester shall exercise the IUT to generate at least 10 candidate numbers and verify that they are mutually independent.

9.9.3.1.2 Guidelines for KAT for deterministic prime number generation

If the IUT implements a deterministic prime number generation method, a KAT can be used to verify the conformance by inserting intermediate values from the testing tool.

9.9.3.1.3 Representation of test vectors

For prime number generation, octet strings which are converted from integers should be represented in the order of the most significant octet on the left and the least significant octet on the right.

The other parameters should be represented in natural order, from left to right.

9.9.3.2 Guidelines for white box testing

9.9.3.2.1 Guidelines for source code inspection

Care should be taken when claiming conformance to prime number generation based on the specific standard. There are some differences between standards for specific prime number generation method, e.g. prime number generation using Miller-Rabin primality test. So, the identification of selected prime number generation methods is quite important.

There are many error conditions in prime number generation methods compared with the other cryptographic algorithms. The tester should verify that error conditions are implemented as specified (see 9.1.3.1.3). Also, the tester should verify that status outputs are implemented as specified (see 9.1.3.1.4). The Miller-Rabin primality test involves a random integer or random bits. The random integer has to be generated in the specific way (or using an approved RBG) in some standards. If there is such a requirement, guidelines in 9.1.3.1.2.2 should be followed.

9.9.3.2.2 Guidelines for binary analysis

Guidelines in 9.1.3.2 should be applied.

10 Conformance testing

10.1 Level of conformance testing

For each type of algorithm in Clause 6, this document will recommend an evaluation and test method and the relevant criteria from Clause 7 for conformance testing. Different levels of evaluation (EAL type) can be taken into consideration to be associated with ISO/IEC 15408 Evaluation Assurance level (EAL) and ISO/IEC 19790 security levels.

10.2 Symmetric key cryptographic algorithms

10.2.1 n-bit block cipher

10.2.1.1 General

10.2.1.1.1 Block cipher algorithms in ISO/IEC 18033-3

ISO/IEC 18033-3 specifies four 64-bit block ciphers:

a) TDEA;

b) MISTY1;

c) CAST-128;

d) HIGHT.

TDEA uses a 3-tuple of 64-bit keys, each of which has the property of odd-parity. 64-bit block ciphers other than TDEA use a 128-bit key.

ISO/IEC 18033-3 specifies three 128-bit block ciphers:

e) AES;

f) Camellia;

g) SEED.

128-bit block ciphers other than SEED use a key, and the bit length of the key is either 128, 192 or 256 bits. SEED uses a 128-bit key.

10.2.1.1.2 Block cipher modes of operation in ISO/IEC 10116

Conformance testing for n-bit block cipher testing can be performed together with that for block cipher mode of operation. ISO/IEC 10116 specifies five modes of operation:

a) Electronic Codebook (ECB);

b) Cipher Block Chaining (CBC);

c) Cipher Feedback (CFB);

d) Output Feedback (OFB);

e) Counter (CTR).

All of the above modes can be combined with the block cipher algorithms listed in ISO/IEC 18033-3.

10.2.1.2 Black box testing

10.2.1.2.1 Known-answer tests

For encryption and decryption, there are at least three types of known-answer tests which shall be performed:

a) KAT-Text (see 9.2.3.1.1);

b) KAT-Key (see 9.2.3.1.2);

c) KAT-Sbox (see 9.2.3.1.3).

In addition to these known-answer tests, other known-answer tests may be employed and performed to test specific parts of block cipher algorithm.

EXAMPLE 1 Permutation Operation KAT (see Reference [17]).

EXAMPLE 2 Inverse Permutation KAT (see Reference [17]).

EXAMPLE 3 Initial Permutation KAT (see Reference [17]).

10.2.1.2.2 MMT

The tester shall follow guidelines in 9.2.3.1.4.

10.2.1.2.3 MCT

The tester shall follow guidelines in 9.2.3.1.5. The quantitative parameters, *inner_loops* and *outer_loops*, in 9.2.3.1.5.1 shall meet the following:

a) *inner_loops* ≥ 1 000;

NOTE 1 In Reference [17], *inner_loops* is selected as 10 000.

b) *outer_loops* ≥ 100;

NOTE 2 In Reference [17], *outer_loops* is selected as 400.

10.2.1.3 White box testing

Guidelines in 9.2.3.2 should be followed.

10.3 Asymmetric key cryptographic algorithms

10.3.1 Digital Signature Algorithm (DSA)

10.3.1.1 General

In DSA, there are the following two security functions specified:

a) Signature Generation;

b) Signature Verification.

In relation to DSA, the following three security parameter generation and establishment methods are defined:

c) Domain Parameter Generation;

d) Domain Parameter Verification;

e) Key Pair Generation.

The black box testing is defined for each of the above. The variables and function commonly used in the black box testing are:

counter The counter value that results from the domain parameter generation process when the domain parameter seed is used to generate DSA domain parameters.

g One of the DSA domain parameters; *g* is a generator of the *q*-order cyclic group of GF(*p*)*, that is, an element of order *q* in the multiplicative group of GF(*p*).

N_{BB} The number of test records.

inhlen The input block length of the underlying hashing algorithm, in bits.

p One of the DSA domain parameters; a prime number that defines the Galois Field GF(*p*) and is used as a modulus in the operations of GF(*p*).

q One of the DSA domain parameters; a prime factor of $p - 1$.

r One component of a DSA digital signature.

s One component of a DSA digital signature.

SEED A seed used for the generation of domain parameters.

x The DSA private key.

y The DSA public key.

10.3.1.2 Domain Parameter Generation Test

10.3.1.2.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported domain parameter size and supported generation algorithm, the tester shall request the IUT to generate N_{BB} sets of domain parameters *p*, *q* and *g*, together with *SEED* and *counter* as specified in ISO/IEC 14888-3:2016, Annex D.

NOTE FIPS 186[16] uses the different variable name, *domain_parameter_seed*, instead of *SEED*.

b) N_{BB} shall be greater than or equal to 10.

c) The tester shall test if the resultant domain parameters and their associated values (*SEED* and *counter*) can be verified using the domain parameter verification function of the reference implementation.

10.3.1.2.2 White box testing

The source code inspection shall be applied to verify that

a) the selected primality test is implemented and used as specified in the selected cryptographic algorithm standard (see 9.9), and

b) the tested implementation of the hashing algorithm is used.

10.3.1.3 Domain Parameter Verification Test

10.3.1.3.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported domain parameter size, and supported generation algorithm, the tester shall generate N_{BB} sets of domain parameter p and q, together with *SEED* and *counter* as specified in ISO/IEC 14888-3:2016, Annex D, using a reference implementation.

   1) N_{BB} shall be greater than or equal to 10.

   2) The tester shall modify approximately one half of N_{BB} sets randomly so that either (i) *SEED* does not produce q, (ii) q is not prime, (iii) *SEED* and *counter* do not produce p, (iv) p is not prime, or (v) q does not divide p -1; the rest of N_{BB} sets shall remain unmodified.

b) The tester shall obtain a set of N_{BB} PASS or FAIL values resulting from the IUT's domain parameter verification method.

c) If the verifiable generation of g is claimed,

   1) the tester shall generate additional N_{BB} sets of domain parameter p, q and g, together with *SEED* and *index*, using a reference implementation,

   2) the tester shall modify three of the g values, and

   3) the tester shall supply the five sets of domain parameter to the IUT and test if the IUT determines unmodified g values as valid, and modified g values as invalid.

10.3.1.3.2 White box testing

The tester should inspect the source code to verify that the selected primality test is really implemented.

10.3.1.4 Key Pair Generation Test

10.3.1.4.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported domain parameter size, the tester shall generate one set of domain parameter p, q and g, using a reference implementation.

b) The tester shall request the IUT to generate N_{BB} sets of private/public key pair (x, y), using the domain parameter.

   1) N_{BB} shall be greater than or equal to 10.

c) To determine correctness, the tester shall verify that the equality ($y \equiv g^x \bmod p$) holds for each generated key pair (x, y), by using the domain parameter.

10.3.1.4.2 White box testing

The source code inspection should be applied to verify that

a) the private key is generated using the RBG which meets the requirements on the selected cryptographic algorithm standard, and

b) the private key value resides in the range of the selected cryptographic algorithm standard.

10.3.1.5 Signature Generation Test

10.3.1.5.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported domain parameter size and underlying hashing algorithm pair, the tester shall generate (i) a single set of domain parameter p, q and g, and (ii) N_{BB} sets of long messages.

   1) N_{BB} shall be greater than or equal to 10. If the tester focuses on the black box testing more, at least 512 sets of test vectors should be supplied to the IUT (see 9.1.2).

   2) The length of messages should be selected by considering the underlying hashing algorithms.

   EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB}/2) messages of the length, (2 * *inhlen* + k), where k is less than the input block length.

b) The tester shall supply values generated in the previous step to the IUT, and obtain, for each message, a public key and the resulting signature value (r, s).

c) The tester shall test if the resultant signatures can be verified using the signature verification function of the reference implementation.

Note that if the insertion of a per-message secret number is allowed by the IUT, this black box testing becomes deterministic. If not, this independent verification test still works without the internally generated per-message secret number.

10.3.1.5.2 White box testing

The source code inspection shall be applied to verify that

a) the random bit generator is called each time to generate the per-message secret number,

b) the salt is zeroised, and

c) integer to octet string conversion is conformant to the selected standard while considering common mistakes in A.1.

10.3.1.6 Signature Verification Test

10.3.1.6.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported modulus length and underlying hashing algorithm pair, the tester shall generate one set of domain parameter p, q and g, and a set of N_{BB} sets of message, public key and signature (r, s) tuples.

   1) N_{BB} shall be greater than or equal to 15.

   2) The length of messages should be selected by considering the underlying hashing algorithms.

   EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB}/2) messages of the length, (2 * *inhlen* + k), where k is less than the input block length.

   3) The tester shall select approximately one half of N_{BB} sets randomly, and shall modify one of the values (message, public key, r or s) in the tuples; the rest of N_{BB} sets shall remain unmodified.

4) Considering common mistakes in A.1, signatures with non-conformant lengths should be included in the black box testing.

b) The tester shall supply values generated in the previous step to the IUT, and obtain in response a set of N_{BB} PASS or FAIL values.

c) The tester shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

10.3.1.6.2 White box testing

The source code inspection should be applied to verify that

a) signatures with non-conformant lengths are rejected by the IUT.

10.3.2 RSA

10.3.2.1 Introduction

In RSA, there are the following security functions specified:

a) Signature Generation;

b) Signature Verification.

NOTE ISO/IEC 14888-2 includes RSA and RW schemes which can be made identical to RSASSA-PSS algorithm.

The black box testing is defined for each of the above. The variables commonly used in the black box testing are:

e The public verification exponent of an RSA public key.

N_{BB} The number of test records.

inhlen The input block length of the underlying hashing algorithm, in bits.

10.3.2.2 Signature Generation Test

10.3.2.2.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported modulus size and underlying hashing algorithm pair, the tester shall generate N_{BB} sets of long messages and per-message secret number, and generate one RSA key pair.

1) N_{BB} shall be greater than or equal to 10. If the tester focuses on the black box testing more, at least 512 sets of test vectors should be supplied to the IUT (see 9.1.2).

2) The length of messages should be selected by considering the underlying hashing algorithms.

EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB}/2) messages of the length, (2 * *inhlen* + *k*), where *k* is less than the input block length.

b) The tester shall supply values generated in step a) to the IUT, and obtain N_{BB} sets of signatures from the IUT.

c) The tester shall test if the resultant signatures can be verified using the signature verification function of the reference implementation.

Note that if the insertion of the per-message secret number is allowed to the IUT, this black box testing becomes deterministic. If not, this independent verification test still works without the internally generated per-message secret number.

10.3.2.2.2 White box testing

The source code inspection shall be applied to verify that

a) the random bit generator is called each time to generate salt,

b) the salt is zeroised, and

c) the integer to octet string conversion is conformant to the selected standard while considering common mistakes in A.1.

10.3.2.3 Signature Verification Test

10.3.2.3.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported modulus size and underlying hashing algorithm pair, the tester shall generate N_{BB} sets of messages, public key and signature.

   1) N_{BB} shall be greater than or equal to 15. If the tester focuses on the black box testing more, at least 512 sets of test vectors should be supplied to the IUT (see 9.1.2).

   2) The length of messages should be selected by considering the underlying hashing algorithms.

   EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB}/2) messages of the length, (2 * *inhlen* + *k*), where *k* is less than the input block length.

   3) The tester shall select approximately one half of N_{BB} sets and shall modify one of the values (message, public key exponent *e* or signature) in the sets.

   4) Considering common mistakes in A.1, signatures with non-conformant lengths should be included in the black box testing.

   5) Some standards have specific requirements on the public key exponent *e* (see Reference [16]), so the tester should modify the public key exponent *e*, in order to test if the IUT rejects such invalid public key exponent *e*.

b) The tester shall supply the N_{BB} sets to the IUT and obtain in response a set of N_{BB} PASS or FAIL values.

c) The tester shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

10.3.2.3.2 White box testing

The source code inspection should be applied to verify that

a) signatures with non-conformant lengths are rejected by the IUT,

b) invalid public key exponent *e* is rejected by the IUT, and

c) specified encoding rules are verified by the IUT.

10.3.3 Elliptic Curve Digital Signature Algorithm (ECDSA)

10.3.3.1 General

In ECDSA, there are the following two security functions specified:

a) Signature Generation;

b) Signature Verification.

In relation to ECDSA, the following two security parameter generation and establishment methods are defined:

a) Key Pair Generation;

b) Public Key Validation.

The black box testing is defined for each of the above. The variables and function commonly used in the black box testing are:

N_{BB} The number of test records.

inhlen The input block length of the underlying hashing algorithm, in bits.

r One component of an ECDSA digital signature.

s One component of an ECDSA digital signature.

10.3.3.2 Key Pair Generation Test

10.3.3.2.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each selected domain parameter, the tester shall request the IUT to generate N_{BB} private/public key pairs.

   1) N_{BB} shall be greater than or equal to 10.

b) To determine correctness, the tester shall verify that the generated key pairs pass the public key verification (PKV) function of a reference implementation.

10.3.3.2.2 White box testing

The source code inspection should be applied to verify that

a) the private key is generated using the RBG which meets the requirements on the selected cryptographic algorithm standard, and

b) the private key value resides in the range of the selected cryptographic algorithm standard.

10.3.3.3 Public Key Validation

The following or equivalent steps shall be taken to perform black box testing.

a) For each selected domain parameter, the tester shall generate N_{BB} private/public key pairs using the key generation function of a reference implementation.

   1) N_{BB} shall be greater than or equal to 10.

2) The tester shall modify approximately one half of N_{BB} sets so that public keys are invalid, leaving the rest of N_{BB} sets unmodified (i.e. valid).

b) The tester shall supply values generated in the previous step to the IUT, and obtain in response a set of N_{BB} PASS or FAIL values.

c) The tester shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

10.3.3.4 Signature Generation Test

10.3.3.4.1 Black box testing

The following or equivalent steps shall be taken to perform black box testing.

a) For each selected domain parameter and underlying hashing algorithm pair, the tester shall generate N_{BB} sets of long messages.

1) N_{BB} shall be greater than or equal to 10. If the tester focuses on the black box testing more, at least 512 sets of test vectors should be supplied to the IUT (see 9.1.2).

2) The length of messages should be selected by considering the underlying hashing algorithms.

EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB}/2) messages of the length, (2 * *inhlen* + *k*), where *k* is less than the input block length.

b) The tester shall supply values generated in the previous step to the IUT, and obtain, for each message, a public key and the resulting signature value (*r*, *s*).

c) To determine correctness, the tester shall use the signature verification function of a reference implementation.

10.3.3.4.2 White box testing

The source code inspection should be applied to verify that

a) the random bit generator is called each time to generate salt, and

b) the salt is zeroised.

10.3.3.5 Signature Verification Test

The following or equivalent steps shall be taken to perform black box testing.

a) For each selected domain parameter and underlying hashing algorithm pair, the tester shall generate N_{BB} sets of message, public key and signature (*r*, *s*) tuples.

1) N_{BB} shall be greater than or equal to 15.

2) The length of messages should be selected by considering the underlying hashing algorithms.

EXAMPLE By introducing the variable *inhlen* for the input block length of the underlying hashing algorithm, N_{BB} sets of long messages are selected by generating (N_{BB}/2) messages of the length, (2 * *inhlen*), and (N_{BB} /2) messages of the length, (2 * *inhlen* + *k*), where *k* is less than the input block length.

3) The tester shall select approximately one half of N_{BB} sets randomly and shall modify one of the values (message, public key or signature) in the tuples; the rest of N_{BB} sets shall remain unmodified.

b) The tester shall supply values generated in the previous step to the IUT, and obtain in response a set of N_{BB} PASS or FAIL values.

c) The tester and shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

10.4 Dedicated hashing algorithms

10.4.1 General

The IUTs may be either octet-oriented implementations, or bit-oriented implementations. The former implementations hash messages that are an integral number of octets in length, i.e. the length (in bits) of the message to be hashed is divisible by 8. The latter implementations hash messages of arbitrary length.

10.4.2 Black box testing

10.4.2.1 Short message test

For each hashing algorithm implementation, the tester shall follow guidelines in 9.4.3.1.1. In order to apply guidelines, the values of *m* are listed in Table 11.

Table 11 — Values of *m* for dedicated hashing algorithms

Name of hashing algorithm	Value of *m* in 9.4.3.1.1.
SHA-1	512
SHA-224	512
SHA-256	512
SHA-384	1024
SHA-512	1024
SHA-512/224	1024
SHA-512/256	1024

10.4.2.2 Selected long message test

For each hashing algorithm implementation, the tester shall follow guidelines in 9.4.3.1.2.

10.4.2.3 Monte Carlo test

For each hashing algorithm implementation, the tester shall follow guidelines in 9.4.3.1.3. In order to apply guidelines, the value of *inner_loops* shall be greater than or equal to 1 000, and *outer_loops* shall be greater than equal to 100.

10.4.3 White box testing

For each hashing algorithm implementation, the tester shall follow guidelines in 9.4.3.2.

10.5 Message Authentication Codes (MAC)

10.5.1 Black box testing

10.5.1.1 MAC generation function

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported bit length or range of bit lengths of the cryptographic key, the tester shall randomly generate N_{BB} sets of cryptographic key and message.

   1) For dedicated hashing algorithms, several length of keys shall be included.

NOTE 1 In MAC Algorithm 2 of ISO/IEC 9797-2, the key size k in bits is defined to be at least L_2, where L_2 is the bit length of the message digest, and at most L_1 bits, where L_1 in ISO/IEC 10118-1:2000, 4.1, i.e. $L_2 \leq k \leq L_1$.

NOTE 2 In Reference [21], the key size k in bits is divided into three cases: (i) $k < L_1$, (ii) $k = L_1$ and (iii) $L_1 < k$.

2) Several lengths of messages shall be included.

EXAMPLE 1 Message of zero length, if supported.

EXAMPLE 2 Message, length of which is not a multiple of length of block size of underlying block cipher algorithm (see Reference [24]).

EXAMPLE 3 Message of length up to 2^{16} octets (see Reference [24]).

EXAMPLE 4 For MACs using dedicated hashing algorithms, guidelines in 9.4.3.1 can be applicable.

b) The tester shall supply values generated in the previous step to the IUT, and obtain, for each message, the resulting MAC *T*.

c) The tester shall test if the resultant MACs can be verified using the MAC verification function of reference implementation.

10.5.1.2 MAC verification function

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported bit length or range of bit lengths of the cryptographic key, the tester shall randomly generate N_{BB} sets of cryptographic key, message and corresponding MAC.

1) For dedicated hashing algorithms, several lengths of key shall be included.

NOTE 1 In MAC Algorithm 2 of ISO/IEC 9797-2, the key size k in bits is defined to be at least L_2, where L_2 is the bit length of the message digest, and at most L_1 bits, where L_1 in ISO/IEC 10118-1:2000, 4.1, i.e. $L_2 \leq k \leq L_1$.

NOTE 2 In Reference [21], the key size k in bits is divided into three cases: (i) $k < L_1$, (ii) $k = L_1$ and (iii) $L_1 < k$.

2) The several lengths of messages shall be included.

EXAMPLE 1 Message of zero length, if supported.

EXAMPLE 2 Message, length of which is not a multiple of length of block size of underlying block cipher algorithm (see Reference [24]).

EXAMPLE 3 Message of length up to 2^{16} octets (see Reference [24]).

EXAMPLE 4 For MACs using dedicated hashing algorithms, guidelines in 9.4.3.1 can be applicable.

3) The tester shall select approximately one half of N_{BB} sets randomly, and shall modify one of the values (cryptographic key, message, or MAC) in the tuples.

b) The tester shall supply the N_{BB} sets to the IUT and obtain in response a set of N_{BB} PASS or FAIL values.

c) The tester and shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

10.5.2 White box testing

The source code inspection shall be applied to verify that

a) intermediate secret values related to subkeys or derived keys are zeroised, and

b) for the MAC verification function, the final step of comparing generated MAC with supplied MAC is included in the boundary of the IUT.

10.6 Authenticated encryption

10.6.1 Black box testing

10.6.1.1 Generation-encryption function

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported bit length of the cryptographic key and supported MAC tag length, the tester shall generate N_{BB} sets of the following information:

   1) a cryptographic key, K;

   2) a message;

      i) Several lengths of messages shall be included.

   NOTE 1 This case is called the "variable payload test" in Reference [25].

   3) an additional authentication data, A;

      i) Several lengths of additional authentication data shall be included.

   NOTE 2 This case is called the "variable associated data test" in Reference [25].

   4) a starting variable, S, for CCM and GCM.

   NOTE 3 The starting variable S is also known as the nonce in CCM (see Reference [26]).

   NOTE 4 The starting variable S is also known as the initialization vector in GCM (see Reference [27]).

   NOTE 5 If the IUT itself generates the starting variable S, the tester does not have to generate starting variables, but has to obtain starting variables generated by the IUT (see Reference [28]).

      i) The supported lengths of the starting variable shall be included.

   NOTE 6 This case is called the variable nonce test in Reference [25].

b) N_{BB} shall be greater than or equal to 15.

c) The tester shall supply values generated in the previous step to the IUT, and obtain, for each message

   1) a ciphertext and a MAC tag, which are either concatenated into single octet array or separately provided, and

   2) a starting variable, S, for GCM, if the IUT itself generates the starting variable S.

d) The tester shall test if the resultant ciphertexts and MAC tags can be verified using the decryption-verification function of reference implementation.

10.6.1.2 Decryption-verification function

The following or equivalent steps shall be taken to perform black box testing.

a) For each supported bit length of the cryptographic key and supported MAC tag length, the tester shall randomly generate N_{BB} sets of the following information:

   1) a cryptographic key, K;

2) a message;

i) Several lengths of messages shall be included.

3) an additional authentication data, *A*;

i) Several lengths of additional authentication data shall be included.

4) a starting variable, *S*, for CCM and GCM;

NOTE 1 The starting variable *S* is also known as the nonce in CCM (see Reference [26]).

NOTE 2 The starting variable *S* is also known as the initialization vector in GCM (see Reference [27]).

i) The supported lengths of the starting variable shall be included.

5) corresponding ciphertext and MAC tag.

b) N_{BB} shall be greater than or equal to 15.

c) The tester shall select approximately one half of N_{BB} sets randomly and shall modify one of the values (additional authentication data, starting variable, ciphertext or MAC tag) in the tuples; the rest of N_{BB} sets shall remain unmodified.

d) The tester shall supply values generated in the previous step to the IUT, and obtain

1) N_{BB} sets of PASS or FAIL values, and

2) decrypted message for each PASS case.

e) The tester shall verify that PASS results correspond to the unmodified sets and FAIL results correspond to modified sets.

f) The tester shall verify that the decrypted message matches the original message for each PASS case.

10.6.2 White box testing

The source code inspection shall be applied to verify that

a) intermediate secret values are zeroised.

10.7 Deterministic Random Bit Generation algorithms

10.7.1 DRBG based on ISO/IEC 18031

10.7.1.1 Black box testing

10.7.1.1.1 General

In ISO/IEC 18031, the reseed function and prediction resistance request parameter are defined. The prediction resistance request parameter makes sense for IUTs which support reseed function. Therefore, IUTs based on ISO/IEC 18031 will be categorized into the following three cases:

a) DRBG with reseed capability and with capability of processing prediction resistance request;

b) DRBG with reseed capability, but without capability of processing prediction resistance request;

c) DRBG without reseed capability.

The black box testing for cases a), b) and c) are defined in 10.7.1.1.2, 10.7.1.1.3, and 10.7.1.1.4, respectively.

The variables and function commonly used in the black box testing are:

