SOG-IS Crypto Working Group

SOG-IS Crypto Evaluation Scheme Harmonised Cryptographic Evaluation Procedures

21/12/2020

Version 0.16

Table of contents

1 Introduction

1.1 Target Audience

This document is addressed to:

- developers who are preparing for a cryptographic evaluation of the mechanisms involved in their security product;
- evaluators who are developer independent people commissioned to evaluate the security of cryptographic mechanisms.

As this document is an informal evaluation scheme methodology, "the tester" denotes both roles in the next sections.

In this document, "the system" denotes the product under evaluation.

1.2 Aim of the document

This document describes the evaluation tasks related to the implementation of cryptographic mechanisms and the secure composition of cryptographic mechanisms that shall be performed during the evaluation of a product implementing cryptographic mechanisms under the SOG-IS Crypto Evaluation Scheme (SCES).

In order to avoid unnecessary repetitions, the document presents evaluation tasks in a hierarchical manner. For example, some tasks are relevant for all symmetric atomic primitives and others are specific to a given block cipher only.

In addition, every evaluation task is framed and identified as follows:

[Mechanism-Name-Number] <Evaluation task description>	Inputs

The **Mechanism** is the cryptographic mechanism targeted by the evaluation task. The **Name** describes the nature of the evaluation task. The **Number** enables referring to several related evaluation tasks for a mechanism, under a single name. "Inputs" refers to the input(s) required for the evaluation task, and could be provided by the developer.

Some tasks are defined with an array of several lines as follows:

[Mechanism-Name-Number] <Evaluation task description>	Inputs
[Mechanism-Name-Number] <Evaluation task description>	Inputs

This means that only one of these tasks are necessary as they intend to evaluate the same point. The task of the first line is recommended. The other lines are alternative tasks. They generally differ in the inputs available.

1.2.1 Inputs

To perform a task, some inputs are required by the tester. The following inputs are considered:

I1: Cryptographic design: an unambiguous description of the mechanism definition and the main characteristics of the implementation(s)

For a mechanism under evaluation that is implemented according to a (recognized) standard specification, the design document can declare compliance with the standard reference. Otherwise, the design document must specify differences with the standard. The document must indicate:

- The used cryptographic protocols (such as TLS) and their specification (for non-standard protocols) or a reference to their specification (for standard-compliant protocols).

- The used cryptographic mechanisms/primitives and their specification (for non-standard mechanisms/primitives) or a reference to their specification (for standard-compliant mechanisms/primitives). For non-standard primitives and/or mechanisms, implementation test vectors with enough details (i.e. sets of input/output or plaintext/cipher text data and internal states) shall be provided. This gives the ability for the tester to implement the primitive/mechanism.
- The exact supported ranges of input and/or output parameter values if some freedom is left by the specification concerning these ranges - for instance the ranges of supported bitlengths in the case of variable lengths binary parameters.
- The significant implementation choices, e.g., software or hardware, bit-, byte-word- oriented, bitslice.
- A specification of the key management concept together with all supporting key management functions and mechanisms such as random number generation, key generation, key distribution, key storage, key wrap, use of certificates, key revocation/renewal, crypto period of key, etc. In particular, specification should include a description of the maximum life expectancy of all keys that are in use.

I1 **also contains Guidance Documentation**

The cryptographic design may not be enough for a complete overview of all cryptographic mechanisms used in the system. Indeed, some parameters may be chosen by a user (generally an administrator) of the system under evaluation and do not belong to the cryptographic design. For example, if the system requires a TLS configuration prior the normal usage of the system, it is likely that the guidance contains information on how to configure the TLS server. Therefore, it is likely that the guidance documentation contains recommendations on the ciphersuites and authentication keys.

The guidance documentation shall contain a description of the system preparation, and how to manage and/or use the system under evaluation.

As the cryptographic design and guidance documentation are complementary for a complete overview of all cryptographic mechanisms used in the system, they belong to the same input I1.

I2: A testing interface of the cryptographic mechanisms

This specifies how the mechanisms can be addressed, allowing to submit input values, such as the key's value in the case of keyed mechanisms. In the case of randomized mechanisms, this testing interface may also possibly allow to submit the values to be used instead of the output of a random generator. The exact formatting conventions for the external interface of the mechanism implementation of the system under evaluation, e.g. whether the input and output parameters are represented as binary strings, byte strings shall be given. I2 also specifies how to collect the corresponding output values. For all mechanisms of the system under evaluation, this interface must allow the tester to directly access the primitives and not only the modes.

The final system under evaluation might not have such interfaces. In fact, such interfaces might be deliberately unavailable for security considerations. It is not mandatory to provide such interfaces in the final system under evaluation. A modified system that permits access to such interfaces is an acceptable input. Sub-systems containing such interfaces, before the system integration, also constitute an acceptable input. In either cases, a detailed description of the differences with the final system is necessary for justification that the tests are valid in the final system.

I3: Implementation representation

For each mechanism in the system under evaluation, the implementation representation is themost abstract representation of the mechanism. The representation is used to create the implementation. This may be software source code, firmware source code, hardware

diagrams and/or hardware design language code or layout data. Source code annotation helping the tester to establish the correspondence between the specification and the implementation is also required.

If the cryptographic implementation is provided by an external library, I3 must include the code calling the library functions. This allows the tester to analyse the correctness of the calls to the library.

Additional information such as compilers, compilation options, scripts, are part of the implementation if they contribute to the generation of the mechanism implementation. The specifications of the programming language shall be provided as well. Build configuration files may also be part of inputs for evaluation as they contribute to the implementation of cryptographic mechanisms.

Output

All the mentioned tasks below aim to conclude on the security of the cryptographic mechanisms used by the product under evaluation and their secure implementation. All the evaluation tasks are detailed in an output report which allow the tester to justify the correct covering of the evaluation tasks.

The outcome of each task shall be either "Pass" or "Fail" with a rationale for justification of the result and evaluation evidences. For instance, if the tester performs source code analysis for completing an evaluation task, the analysed source code sample should be considered in the rationale.

Also, as the tester is either the developer or the evaluator, the output report allows the reader to make sure the evaluation tasks were performed correctly and entirely. For instance, if the developer performs the conformance testing, a simple checklist of "Pass" might not be sufficient. The output report should have to provide the evidence that the conformance tests have been done correctly and entirely.

It is not the purpose of this document to give additional details on the content of the testing report. This document is rather a tasks list with detailed inputs required to execute them.

1.3 Structure of the document

The document is organized as follows:

- Chapter 2 introduces the overall methodology;
- Chapter 3 specifies the evaluation tasks for every considered cryptographic mechanisms, with clarifications;
- Chapter 4 provides a global overview of the evaluation tasks;
- Chapter 5 presents conformity test vectors for KATs and MCT.

1.4 Related Documents

[ACM] "*SOG-IS Crypto Evaluation Scheme Agreed Cryptographic Mechanism*", Version 1.1, June 2018

[AFG+14] "*GLV/GLS Decomposition Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias*", D.F. Aranha, P.-A. Fouque, B. Gérard, J.-G. Kammerer, M. Tibouchi, J.-C. Zapalowicz, ASIACRYPT'14.

[BBD+13] "*Differential Power Analysis of HMAC SHA-2 in the Hamming Weight Model*", S. Belaïd, L. Bettale, E. Dottax, L. Genelle, F. Rondepierre, SECRYPT'13.

[Ble98] "*Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS#1*", D. Bleichenbacher, CRYPTO'98

[BM06] "*New Attacks on RSA with Small Secret CRT-Exponents*", D. Bleichenbacher, A. May, PKCS'06

[CJ05] "*Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults*", B. Chevallier-Mames, M. Ciet, M. Joye, Des. Codes Cryptography vol. 36, iss. 1, 2005

[CJNP00] "*New Attacks on PKCS#1 v1.5 Encryption*", J.-S- Coron, M. Joye, D. Naccache, P. Paillier, EUROCRYPT'00

[CMACVS] "*The CMAC Validation System (CMACVS)*", NIST, Updated August 2011

[DRBGVS] "*The NIST SP 800-90A Deterministic Random Bit Generator Validation System* (DRBGVS)", NIST, october 2015

[EAX-03] *"EAX: A Conventional Authenticated-Encryption Mode",* M. Bellare, P. Rogaway, D. Wagner, September 2003, https://eprint.iacr.org/2003/069

[EAX-04] *"The EAX Mode of Operation (A Two-Pass Authenticated-Encryption Scheme Optimized for Simplicity and Efficiency)",* M. Bellare, P. Rogaway, D. Wagner, January 2004, http://web.cs.ucdavis.edu/~rogaway/papers/eax.pdf

[FIPS 186-4] "*Digital Signature Standard (DSS) - FIPS PUBS 186-4*", FIPS, November 2001

[FIPS 197] "*Specification for the Advances Encryption Standard (AES) - FIPS PUBS 197*", FIPS, July 2013

[GCMVS] "*The Galois/Counter Mode (GCM) and GMAC Validation System (GMACVS) with the Addition of XPN Validation Testing*", NIST, June 2016

[ISO18031] "*Security techniques – Random bit Generation*", ISO/IEC 18031, 2011

[ISO18033-3] "*Security techniques – Encryption algorithms – Part 3: Block ciphers*", ISO/IEC 18033-3, 2010

[JIL13-1] *"Application of Attack Potential to Smartcards – Joint Interpreation Library",* SOGIS version 2.9, January 2013

[JIL13-2] *"Attack Methods for Smartcards and Similar Devices - Joint Interpretation Library",* SOGIS version 2.2, January 2013

[JSS15] *"Practical Invalid Curve Attacks on TLS-ECDH"*, J. Jager, J. Schwenk, J. Somorovsky, ESORICS'15

[L13] "*Practical malleability attacks against CBC-Encrypted LUCKS partitions", J.* Lell, 2013

[Man01] *"A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0",* J. Manger, CRYPTO'01

[NESSIE] "*Test vectors for the NESSIE candidates and other cryptographic primitives*", https://www.cosic.esat.kuleuven.be/nessie/testvectors/

[PKCS#1 v2.2] "*PKCS#1 v2.2: RSA Cryptography Standard*", RSA Laboratories, October 2012

[PQ03] "*A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD*", G. Piret, J.-Q. Quisquater, CHES'03

[RFC 2898] "*PKCS #5: Password-Based Cryptography Specification Version 2.0*", RSA Laboratories, September 2000

[RFC 3394] "*Advanced Encryption Standard (AES) Key Wrap Algorithm*", September 2002

[RFC 3526] "*More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*", May 2003

[RFC 3610] "*Counter with CBC-MAC (CCM)*", https://tools.ietf.org/html/rfc3610

[RFC 5297] "*Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*", https://tools.ietf.org/html/rfc5297

[Rog95] "*Problems with proposed IP Cryptography*", http://web.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt, April 1995

[SEC1] "*Standard for Efficient Cryptography (SEC 1): Elliptic Curve Cryptography*", v2.0, May 2009

[SHA3VS] "*The Secure Hash Algorithm 3 Validation System (SHA3VS)*", NIST, April 2016

[SHAVS] "*The Secure Hash Algorithm Validation System (SHAVS)*", NIST, July 2014

[SP800-38A] "*Recommendations for Block Cipher Modes of Operation: Methods and Techniques*", NIST Special Publication 800-38A, December 2001

[SP800-38C] "*Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*", NIST Special Publication 800-38C, May 2004

[SP800-38D] "*Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*", NIST Special Publication 800-38D, November 2007

[SP800-38F] "*Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*", NIST Special Publication 800-38F, December 2012, https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/mac/kwtestvectors.zip

[SP800-56A] "*Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*", NIST Special Publication 800-56A Revision 2, May 2013

[SP800-56B] "*Recommendation for Pair-Wise Key-Establishment Schemes Using Integer Factorization Cryptography*", NIST Special Publication 800-56B Revision 1, September 2014

[SP800-56C] "*Recommendation for Key Derivation through Extraction-then-Expansion*", NIST Special Publication 800-56C, November 2011

[SP800-67r2] "*Recommendations for the Triple Data Encryption Algorithm (TDEA) Block Cipher*", NIST Special Publication 800-67 Revision 2, November 2017

[SP800-90Ar1] "*Recommendations for Random Number Generation Using Deterministic Random Bit Generators*", NIST Special Publication 800-90A, Revision 1, June 2015

[SP800-132] "*Recommendations for Password-Based Key Derivation",* NIST Special Publication 800-132, December 2010

[Vau02] "*Security Flaws Induced by CBC Padding Applications to SSL, IPsec, WTLS, …*", S. Vaudenay, EUROCRYPT'02

[XTSVS] *"The XTS-AES Validation System (XTSVS)"*, S.S. Keller, T.A. Hall, Septemb**e**r 2013, https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/aes/XTSTestVectors.zip

1.5 Acronyms

ACM	Agreed Cryptographic Mechanism
CC	Common Criteria
CRT	Chinese Remainder Theorem
DRG	Deterministic Random Generator
EC	Elliptic Curve
ECSM	EC Scalar Multiplication
ESSIV	Encrypted Salt-Sector Initialization Vector
IC	Integrated Circuit
KAT	Known Answer Test
KEM	Key Encapsulation Mechanism
MAC	Message Authentication Code
MCT	Monte-Carlo Test
SCA	Side-Channel Attack
SCES	SOG-IS Crypto Evaluation Scheme

2 Evaluation of cryptographic mechanisms

The objective of this section is to give an overview of the evaluation tasks that can be performed while evaluating cryptographic mechanisms. The tasks are broadly outlined. They are refined into actionable tasks in section 3.

2.1 Checking that all tested mechanisms are agreed

It is assumed here that the context of the evaluation (e.g. a CC evaluation under SOG-IS MRA) allows the tester to identify a crypto evaluation perimeter within the evaluated product where all cryptographic mechanisms of the system under the evaluation must be ACMs. With I1, the first task for the tester is to identify the cryptographic mechanisms used by the product within this perimeter and then check that all these cryptographic mechanisms are agreed as specified in .

In order to consider a cryptographic mechanism within the SOG-IS Crypto Evaluation scheme, its cryptographic robustness shall be evaluated by SOG-IS certification bodies, and recognised as agreed. lists all the agreed mechanisms. In the general case, this evaluation task consists in checking that the considered cryptographic mechanism is included in these lists and that all the parameter sizes or options retained for that mechanisms satisfy all the requirements expressed in [ACM].

In addition to this task, the tester shall identify for each keyed mechanism how the key is generated in order to analyse its cryptographic robustness. In particular, the tester shall check that the key generation is agreed by [ACM].

2.2 Conformance testing

The tester shall perform conformance testing tasks for all implemented instantiations of all cryptographic mechanisms used in the system. The tasks, for each mechanism, are described in Section 3.

According to this document, the objective of conformance evaluation is to establish with a high degree of confidence that a (non-malicious) implementation correctly implements the cryptographic mechanisms being evaluated.

Three different kinds of testing methods, with different purposes, are described below.

2.2.1 Random Test/Monte-Carlo Test (MCT)

A cryptographic function is (pseudo-)randomly tested by applying it iteratively, starting from a seed. The large number of iterations enables to test the function against a large number of pseudo-random values. The seed is either predefined or randomly generated when starting the MCT. In the former case, the final result is compared against a predefined reference value. In the latter case, an MCT, with the same seed, is run with a reference implementation of the function; the two final results are then compared. In both cases, the outcome is obtained from the result of the comparison.

MCT aims to intensively test cryptographic primitives such as:

- block ciphers;
- hash functions with fixed length inputs, to validate the fixed-length building block of hash functions;
- modular exponentiations;
- EC scalar multiplications and multi scalar multiplications;
- multiplications in $GF(2^{128})$.

In order to do so, the tested function f is applied to values that are updated by applying encoding functions to the latest outputs of f. The random seed of the MCT provides initial values, possibly together with enough pseudo-outputs of f that can be used to bootstrap the iteration.

For keyed algorithms, f is applied on a key and an input; and the iteration consists in two nested loops. The input is updated between each inner loop while the key remains unchanged. The key part is updated between each outer loop. The number of iterations of the outer loop, **N_key,** and the number of iterations of the inner loop, **N_input,** are chosen for each mechanism in such a way that every program branch, and every constant of the mechanism have been tested with very high probability. **N_key** and **N_input** must be greater than one for testing the function against at least two different keys and two different inputs (for all keys).

An **encode_input** function, resp. **encode_key** function, forms an updated input, resp. key, from one or several outputs of f. If the encoding function requires several outputs, the **first_outputs** argument provide values to be used in place of f outputs to bootstrap the iteration.

Thus, MCT calls two helper functions:

- **encode_key**: a deterministic function that converts outputs of the function f to a valid key.
- **encode_input**: a deterministic function that converts outputs of the function f to a valid input.

Several outputs might be required for **encode_key** to gather enough entropy for generating a valid key. The number of outputs (nb_for_key) required will be specified for each primitive as long as this function. For example, an AES-256 key needs two outputs of 128 bits.

The same goes for **encode_input**. Several iterations might be required to generate enough outputs for **encode_input**. The number of outputs (nb_for_input) required will be specified for each primitive.

The specifications of encode_input and **encode_key** depend on the primitive under consideration, and are detailed later in this document.

The following algorithm describes the generic MCT for keyed primitives (e.g. block ciphers, modular exponentiation and EC scalar multiplication[1]):

[1] The exponent of modular exponentiation, and the scalar of scalar multiplication can be considered as keys.

Require: N_key, N_input, nb_for_key, nb_for_input, first_key, first_input, first_outputs (if necessary) which is an array of outputs.
Output: an output of f.
Algorithm:

```
ctr = 0 // Number of outputs
K = first_key
input = first_input
output = [] // an array of outputs.
            // For i ≥ 0, output[i] is the i^th output of the function f

if length(first_outputs)>0 // if first_outputs are provided
    for i = 1 until i= length(first_outputs) do
        // For i < 0, output[i] is provided by first_outputs
        output[−i] = first_outputs[i]

do N_key times
    do N_input times
        output[ctr] =f(K, input)
        ctr = ctr + 1
        input = encode_input(output[ctr−1], …, output[ctr−nb_for_input])
    input = first_input
    K = encode_key(output[ctr−1] , …, output[ctr−nb_for_key)])

return output[ctr−1] // return last output
```

N_key and **N_input** are chosen for each mechanism in such a way that every program branch, and every constant of the mechanism have been tested with very high probability. **N_key** and **N_input** must be greater than one for testing the function against at least two different keys and two different inputs (for all keys).

Note that the last **encode_input** of each inner loop and the very last **encode_key** executions are unnecessary since the output is not used afterwards. This simplifies the algorithm description but the tester can take this note into account and avoid these unnecessary executions.

For unkeyed primitives (e.g., hash functions and multiplication in $GF(2^{128})$), this algorithm is adapted by removing the outer loop :$GF(2^{128})$

Require: N_input, nb_for_input, first_input, first_outputs (optional) which is an array of outputs.
Output: an output of f
Algorithm:

```
ctr = 0 // Number of outputs
input = first_input
output = [] // an array of outputs.
            // For i ≥ 0, output[i] is the i^th output of the function f

if length(first_outputs)0 // if first_outputs are provided
    for i = 1 until i = length(first_outputs) do
        // For i < 0, output[i] is provided by first_outputs
        output[−i] = first_outputs[i]

do N_input times
    output[ctr] =f(input)
    ctr = ctr + 1
    input = encode_input(output[ctr−1], …, output[ctr−nb_for_input])
```

return output[ctr−1] // return last output

2.2.2 Known Answer Test (KAT)

This type of tests consists in applying the cryptographic mechanism to a fixed set of input(s) and in comparing its output(s) to the expected corresponding output(s).

The MCT method is useful as it permits to intensively test a primitive. However, by construction, only inputs of constant size and with unpredictable contents are submitted to the primitive. KATs can complement MCT, by allowing to test a particular behaviour of the mechanism by submitting specific inputs.

2.2.2.1 Correct behaviour of the implementation

KATs may contain non-specific inputs for a first rough check of the correct behaviour of the implementation. Those tests are particularly useful for complex mechanisms, such as Asymmetric Constructions, relying on several other primitives.

2.2.2.2 Length tests

These tests are relevant when the cryptographic mechanism considered is a variable input length mechanism (e.g. hash functions or an encryption mechanism), or variable output length mechanism (e.g. MAC with truncation). These tests allow the correctness verification of the chaining mechanism implementation, as well as the padding implementation. When the range of input length is limited by the implementation, such limits shall also be checked through appropriate length tests.

2.2.2.3 Corner cases

MCT, non-specific KATs and different input length KATs all permit to test the normal behaviour of the mechanism. However, some specific cases within a mechanism are very unlikely to happen by chance. This is particularly the case with schemes that need to verify a padding correctness. Some specific inputs are designed to trigger those corner cases, either positive or negative.

2.2.3 Source code analysis

As mentioned above, source code analysis shall be used to complete conformance evaluation, especially to identify or confirm the underlying primitives and mechanisms. For instance, source code analysis permits the tester to:

- check that a refresh of the random generator is done;
- check that intermediate random values within a mechanism (for instance for paddings schemes) are correctly generated.

Source code analysis also provides an alternative way to verify that the implementation is compliant with the specifications by direct inspection. For instance, source code analysis permits the tester to:

check that a refresh of the random generator is done;

check that intermediate random values within a mechanism (for instance for paddings schemes) are correctly generated. The KATs described above permit to test the general behaviour of the implementation, with different input sizes and corner cases in a quick and exhaustive manner. The tester gains assurance on the primitive/mechanism by running the KATs. However, the tester needs access to itsthe primitive mechanism interface (I2). If this interface is not easily available but the implementation representation (I3) is, then the tester can gain assurance on the primitive/mechanism conformance with an implementation representation analysis. In this case, the tester can use KATs provided to be reminded to check if each specific test is taken into account.

Note that MCT cannot be replaced by an implementation representation inspection as the implementation representation of the primitive is overly complex. Indeed, it seems not to

be possible to establish the conformance of a modular exponentiation only by an implementation representation inspection.

2.3 Robustness Analysis

2.3.1 Implementation Representation Analysis

In addition to being usable for conformance testing purposes, implementation representation analysis can allow to identify some vulnerabilities. For instance, source code analysis permits the tester to:

- identify the entropy sources gathered by a random number generator for instantiation and reseed procedures;
- support the correct erasure of sensitive data buffers (e.g. to avoid cold boot attacks).

2.3.2 Avoidance of Implementation Pitfalls

The implementation of cryptographic mechanisms is a sensitive process, where several errors might be introduced. Over the course of academic research and evaluation efforts, common implementation errors have been identified, and their impact on the security of the mechanism assessed. These errors may be linked to non-conformities of the implementation, or to bad implementation choices of mechanisms weakening the security of the mechanism. lists such implementation pitfalls for agreed mechanisms.

Note: The distinction between conformance evaluation and pitfalls detection is sometimes tenuous, since non-conformity can be security impacting and result in a vulnerability.

Testing tasks shall ensure that these known implementation pitfalls are avoided.

2.3.3 Resistance to Leakage and Perturbation Attacks

Side-channel and Perturbation attacks are defined as follows in .

«*Side-channel attacks target secret information leaked through unintentional channels in a concrete, i.e. physical, implementation of an algorithm. These channels are linked to physical effects such as timing characteristics, power consumption, or electromagnetic radiation.*

Perturbation attacks change the normal behaviour of an Integrated Circuit (IC) in order to create an exploitable error in the operation of a TOE. The behaviour is typically changed either by operating the IC outside its intended operating environment (usually characterised in terms of temperature, Vcc and the externally supplied clock frequency) or by applying one or more external sources of energy during the operation of the IC. These energy sources can be applied at different times and/or places on the IC.»

The document is an evaluation guidance related to Side-channel and Perturbation attacks in the context of smartcards and similar devices. The concept of side-channel attacks can be expanded to include attacks leveraging the leakage of sensitive information on logical interfaces, e.g. through status or error messages, or by design of a cryptographic mechanism, e.g. the mechanism behaves in a biased way. Leakage and perturbation attacks are not restricted to the smartcard domain and may affect any implementation domain. The present document provides general notes on sensitive data within mechanism implementations that shall not be disclosed (using, for example a Side-channel) to an attacker. A data is said *sensitive* in the sense of confidentiality and/or integrity. The amount of information on such data that an attacker could recover through Side-Channel or Perturbation attacks shall be as small as possible[2]. Also, such sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

[2] It is not possible to get a "zero leakage" implementation. For many evaluation procedures, an attacker level is defined. The term "as small as possible" would be refined

in those procedures. This document is not intended to define what a small amount of information is.

3 Specific cryptographic mechanisms evaluation tasks

3.1 Symmetric Atomic Primitives

3.1.1 Block Ciphers

[BlockCipher-AgreedMechanism-1] The tester shall verify that the tested block cipher is agreed .	I1

The following block ciphers are agreed in : AES-128, AES-192, and AES-256 specified in , and Two-Key-3DES and Three-Key-3DES specified in .

The following table summarises the parameters for each agreed symmetric block cipher.

Symmetric block cipher	Key size (in bits)	Block size (in bits)
AES	128 bits	128
	192 bits	
	256 bits	
Triple-DES	112 bits	64
	168 bits	

[BlockCipher-ConformanceTesting-1] The tester shall test all key lengths of the primitive being used in the system.	I1 I2

Analysis: The key length of a given block cipher may have an impact on the block cipher implementation. For instance, the AES key schedule varies depending on the key length. As a consequence, the variants of the algorithm should be considered as independent algorithms, and should all be tested.

Recommendation: Even if a key length is not used in the system under evaluation, it should be tested as long as it is supported or implemented in the system. Indeed, it is unsafe to keep an untested unused function in a product. If the function is activated later during a system evolution, the function would not have been tested.

[BlockCipher-ConformanceTesting-2] The tester should test all the directions of the block cipher operation being used in the system.	I1 I2

Analysis: the decryption operation generally differs substantially from the encryption operation, and needs to be separately tested.

Recommendation: Even if a direction is not used in the system under evaluation, it is recommendedto test it as long as it is supported or even implemented in the product. Indeed, it is unsafe to keep an untested unused function in a system. If the function is activated later during a system evolution, the function would not have been tested.

[BlockCipher-ConformanceTesting-3] The tester shall perform the MCT, detailed in the section below for each primitive.	I2

Analysis: MCT aims to test the block cipher primitive. MCT described in section 2.2.1 of this document shall be used. A refinement for each agreed block cipher is proposed in the remainder of this section.

Sensitive data within Block Ciphers implementations

The encryption or decryption key is a sensitive data.

Depending on the mechanism using the block cipher, the plaintext of the block cipher and/or the ciphertext are sensitive. This will be specified in the sections related to these mechanisms.

In addition, the knowledge of the following sensitive data permits to derive the encryption key:

- Each round key.
- Temporary variables such as the state after each round are sensitive since their disclosure permits to perform a cryptanalysis on the block cipher with a smaller number of rounds. As a consequence, the round counter is also a sensitive data in the sense of integrity, since perturbation of the round counter may lead to the disclosure of temporary variables.
- Erroneous ciphertexts are sensitive data because they can be used to derive the key. The number of erroneous ciphertexts required by the attacker depends on the fault model. For AES encryption, a single incorrect ciphertext isenough to recover sensitive data if the faults are very accurate .

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

AES

In the case of the AES blockcipher, **[BlockCipher-ConformanceTesting-3]** is achieved by applying the MCT described in section 2.2.1 with the following details.

The functions **encode_input** and **encode_key** are defined as follows:

- **encode_input** is the identity function; nb_for_input=1 (**first_outputs** is void);
- **encode_key** is:
  - the identity function for AES-128; nb_for_key=1;
  - the concatenation of the last output and the first 64 bytes of the penultimate output of AES-192; nb_for_key=2;
  - the concatenation of the last two outputs AES-256; nb_for_key=2.
- N_key=5500. This number has been chosen[3] as to ensure that the probability that all possible inputs of the S-box function are covered for each round key computation is greater than $1-\frac{1}{2^{80}}$.
- N_input=2 in order to ensure that the f function is applied with a different input for one iteration (the input is set to first_input at the end of the inner iteration).

Sensitive data within AES implementations

There are no additional sensitive data other than the ones specified for Block Ciphers.

Triple-DES

In addition to the evaluation tasks mentioned in §3.1.1, this section provides additional procedures regarding Triple-DES.

In the case of Triple-DES; **[BlockCipher-ConformanceTesting-3]** is achieved by applying the MCT described in section 2.2.1 with the following details.

The functions **encode_input** and **encode_key** are defined as follows:

- **encode_input** is the identity function; nb_for_input=1 (**first_outputs** is void);
- **encode_key** is:

[3] This is the sampling with replacement problem. Rather some variant of the coupon collector problem. Anyway this reference is insufficient.

- the concatenation of the last two outputs, where the parity bit is replaced by the right value for each byte; nb_for_key=2 for Triple-DES with two keys;
- the concatenation of the last three outputs, where the parity bit is replaced by the right value for each byte; nb_for_key=3 for Triple-DES with three keys.

- N_key=1325. This number has been chosen[4] as to ensure that the probability that all possible inputs of the S-box function are covered for each round key computation is greater than $1 - \frac{1}{2^{80}}$.
- N_input=2 in order to ensure that the f function is applied with a different input for one iteration (the input is set to first_input at the end of the inner iteration). Indeed, if N_input=1, the new key and the new input are both equal to the last output.

[3DES-ImplementationPitfall-1] The tester shall verify that the number of blocks processed by the block cipher with the same key is limited to 2^{27} for Triple-DES with three keys and 2^{20} for Triple-Des with two keys.	I1 or I3

Analysis: This is to avoid collision linked to the block size of the Triple-DES (64 bits). A limit exists as well for AES but it is so high that no care is needed. The limit is lowered to 2^{20} for Triple-DES of two keys because of existing attacks .

Sensitive data within Triple-DES implementations

This paragraph specifies sensitive data for Triple-DES in addition to the ones specified for Symmetric Atomic Primitives.

Triple-DES encryption algorithm applies the simple DES algorithm three times (in encryption and decryption modes). The output of the first and second DES algorithm, which are also respectively the input of the second and third DES algorithm application, as particular temporary variables, are sensitive.

3.1.2 Stream Ciphers

No stream cipher is currently agreed in .

3.1.3 Hash Functions

[HashFunctions-AgreedMechanism-1] The tester shall verify that the used hash function is agreed .	I1

The following hash functions are agreed in : SHA-2 family, SHA3 with 256, 384 and 512-bit digest lengths.

The functions of the SHA-2 and SHA-3 families, with digest length greater or equal to 256 bits, are recommended ACMs . The functions SHA-224 and SHA-512 truncated to 224 bits are legacy ACMs . SHA-1 is not an agreed hash function but in the MAC construction HMAC-SHA-1, it is accepted as a legacy underlying function .

Hash functions process an input message by splitting it into fixed size blocks. The following table summarises the block size for each agreed hash function. The implementation of a hash function must be able to correctly generate message digests for messages that span multiple data blocks.

Function	digest length *d* (in bits)	block size *m* (in bits)
SHA-224	224	512
SHA-256	256	512

[4] This is the sampling with replacement problem.

SHA-384	384	1024
SHA-512	512	1024
SHA-512/224	224	1024
SHA-512/256	256	1024
SHA3-256	256	1088
SHA3-384	384	832
SHA3-512	512	576

[HashFunctions-ConformanceTesting-1] The tester shall test each hash function used in the system.	I1 I2

Analysis: Hash function implementations may vary depending on the length of the digest output.

[HashFunctions-ConformanceTesting-2] The tester shall perform the MCT defined below.	I2

The tester shall use the MCT without key described in section 2.2.1 of this document, with:

- f being the hash function;
- **encode_input** is the concatenation nb_for_input outputs of the function f truncated in order to get an input with a size exactly equal to the hash function block size;
- nb_for_input=2 for SHA-256, SHA-512 and SHA3-512; nb_for_input=3 for SHA-224 and SHA-384, SHA3-384; nb_for_input=4 for SHA-512/256; nb_for_input=5 for SHA-512/224 and SHA3-256.
- N_input to determine

[HashFunctions-ConformanceTesting-3] The tester shall perform KATs or the methodology of §5.1.1 related to hash functions.	I2
[HashFunctions-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.1 for source code analysis, as §5.1.1 may contain some specific cases.	I3

Sensitive data within Hash Functions implementations

The input of the hash function is generally a sensitive data.

Depending on the mechanism that relies on this primitive, the output of the hash function can also be a sensitive data. For example, if the hash function is used to derive an encryption key, the output is obviously sensitive. This will be specified in the sections related to mechanisms relying on a hash function.

Also, the knowledge of internal states during the execution of the hash function implementation can be used to derive the input or the output.

If relevant, the amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

SHA-2 Family

There is no additional task specific to SHA-2.

Sensitive data within SHA-2 implementations

There is no additional data other than the ones specified for Hash functions.

SHA-3 Family

There is no additional task specific to SHA-3.

Sensitive data within SHA-3 implementations

There are no additional data other than the ones specified for Hash functions.

3.1.4 Secret Sharing

Secret sharing schemes aim to distribute a shared secret among several parties in the form several *key shares*.

[SecretSharing-AgreedMechanism-1] The tester shall verify that the used secret sharing mechanism is agreed .	I1

Shamir's scheme is the only agreed secret sharing scheme.

[SecretSharing-ConformanceTesting-1] The tester shall perform KATs or the methodology of §5.1.1 related to secret sharing.	I2
[SecretSharing-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.1 for implementation representation analysis, as §5.1.1 may contain some specific cases.	I3

[SecretSharing-ConformanceTesting-2] The tester shall test sharing generation scheme.	I2

[SecretSharing-ConformanceTesting-2] The tester shall test sharing recombination scheme.	I2

Sensitive data within Secret Sharing implementations

The sensitive data are the shared secret and each individual share.

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

Shamir's secret sharing

No additional tasks for the Shamir's secret sharing is provided in addition to those mentioned in §3.1.4.

3.2 Multiplication in $GF(2^{128})$

For some mechanisms, multiplications in the field $GF(2^{128})$ are required (GCM and GMAC).

[MultiplicationGF2128-ConformanceTesting-1] The tester shall perform the MCT defined below.	I2

The unkeyed MCT, described in section 2.2.1 of this document, shall be used, for the multiplication in $GF(2^{128})$ with the following features:

- the input of the function f is a pair of two elements in $GF(2^{128})$;
- f is the multiplication of the two elements of the pair;

- nb_for_input=2 (**first_outputs** contains one element) and **encode_input** is simply the construction of a pair (a, b) with a being the penultimate output and b the last output; at each iteration, a is the element b of the previous iteration and b is the output of the last iteration.

Sensitive data within Multiplication in $GF(2^{128})$ implementations

Depending on the mechanism that relies on this primitive, the sensitive data can be one or both operands of the multiplication, or the result of the multiplication. This will be specified in the sections related to such mechanisms. The knowledge of two of these variables permits to trivially derive the third.

If relevant, the amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

3.3 Symmetric Constructions

3.3.1 Symmetric Encryption (Confidentiality Only)

A symmetric encryption mechanism is built around a primitive, either a block cipher or a stream cipher. All agreed encryption mechanism use a block cipher, with an encryption mode of operation.

[SymmetricEncryptionConfOnly-AgreedMechanism-1] The tester shall verify that the symmetric encryption scheme is agreed .	I1

The following Symmetric Encryption schemes are agreed in : CBC, CBC-CS, CTR, OFB, CFB. They do not ensure the integrity of the protected data.

A symmetric encryption mode is defined by a block cipher, a mode of operation, and possibly a padding scheme. The padding scheme adds data to a plaintext of arbitrary length to ensure that its size becomes a multiple of the block size. The padding scheme is generally used for the CBC mode. Indeed, a padding is unnecessary for the other agreed modes since the last cipher block can be truncated for CTR, OFB and CFB (the decryption operation does not require a full final block for decryption), and CBC-CS has been especially designed to dispense the use of padding.

[SymmetricEncryptionConfOnly-AgreedMechanism-2] The tester shall verify that the block cipher used by the symmetric encryption scheme is agreed.	I1

[SymmetricEncryptionConfOnly-AgreedMechanism-3] The tester shall analyse the rational of the absence of ciphertext integrity provided by the developer to determine if the scheme does not bring any threat to the assets of the system.	I1

Analysis: Absence of ciphertext integrity may be a critical threat in some systems. The tester shall verify that the attacker cannot modify the ciphertexts or that any modification of ciphertexts does not comprise the system. This is a general task that is not more detailed this task strongly depends on the system being tested. An example of such vulnerabilities are the EFAIL vulnerabilities.

[SymmetricEncryptionConfOnly-ConformanceTesting-1] The tester shall perform the conformance tests of the underlying block cipher.	I1 I2

[SymmetricEncryptionConfOnly-ConformanceTesting-2] The tester should test all the directions of operation, encryption or decryption, used in the system.	I1 I2

Analysis: The analysis is the same as **[BlockCipher-ConformanceTesting-2]**.

[SymmetricEncryptionConfOnly-ConformanceTesting-3] The tester shall perform KATs or the methodology of §5.1.3.1 related to symmetric encryption mechanisms.	I2
[SymmetricEncryptionConfOnly-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.3.1 for implementation representation analysis, as §5.1.3.1 may contain some specific cases.	I3

[SymmetricEncryptionConfOnly-ImplementationPitfall-1] The tester shall determine whether the implementation ensures that the expected property of the Initial Vector or the counters is satisfied. For CBC, CBC-CS and CFB modes, the IV must be *unpredictable*. For the OFB mode, the IV must only be unique for each encryption process. In this case, the IV is called *nonce*. For the CTR mode, the counters shall be generated in such a way that all counters are unique **across all plaintexts** (and not only within a single plaintext).	I1 or I3

Analysis: For OFB, reusing a nonce with the same key voids the confidentiality guarantee of the mode. For the CTR mode, the standard specifies input blocks that are encrypted to produce the key stream. Using a counter twice also voids the confidentiality guarantee of the mode. Appendix B of recommends two methods to generate such counters. Also, the generation method of the first counter and the next counters for GCM is more formal and accurate. This method can be used for the CTR mode as well. In this case, the first counter is generated from the concatenation of an IV and a counter buffer initialized with zero. For this method, unicity of the IV guarantees the unicity of all counters (if there is no overflow in the counter buffer, see **[GCM-AgreedMechanism-1]** and **[GCM-ImplementationPtifalls-1]**).

For CBC, CBC-CS and CFB, reusing an IV with the same key leaks some information about the first block of the message. In addition, it must be unpredictable . recommends, in Appendix C, two methods to generate unpredictable IV. The first method is to generate an IV by encrypting a nonce with the same key that is used for the encryption of the plaintext. The second method is to generate a random IV with a strong DRG.

Over all, especially for stream modes of operation, care shall be taken when a nonce or IV is chosen at random. This is due to the birthday paradox. As an example, if a nonce or IV of size 64 bits is chosen (for example for Triple-DES, or if a random nonce of size 64 bits is concatenated with a 64 bits buffer block), then the probability of collision is greater than 2^{-32} after 92681 draws. The number of draws correspond to the number of messages being encrypted. Thus the possible number of messages encrypted with the same key shall be analysed to conclude for this task. For 96 bits, the probability that a collision occurs is greater than 2^{-32} after 6074000999 draws, which can be an issue for very long lifetime keys.

Determining whether each nonce is unique and each IV is unique **and** unpredictable is not achievable through known answer tests. It rather requires an analysis of the cryptographic specifications rationale of the implementation, possibly supported by a source code analysis.

[SymmetricEncryptionConfOnly-ImplementationPitfall-2] The tester shall verify that an attacker has no access to the correctness of the padding or format of the deciphered ciphertext[5] of arbitrary ciphertexts, chosen by the attacker.	I1 or I3

Analysis: This applies in particular to schemes with paddings and the attack is then called a *padding oracle attack* . However, this task more generally applies to all schemes if any verification is performed on the format of the deciphered ciphertext. In this case, the attack is called *format oracle attack*. Such attacks can be used to decrypt any ciphertext. It is pointed out that these attacks can be mounted if the attacker is able to submit arbitrary data to be deciphered.

[SymmetricEncryptionConfOnly-ImplementationPitfall-3] If the underlying block primitive is the Triple-DES, the tester shall verify that the number of blocks processed by the block cipher with the same key is limited to 2^{27} for Triple-DES with three keys and 2^{20} for Triple-DES with two keys.	I1 or I3

Analysis: This results from **[3DES-ImplementationPitfall-1]**. Indeed, for all modes described below, except CTR, the input of each encryption block is unpredictable. Hence, this is to avoid any collision. This number has to be taken into account **across all messages** encrypted with the same key. Hence, the size of messages being encrypted **and** the number of messages have to be taken into account in the calculation.

Sensitive data within Symmetric Encryption (Confidentiality Only) implementations

Symmetric Encryption relies on a block cipher where the key and the plaintext are both sensitive. Hence, the tester can refer to §3.1.1 for sensitive data related to block ciphers.

In addition, as described in the task **[SymmetricEncryptionConfOnly-ImplementationPitfall-2]**, information on the deciphered ciphertexts format of arbitrary ciphertexts may be used to recover the plaintext of any ciphertext. Such information shall not be accessible to an attacker either by direct error code, as stated in the task, or by Side-Channel attacks, particularly timing attacks.

Furthermore, as mentioned in [**SymmetricEncryptionConfOnly-ImplementationPitfall-1]**, some properties have to be satisfied by initialization vectors and counters on the encryption side. As a consequence, they constitute sensitive data in the integrity sense.

CTR

In addition to the evaluation tasks mentioned in §3.3.1, this section provides additional procedures regarding CTR.

[CTR-ImplementationPitfall-1] The tester shall analyse the counter block and verify that it is unique for each plaintext block that is encrypted under a given key.	I1 or I3

Analysis: Appendix B.2 provides two methods to select the initial counter value for the counter mode as to satisfy this uniqueness requirement. **[SymmetricEncryptionConfOnly-ImplementationPitfall-1]** can be ignored because of **[CTR-ImplementationPitfall-1]**.

Sensitive data within CTR implementations

[5] The term "deciphered ciphertext" has been deliberately chosen. It corresponds to the output blocks when applying the decryption mode of operation. It is equal to the "plaintext" if no padding is used. It is equal to the "padded plaintext" if a padding is used in the scheme, with possibly an incorrect padding.

This paragraph specifies sensitive data for CTR in addition to the ones specified for Symmetric Encryption.

CTR is a stream mode of operation. Consequently, each output of the block cipher is a sensitive data as its knowledge permits to trivially derive the plaintext. Thus, this shall be considered when referring to §3.1.1.

Since a key stream is generated that is XORed to the plaintext during encryption, the inputs of the XOR operations are sensitive. During decryption, one input (the key stream) and the output of the XOR operations are sensitive.

OFB

There is no additional task specific to OFB.

Sensitive data within OFB implementations

This paragraph specifies sensitive data for OFB in addition to the ones specified for Symmetric Encryption.

OFB is a stream mode of operation. Consequently, each output of the block cipher is a sensitive data as their knowledge permit to trivially derive the plaintext. Thus, this shall be considered when referring to §3.1.1.

As the CTR mode, since a key stream is generated that is XORed to the plaintext during encryption, the inputs of the XOR operations are sensitive. During decryption, one input (the key stream) and the output of the XOR operations are sensitive.

CBC

There is no additional task specific to CBC.

Sensitive data within CBC implementations

There are no additional sensitive data other than the ones specified for Symmetric Encryption.

CBC-CS

There is no additional task specific to CBC-CS.

Sensitive data within CBC-CS implementations

There are no additional sensitive data other than the ones specified for Symmetric Encryption.

CFB

There is no additional task specific to CFB.

Sensitive data within CFB implementations

This paragraph specifies sensitive data for CFB in addition to the ones specified for Symmetric Encryption.

CFB is a self-synchronizing stream mode of operation. Hence, as for stream modes of operations, each output of the block cipher is a sensitive data as its knowledge permits to trivially derive the plaintext. Thus, this shall be considered when referring to §3.1.1.

As stream modes of operation, since a key stream is generated that is XORed to the plaintext during encryption, the inputs of the XOR operations are sensitive. During

decryption, one input (the key stream) and the output of the XOR operations are sensitive.

3.3.2 Symmetric Disk Encryption

[DiskEncryption-AgreedMechanism-1] The tester shall verify that the disk encryption scheme is agreed .	I1

The following Disk Encryption schemes are agreed in : XTS and CBC-ESSIV.

[DiskEncryption-AgreedMechanism-2] The tester shall verify that the underlying primitives used by the disk encryption scheme are agreed.	I1

[DiskEncryption-ConformanceTesting-1] The tester shall perform the conformance tests of the underlying primitives.	I1 I2

[DiskEncryption-ConformanceTesting-2] The tester should test all the directions of operation, encryption or decryption, used by the system.	I1 I2

[DiskEncryption-ConformanceTesting-3] The tester shall perform KATs or the methodology of §0 related to disk encryptions.	I2
[DiskEncryption-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §0 for implementation representation analysis, as §0 may contain some specific cases.	I3

[DiskEncryption-ImplementationPitfall-1] If the underlying block primitive is the Triple-DES, the tester shall verify that the number of blocks processed by the block cipher with the same key is limited to 2^{27} for Triple-DES with three keys and 2^{20} for Triple-DES with two keys.	I1 or I3

Analysis: This results from **[3DES-ImplementationPitfall-1]**. Indeed, for all modes described below, the input of each encryption block is unpredictable. Hence, this is to avoid any collision.

[DiskEncryption-ImplementationPitfall-1] The tester shall determine whether the implementation ensures that the tweak value (for XTS) or sector number (for CBC-ESSIV) is unique for each encryption with the same key.	I1 or I3

Analysis: Indeed, like a nonce, the tweak value must be unique and the sector number is encrypted to produce an IV that must be unpredictable. This is similar to **[SymmetricEncryptionConfOnly-ImplementationPitfall-1]**.

The tweak value is usually derived from the address where the ciphertext is stored, and the sector number is usually derived from the address where the sector is stored.

Extra care shall be applied for storage devices or disk drivers which determine the used tweak or the sector number from logical address rather than physical address.

Sensitive data within Symmetric Disk Encryption implementations

Symmetric Encryption relies on a block cipher where the key and the plaintext are both sensitive. Hence, the tester can refer to §3.1.1 for sensitive data related to block ciphers. Additionally tweak values on the encryption side are sensitive in an integrity sense.

XTS

In addition to the evaluation tasks mentioned in §3.3.2, this section provides additional procedures regarding XTS.

XTS supports different key lengths (256 and 512 bits) and data unit lengths (data complete block sizes, partial block sizes, the largest block size supported by the implementation), various formats for the tweak value (random 128-bit hexadecimal string or Data Unit Sequence Number (base-10 number between 0 and 255)). As a consequence, the supported variants of the algorithm shall be considered as independent algorithms, and shall all be tested.

For the task **[DiskEncryption-ConformanceTesting-3]**, the appendix of this document lists KATs that shall be used for this mechanism.

Sensitive data within XTS implementations

In addition to the sensitive data specified for Symmetric Disk Encryption, the encrypted tweak values are sensitive data in the confidentiality sense.

CBC-ESSIV

In addition to the evaluation tasks mentioned in §3.3.2, this section provides additional procedures regarding CBC-ESSIV.

For **[DiskEncryption-AgreedMechanism-2]** and **[DiskEncryption-ConformanceTesting-1]**, the underlying primitives of CBC-ESSIV are:

- a block cipher;
- a hash function.

[CBC-ESSIV-AgreedMechanism-2] The tester shall determine that the malleability of the CBC mode does not bring any flaw in the system.	I1 or I3

Analysis: Due to the malleability of the CBC mode, attacks as are possible. While XTS provides no integrity and thus offers no inherent immunity against similar security issues, it is less malleable than CBC-ESSIV.

Sensitive data within CBC-ESSIV implementations

This paragraph specifies sensitive data for CBC-ESSIV in addition to the ones specified for Symmetric Disk Encryption.

In CBC-ESSIV, the IV of a sector number is generated as $E_s(SN)$ where:

- E_s is the encryption using a block cipher and the key s;
- s is the digest of the encryption key;
- SN is the sector number.

Therefore, during the computation of s, the input and the output of the hash function are sensitive and shall not be accessible by an attacker. Hence, the tester can refer to §3.1.3 for sensitive data related to hash function where the input and output are sensitive.

TH-comments until this position (03.05.2019)

3.3.3 Message Authentication Code

Message Authentication Code (MAC) can be based on a block cipher or a hash function.

[MAC-AgreedMechanism-1] The tester shall verify that the Message Authentication Code scheme is agreed .	I1

The following MACs are agreed in : CMAC, CBC-MAC, HMAC and GMAC.

[MAC-AgreedMechanism-2] The tester shall verify that the underlying primitives of the MAC are agreed.	I1

Exception: Even though SHA-1 is not an agreed primitive since is not considered as an acceptable general purpose hash function, HMAC-SHA-1 is considered for the time being as an acceptable legacy mechanism .

[MAC-AgreedMechanism-3] If the MAC of the agreed mechanism is truncated, the tester shall verify that the truncated MAC is larger than 96 bits.	I1 or I3

Analysis: It is a common practice to truncate the result of a MAC scheme. The truncated MAC shall be larger than 96 bits for the scheme to be agreed.

Exception 1: This note does not apply for GMAC where no truncation is allowed.

Exception 2: If the number of MAC verifications performed for a given key can be bounded by 2^{30}, the limit of 96 bits is lowered to 64 bits. Note that this is a legacy bound and a MAC larger than 96 bits is still recommended.

[MAC-ConformanceTesting-1] The tester shall perform the conformance tests of the underlying primitives.	I1 I2

[MAC-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.3.3 related to MAC.	I2
[MAC-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.3.3 for implementation representation analysis, as §5.1.3.3 may contain some specific cases.	I3

Sensitive data within MACs implementations

The sensitive data is the key. Depending on the usage of the MAC, the input or the output can be sensitive as well.

If relevant, the amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

CMAC

In addition to the evaluation tasks mentioned in §3.3.3, this section provides additional procedures regarding CMAC.

For **[MAC-AgreedMechanism-2]** and **[MAC-ConformanceTesting-1]**, the underlying primitive of CMAC is a block cipher.

[CMAC-ImplementationPitfall-1] The tester shall verify that the number of blocks processed by the block cipher with the same key is limited to $2^{\frac{n}{2}-5}$ (with n being the block size in bits).	I1 or I3

Analysis: The number of blocks processed by the block cipher is the number of blocks being authenticated. This results from **[3DES-ImplementationPitfall-1]**. Indeed, the

input of each encryption block is unpredictable. Hence, this is to avoid any collision. It is recalled that it is not an issue for AES: this limit is very high.

Sensitive data within CMAC implementations

This paragraph specifies sensitive data for CMAC in addition to the ones specified for MACs.

CMAC relies on a block cipher where the key K is sensitive. A sensitive key K_0 is derived from the initial key by encrypting a constant message with K. Hence, the tester can refer to §3.1.1 for sensitive data related to block ciphers where the key and the output are sensitive.

Then, two keys K_1, K_2 are derived from K_0 using shift and XOR operations. K_1 and K_2 play a part in the MAC generation process using again XOR operations with intermediate values. Hence, those keys are obviously sensitive. The input and output of shift operations are sensitive. The inputs and output of the XOR operations are sensitive as well.

CBC-MAC

In addition to the evaluation tasks mentioned in §3.3.3, this section provides additional procedures regarding CBC-MAC.

For **[MAC-AgreedMechanism-2]** and **[MAC-ConformanceTesting-1]**, the underlying primitive of CBC-MAC is a block cipher.

[CBCMAC-ImplementationPitfall-1] The tester shall verify that the size of all the inputs for which CBC-MAC is computed under the same key is the same.	I1 or I3

Analysis: Otherwise, the attacker can trivially forge MAC with two know pairs (message, MAC).

[CBCMAC-ImplementationPitfall-2] If the underlying block primitive is the Triple-DES, the tester shall verify that the number of blocks processed by the block cipher with the same key is limited to 2^{27} for Triple-DES with three keys and 2^{20} for Triple-DES with two keys.	I1 or I3

Analysis: The number of blocks processed by the block cipher is the number of blocks being authenticated. This results from **[3DES-ImplementationPitfall-1]**. Indeed, the input of each encryption block is unpredictable. Hence, this is to avoid any collision.

Sensitive data within CBC-MAC implementations

This paragraph specifies sensitive data for CBC-MAC in addition to the ones specified for MACs.

CBC-MAC relies on a block cipher where the key is sensitive. Depending on the usage of the scheme, the input and output blocks can be sensitive as well. Hence, the tester can refer to §3.1.1 for sensitive data related to block ciphers.

HMAC

For **[MAC-AgreedMechanism-2]** and **[MAC-ConformanceTesting-1]**, the underlying primitive of HMAC is a hash function.

Sensitive data within HMAC implementations

This paragraph specifies sensitive data for HMAC in addition to the ones specified for MACs.

HMAC relies on a hash function. The computation of HMAC of a message m given a key K and a hash function H is as follows:

$$HMAC(K,m) = H\left((K' \oplus opad) \vee H\big((K' \oplus ipad) \vee m\big)\right)$$

with $K' = H(K)$ if K is larger than the hash function block size and $K' = K$ otherwise.

From this construction, obviously, the sensitive data are:

- K;
- K' (and hence the output of the hash function if K is larger than the hash function block size);
- $K' \oplus opad$ and $K' \oplus ipad$;
- $\underset{H}{(K' \oplus ipad)}$.

Hence, the tester can refer to §3.1.3 for sensitive data related to hash functions where the input and the output are sensitive. The current state of the hash function is also particularly sensitive. Indeed, since hash functions process the input blocks, the knowledge of the state of the hash function after processing $K' \oplus opad$ and, in another hash function instance, $K' \oplus ipad$, is enough to produce MACs of arbitrary plaintexts [BBD+.

GMAC

In addition to the evaluation tasks mentioned in §3.3.3, this section provides additional procedures regarding GMAC.

For **[MAC-AgreedMechanism-2]** and **[MAC-ConformanceTesting-1]**, the underlying primitives of GMAC are:

- a block cipher;
- the multiplication in $GF(2^{128})$.

[GMAC-AgreedMechanism-1] The tester shall verify that the IV is not reused to protect different pairs (plaintext, associated data) under the same key.	I1 or I3

Analysis: reminds that the IV must be managed within the security perimeter of the authenticated encryption process. For example, it is crucial to ensure that no adversary can cause the same IV to be reused to protect different (plaintext, associated data) pairs under the same key.

[GMAC-AgreedMechanism-2] The tester shall verify that the IV is 96 bits long and built according to the deterministic construction specified in Section 8.2.1 of .	I1 or I3

[GMAC-AgreedMechanism-3] The tester shall verify that the MAC length is 128 bits long and is not truncated.	I1 or I3

Sensitive data within GMAC implementations

This paragraph specifies sensitive data for GMAC in addition to the ones specified for MACs.

GMAC relies on a block cipher and the multiplication in $GF(2^{128})$. The MAC generation relies on a secret H derived from the authentication key K computed as $E_K(0^{128})$ where:

- E_K is the encryption using a block cipher and the key K;
- 0^{128} is the message consisting of 128 zero bits.

The value H is a sensitive data as its knowledge permits to authenticate arbitrary plaintexts[6]. This value is used as an operand to the multiplication in $GF(2^{128})$.

Hence, because of the sensitive data H, the tester can refer to §3.1.1 for sensitive data related to block ciphers where the key and the output is sensitive, and to §3.2 where both operands are sensitive.

3.3.4 Symmetric Entity Authentication

These schemes aim to authenticate an entity, and may either be derived using a MAC scheme or an encryption scheme in a random challenge-response protocol.

[EntityAuthentication-AgreedMechanism-1] The tester shall verify that the MAC or block cipher used in the symmetric entity authentication scheme is agreed .	I1 or I3

[EntityAuthentication-AgreedMechanism-2] Authentication generally consists in a challenge-response protocol. Therefore, the tester shall verify that the size of the challenge is greater than 96 bits .	I1 or I3

Analysis: This limit is for legacy agreed mechanisms. recommends a challenge of size greater than 125 bits.

[EntityAuthentication-ConformanceTesting-1] The tester shall perform the conformance testing of the underlying mechanisms.	I1 I2

3.3.5 Symmetric Authenticated Encryption

Authenticated Encryptions (AE) provide confidentiality, integrity and data origin authentication of plaintexts. The decryption operation of an AE provides the plaintext if the ciphertext is valid, i.e. the integrity and authenticity has been checked. Additionally, in GCM, a input called *additional data* is an optional input. This additional data is considered when computing the MAC and therefore is protected in integrity and authenticity.

In all AE mechanisms considered in this section this security service is offered either:

- by the combination of two mechanisms: a symmetric encryption and a MAC mechanism, or
- by design with a specific mode of operation and one underlying primitive: a block cipher.

[AuthenticatedEncryption-AgreedMechanism-1] The tester shall verify that the Authenticated Encryption scheme is agreed .	I1

The following Authenticated Encryption schemes are agreed in : Encrypt-then-MAC, CCM, GCM, EAX, MAC-then-Encrypt and Encrypt-and-MAC.

[AuthenticatedEncryption-AgreedMechanism-2] The tester shall verify that the underlying primitives of the Authenticated Encryption scheme are agreed .	I1

[AuthenticatedEncryption-ConformanceTesting-1] The tester should test all the implemented directions of encryption operation.	I1 I2

[6] The procedure is actually not that straightforward. However, for all intended purposes, in this document, the data H is considered sensitive for it can eventually lead to forgeries.

[AuthenticatedEncryption-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.3.4 related to AE. The tester shall test correct and incorrect MAC values.	I2
[AuthenticatedEncryption-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.3.4 for implementation representation analysis, as §5.1.3.4 may contain some specific cases.	I3

[AuthenticatedEncryption-ImplementationPitfall-1] If the integrity of the ciphertext is not properly checked before decryption, the tester shall verify that the attacker cannot access to the specific error condition (format error or MAC error) that can occur in the system.	I1 or I3

Analysis: This applies to schemes with paddings in order to avoid so called *padding oracle attacks* . However, the task applies to all schemes if any verification is performed on the format of the deciphered ciphertext and if the deciphered ciphertexts are sent to consuming applications. Such verifications may lead to *format oracle attack*.

Sensitive data within Symmetric Authenticated Encryption implementations

As described in the task **[AuthenticatedEncryption-ImplementationPitfall-1]**, information on the format of the deciphered ciphertexts of arbitrary ciphertexts can be used to recover the plaintext of any ciphertext. Such information shall not be accessible by an attacker, neither by direct error code, as stated in the task, nor by Side-Channel attacks, particularly timing attacks. If the MAC verification is not performed before decryption, one commonly used countermeasure is to perform a *dummy* MAC verification when the decryption has failed.

3.3.5.1 AE – Combination of Symmetric Encryption (Confidentiality Only) and MAC

Encrypt-then-MAC, MAC-then-Encrypt and Encrypt-and-MAC are AEs that combine a Symmetric Encryption (Confidentiality Only) and a MAC.

[AuthenticatedEncryptionCombination-AgreedMechanism-1] The tester shall check the combination of the encryption and MAC is agreed .	I1

Analysis: Encrypt-then-MAC scheme is a recommended AE in . Encrypt-and-MAC and MAC-then-Encrypt are acceptable legacy schemes in .

[AuthenticatedEncryptionCombination-AgreedMechanism-2] The tester shall check the keys used for encryption and MAC operations are different.	I1 or I3

[AuthenticatedEncryptionCombination-ConformanceTesting-2] The tester shall perform the conformance tests of the underlying mechanisms (encryption and MAC).	I1 I2

[AuthenticatedEncryptionCombination-ImplementationPitfall-1] All implementation pitfalls of the underlying mechanisms shall be considered.	I1 or I3

Analysis: Pitfalls of §3.3.1 and §3.3.3 shall be considered.

Sensitive data within AE – Combination of Symmetric Encryption (Confidentiality Only) and MAC implementations

Additional sensitive data are specified below for each individual AE.

Encrypt-then-MAC

There is no additional task specific to Encrypt-then-MAC.

Sensitive data within Encrypt-then-MAC implementations

This paragraph specifies sensitive data for Encrypt-then-MAC in addition to the ones specified for AEs.

This scheme relies on a symmetric encryption scheme and a MAC. Hence, the tester can refer to §3.3.1 for sensitive data related to the underlying symmetric encryption and to §3.3.3 for sensitive data related to the underlying MAC where only the key is sensitive (the input and the output of the MAC are not sensitive).

Encrypt-and-MAC

In addition to the evaluation tasks mentioned in §3.3.5 and §3.3.5.1, this section provides additional procedures regarding Encrypt-and-MAC.

For the task **[AuthenticatedEncryption-ConformanceTesting-2]**, the appendix of this document lists KATs that shall be used for this mechanism.

[AMOSSYS: to be discussed. Should the following task appear in ACM?]

[EncryptAndMAC-AgreedMechanism-1] The tester shall verify that the confidentiality of the plaintexts is not compromised because of the deterministic MAC value.	I1

Analysis: If plaintexts are non-random messages (such as small human readable texts) or very small random messages (less than 112 bits long), an attacker can mount dictionary attacks with known pairs (plaintext, MAC value).

MACs are generally deterministic schemes. Only GMAC can be made probabilistic if a random nonce is generated for each MAC generation.

Sensitive data within Encrypt-and-MAC implementations

This paragraph specifies sensitive data for Encrypt-and-MAC in addition to the ones specified for AEs.

This scheme relies on a symmetric encryption scheme and a MAC. Hence, the tester can refer to §3.3.1 for sensitive data related to the underlying symmetric encryption and to §3.3.3 for sensitive data related to the underlying MAC where the key and the input are sensitive.

Note that if **[EncryptAndMAC-AgreedMechanism-1]** is correctly applied, the MAC is not sensitive.

MAC-then-Encrypt

There is no additional task specific to MAC-then-Encrypt.

Sensitive data within MAC-then-Encrypt implementations

This paragraph specifies sensitive data for MAC-then-Encrypt in addition to the ones specified for AEs.

This scheme relies on a symmetric encryption scheme and a MAC. Hence, the tester can refer to §3.3.1 for sensitive data related to the underlying symmetric encryption and to §3.3.3 for sensitive data related to the underlying MAC where the key and the input are sensitive. The output of the MAC may be sensitive as well since its knowledge can provide

information on the plaintext. Indeed, the computation of the MAC is deterministic (except for GMAC) and hence dictionary attacks can be mounted if the attacker knows the output of the MAC, as described in task **[EncryptAndMAC-AgreedMechanism-1]** above.

3.3.5.2 AE – Encryption designed with Authentication

CCM, GCM and EAX are encryption schemes specifically designed to provide confidentiality, integrity and authentication. They rely on a block cipher primitive.

[AuthenticatedEncryptionMode-AgreedMechanism-1] The tester shall verify that the underlying primitives are agreed .	I1

[AuthenticatedEncryptionMode-ConformanceTesting-1] The tester shall perform the conformance tests of the underlying primitives.	I1 I2

Sensitive data within AE – Encryption designed with Authentication implementations

Additional sensitive data are specified below for each individual AE.

CCM

In addition to the evaluation tasks mentioned in §3.3.5, this section provides additional procedures regarding CCM.

For **[AuthenticatedEncryptionMode-AgreedMechanism-1]** and **[AuthenticatedEncryptionMode-ConformanceTesting-1]**, the underlying primitive of CCM is a block cipher.

For **[AuthenticatedEncryption-ImplementationPitfall-1]**, it must be not possible for attacker to distinguish whether the error message results from Step 7 of the decryption operation specified in §6.2 of NIST (validity check of nonce *N*, associated data *A* and payload *P*) or from Step 10 (invalid value *T*).

[CCM-ImplementationPitfall-1] All implementation pitfalls of the CTR mode and CBC-MAC shall be considered.	I1 or I3

Analysis: CCM relies on the CTR mode (§3.3.1) for encryption and CBC-MAC mechanism (§3.3.3) for authentication.

Sensitive data within CCM implementations

This paragraph specifies sensitive data for CCM in addition to the ones specified for AEs.

The CCM mode is merely the combination of the CTR mode for encryption and CBC-MAC for authentication. The reader shall refer to the corresponding sections for sensitive data within this mechanism implementation.

GCM

In addition to the evaluation tasks mentioned in §3.3.5, this section provides additional procedures regarding GCM.

For **[AuthenticatedEncryptionMode-AgreedMechanism-1]** and **[AuthenticatedEncryptionMode-ConformanceTesting-1]**, the underlying primitives of GCM are:

- a block cipher;
- the multiplication in $GF(2^{128})$.

[GCM-AgreedMechanism-1] The tester shall verify that the IV is 96 bits long and built according to the deterministic construction specified in Section 8.2.1 of .	I1 or I3

Analysis: As the CTR mode, each counters used across all messages encrypted with the same key must be unique. Using two identical counters with the same key voids the confidentiality guarantee of the mode.

[GCM-AgreedMechanism-2] The tester shall verify that the MAC length in GCM schemes is 128 bits long and cannot be truncated.	I1 or I3

[GCM-ImplementationPitfall-1] The tester shall verify that no message of length strictly greater than $2^{32} - 2$ blocks can be encrypted.	I1 or I3

Analysis: The counters are generated with the concatenation of a unique IV of 96 bits and an incremented counter denoted on 32 bits. This task avoids the overflow of the counter.

[GCM-ImplementationPitfall-2] The tester shall verify that an IV can never be reused in encryption under the same key with differing inputs.	I1 or I3

Analysis: GCM relies on the CTR mode. Then, **[SymmetricEncryptionConfOnly-ImplementationPitfall-1]** shall be considered.

Sensitive data within GCM implementations

This paragraph specifies sensitive data for GCM in addition to the ones specified for AE.

GCM relies on a block cipher and the multiplication in $GF(2^{128})$.

The ciphertext processes the plaintext in a stream mode. Consequently, each output of the block cipher is a sensitive data as its knowledge permits to trivially derive the plaintext. Thus, this shall be considered when referring to §3.1.1.

The MAC generation relies on a secret H derived from the authentication key K computed as $E_K(0^{128})$ where:

- E_K is the encryption using a block cipher and the key K;
- 0^{128} is the message consisting of 128 zero bits.

The value H is a sensitive data as its knowledge permits to authenticate any arbitrary ciphertext and additional data[7]. This value is used as an operand to the multiplication in $GF(2^{128})$.

Hence, because of the sensitive data H, the tester can refer to §3.1.1 for sensitive data related to block ciphers where the key and the output are sensitive, and to §3.2 where one (H or both (H and the additional data if present) operands are sensitive.

Also, as all stream modes, since a key stream is generated that is XORed to the plaintext during encryption, the inputs of the XOR operations are sensitive. During decryption, one input (the key stream) and the output of the XOR operations are sensitive.

EAX

In addition to the evaluation tasks mentioned in §3.3.5, this section provides additional procedures regarding EAX.

[7] The procedure is actually not that straightforward. However, for all intended purposes, in this document, the data $H$is considered sensitive for it can eventually lead to forgeries..

[EAX-ImplementationPitfall-1] All implementation pitfalls of the CTR mode and CMAC shall be considered.	I1 or I3

Analysis: EAX relies on the CTR mode (§3.3.1) for encryption and OMAC mechanism (which is very similar or equivalent to CMAC: §3.3.3) for authentication.

Sensitive data within EAX implementations

This paragraph specifies sensitive data for EAX in addition to the ones specified for AE.

The EAX mode is merely the combination of the CTR mode for encryption and CMAC for authentication. The reader shall refer to the corresponding sections for sensitive data within this mechanism implementation.

3.3.6 Key Protection

Key protection schemes aim to securely store or transmit cryptographic key.

[KeyProtection-AgreedMechanism-1] The tester shall verify that the used Key Protection scheme is agreed .	I1

The following Key Protection schemes are agreed in : SIV and AES-Keywrap.

[KeyProtection-ConformanceTesting-1] The tester shall perform KATs or the methodology of §5.1.3.5 related to Key Protection.	I2
[KeyProtection-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.3.5 for implementation representation analysis, as §5.1.3.5 may contain some specific cases.	I3

Sensitive data within Key protection scheme implementations

In Key Protection schemes, the protected key and the key to protect it are obviously sensitive. Hence, the reader can refer to §3.1.1 for sensitive data related to block ciphers.

SIV

In addition to the evaluation tasks mentioned in §3.3.6, this section provides additional procedures regarding SIV.

[SIV-AgreedMechanism-1] When the SIV scheme uses associated data, the tester shall verify that at most $n-2$ associated data components are used as inputs of the SIV scheme , where n denotes the block size of the underlying block cipher.	I1 or I3

Analysis: As AES is the only underlying function in SIV specification limits the number of associated data to 126.

[SIV-ImplementationPitfall-1] All implementation pitfalls of the CTR mode and CMAC shall be considered.	I1 or I3

Analysis: SIV relies on the CTR mode (§3.3.1) for encryption and CMAC mechanism (§3.3.3) for authentication.

Sensitive data within SIV implementations

As SIV relies on the CMAC and CTR mechanisms using the AES, the reader can refer to §3.3.3 §3.3.1 for sensitive data related to MACs and symmetric encryptions respectively where the key and both the input and output are sensitive.

AES-Keywrap

In addition to the evaluation tasks mentioned in §3.3.6, this section provides additional procedures regarding AES-Keywrap.

[AESKeywrap-AgreedMechanism-1] The tester shall verify that plaintexts are not larger than $2^{54}-1$ semiblocks for Key Wrap without padding and $2^{32}-1$ bytes for Key Wrap with padding .	I1 or I3

Analysis: This is to avoid any collision.

Sensitive data within AES-KW implementations

This paragraph specifies sensitive data for AES-KW in addition to the ones specified for Key Protections.

This scheme relies on AES. Hence, the reader can refer to §3.1.1 for sensitive data related to block ciphers where the key and the output are sensitive.

3.3.7 Key Derivation Functions

Key Derivation schemes produce a derived key from a secret and other parameters (salt value, iteration count).

Key Derivation schemes rely on pseudo-random functions built on hash functions or MAC schemes.

[KeyDerivation-AgreedMechanism-1] The tester shall verify that the used Key Derivation scheme is agreed .	I1

The following Key Derivation schemes are agreed in : NIST SP800-56 ABC, ANSI-X9.63-KDF and PBKDF2.

[KeyDerivation-AgreedMechanism-2] If a MAC algorithm is used in key derivation scheme, the tester shall verify that an agreed MAC algorithm has been selected and that it is used in accordance with **[MAC-AgreedMechanism-1]**.	I1

[KeyDerivation-AgreedMechanism-3] If a hash function is used in key derivation scheme, the tester shall verify that an agreed hash function has been selected and that it is used in accordance with **[HashFunctions-AgreedMechanism-1]**.	I1

[KeyDerivation-ConformanceTesting-1] The tester shall perform KATs or the methodology of §5.1.3.6 related to Key Derivation.	I2
[KeyDerivation-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.3.6 for implementation representation analysis, as §5.1.3.6 may contain some specific cases.	I3

[KeyDerivation-ImplementationPitfall-1] The tester shall check that invalid requests for the keying data generation are not possible.	I1 or I3

Analysis : The computation of the derived key starts with some size controls and that shall not be ignored. In particular, the tester shall verify that no derived key is larger than

$h \times (2^{32} - 1)$ where h is the length (in bits) of the output block of the underlying hash function or pseudo-random function.

Sensitive data within Key Derivation implementations

The sensitive data are the input secret and the generated key.

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

NIST SP800-56 ABC

There is no additional task specific to NIST SP800-56 ABC.

Sensitive data within NIST SP800-56 ABC implementations

This paragraph specifies sensitive data for NIST SP800-56 ABC in addition to the ones specified for Key Derivation Functions.

NIST SP800-56 ABC relies on either a hash function or a HMAC. In the latter case, the HMAC key is a salt that does not need to be kept secret.

If the mechanism relies on a hash function, the input and the plaintext are both sensitive. Hence, the tester can refer to §3.1.3 for sensitive data related to hash functions.

If the mechanism relies on a HMAC, the input and the output are both sensitive. Hence, the tester can refer to §3.1.3 for sensitive data related to MACs. This is special use case of the HMAC where the HMAC key is not sensitive.

ANSI-X9.63-KDF

There is no additional task specific to ANSI-X9.63-KDF.

Sensitive data within ANSI-X9.63-KDF implementations

This paragraph specifies sensitive data for ANSI-X9.63-KDF in addition to the ones specified for Key Derivation Functions.

The mechanism relies on a hash function where the input and the output are both sensitive. Hence, the tester can refer to §3.1.3 for sensitive data related to hash functions.

PBKDF2

In addition to the evaluation tasks mentioned in §3.3.7, this section provides additional procedures regarding PBKDF2.

[PBKDF2-AgreedMechanism-1] The tester shall verify that the underlying function in the password protection scheme is agreed .	I1

Analysis: PBKDF2 applies a pseudo-random function to derive keys from passwords, such as HMAC-SHA1 or HMAC-SHA2 .

[PBKDF2KeyDerivation-AgreedMechanism-1] If PBKDF2 uses HMAC as the MAC function, the tester shall verify that the HMAC key length of the input key is smaller than the digest length of the underlying hash function.	I1

Analysis: If the HMAC key is longer than the hash function block size, the key is hashed. Hence, this can lower the effective entropy of the key derived.

Sensitive data within PBKDF2 implementations

This paragraph specifies sensitive data for PBKDF2 in addition to the ones specified for Key Derivation Functions.

The mechanism relies on a HMAC where the key, the input and the output are sensitive. Hence, the tester can refer to §3.1.3 for sensitive data related to HMACs.

3.3.8 Password Protection/Password Hashing Mechanisms

Password Protection schemes produce a hashed password from a clear password and other parameters (salt value for example), in order to avoid storing authentication passwords in clear form. The strength of a password is related to its length and its randomness properties (characters from several sets: digits, upper and lower cases, special characters, etc). reminds that passwords shorter than 10 characters or passwords composed of personal information (name, phone number, date of birth…) are considered as weak passwords.

Password Protection schemes rely on pseudo-random functions built on hash function or MAC scheme.

[PasswordProtection-AgreedMechanism-1] The tester shall verify that the used password protection scheme is agreed .	I1

Analysis: PBKDF2 is an agreed Password Protection/Password Hashing scheme .

[PasswordProtection-AgreedMechanism-2] The tester shall verify that the salt contains a random value of at least 128 bits. This random value shall be generated using an agreed random bit generator.	I1 or I3

Analysis: This size of 128 bits is specified in .

Sensitive data within Password Protection implementations

The sensitive data are the input password.

Password hashing mechanisms can be used for password verification without storing the password. In this case, the derived password is not a sensitive value. In fact, this is the purpose of the derivation mechanism: the password cannot be recovered even if the derived password are compromised.

However, such a derivation can be used to derive a key from a password. Of course, in this case, the derived password is sensitive.

PBKDF2

The tester shall refer to the tasks of §3.3.7: **[KeyDerivation-…]** and **[PBKDF2-…]** to evaluate the mechanism.

In addition, this section provides additional procedures regarding PBKDF2 used as password protection.

[PBKDF2PasswordProtection-AgreedMechanism-1] The tester shall verify that the number of iterations of PBKDF2 scheme is large enough to avoid a brute force attack.	I1 or I3

Analysis: , published in September 2000, recommends a minimum of 1000 iterations of PBKDF2. This limit appears to be insufficient compared to the current processors. NIST indicates in that an iteration count of 10 000 000 may be more appropriate. However, such iterations number seems to be too strong for smartcard implementations. The exact

number should be commensurate with the acceptable time for key derivation and depends also on the use case (i.e. session keys or storage keys).

Sensitive data within PBKDF2 implementations

This paragraph specifies sensitive data for PBKDF2 in addition to the ones specified for Password Protection.

The sensitive data are the same as PBKDF2 for Key Derivation Functions.

3.4 Asymmetric Atomic Primitives

The agreed asymmetric atomic primitives are RSA/Integer factorization, Discrete Logarithm in Finite Fields and Discrete Logarithm in Elliptic Curves. The main operations for those primitives are a modular exponentiation with RSA parameters, a modular exponentiation, and an Elliptic Curve scalar multiplication, respectively.

RSA/Integer factorisation

The security of the RSA primitive depends on the quality and secrecy of the prime numbers p and q whose multiplication forms the RSA modulus n. The public exponent is denoted e and the private exponent d.

[RSA-AgreedMechanism-1] The tester shall verify that all RSA public exponent in the system are greater than 2^{16} .	I1 I2

[RSA-AgreedMechanism-2] The tester shall verify that bit length of all RSA modulus used in the system are at least 1900 bits.	I1 I2

Analysis: This is a legacy limit and holds until 2024. The recommended limit number is 3000 bits.

The main computation involved in RSA-based mechanisms is modular exponentiation with RSA parameters. Specifically, the following cases may arise:

- A modular exponentiation with an RSA private key. This primitive is used for private operations in RSA based mechanisms. In this case, the following computations can be further distinguished depending on the parameters:
  - modular exponentiation with extended parameters: $(p, q, dP, dQ, qInv)$; in this case, two exponentiations are performed and a Chinese Remainder Theorem (CRT) recombination is performed at the end; this exponentiation is called **RSA in CRT mode**;
  - modular exponentiation with simple parameters: (n, d); in this case, a single exponentiation is performed.
- A modular exponentiation with an RSA public key: (n, e). This primitive is used for public operations in RSA-based mechanisms. In this case a single exponentiation is performed; in addition, the value e is generally equal to 65537 and an optimised exponentiation might be implemented.

[RSA-AgreedMechanism-3] If there are RSA keys generated in the system, the tester shall verify that the key generation process is based on a standard or referenced process.	I1 I2

Analysis: This task aims to check that no flaw in the RSA implementation allows an attacker to recover private keys. The ROCA vulnerability is an example of flaw in RSA implementation allowing attackers to recover private keys from vulnerable devices.

[RSA-ConformanceTesting-2] The tester shall test all key lengths of the primitive being used in the system.	I1 I2

Analysis: The boundaries of the key size shall be verified in the function specification. Testing every single bit-size within the boundary is not mandatory. Only key-lengths

multiple of 32 bits shall be considered. Indeed, RSA keys length are always multiple of 32 bits. However, regarding I1 and I2, it may be possible to have RSA bit-size modulus not multiple of 32 bits, every possible bit-size shall be tested.

[RSA-ConformanceTesting-2] The conformance tests shall be applied to all modular exponentiation implementations used in the system.	I1 I2

Analysis: Several implementations of modular exponentiations might occur in a system. For instance, different side-channel protections may be implemented depending if the exponentiation is used for decryption or signature.

Recommendation: Even if a modular exponentiation implementation is not used in the system under evaluation, it should be tested as long as it is in the system. Indeed, it is unsafe to keep an untested unused function in a product. If the function is activated later during a system evolution, the function would not have been tested.

[RSA-ConformanceTesting-3] The tester shall perform the MCT defined below.	I2

Analysis: For the three above exponentiations, if the implementations differ, the tester shall test all implemented exponentiations.

The generic MCT with key defined in §2.2.1 shall be considered with:

- f is a modular exponentiation: $a^b mod n$; a is considered as the input and b,n are considered as the key;
- nb_for_input=1 (**first_outputs** is void) and **encode_input** is the identity;
- nb_for_key=2 and **encode_key** takes the penultimate output and the last output where the most and least significant bit is set to one, to produce the pair (b,n);
- N_key and N_input are chosen such that the test runs for 48 hours.

In many systems, e is fixed and is equal to 65537. A special implementation of modular exponentiation may occur in such systems. Therefore, a special MCT is defined. The generic MCT with key defined in §2.2.1 shall be considered with the following characteristics:

- f is a modular exponentiation: $a^{65537} mod n$;
- the input of the function f is a number $a \in [0, n-1]$;
- n is considered as the key;
- nb_for_input=1 (**first_outputs** is void) and **encode_input** is simply equal to the output given as parameters;
- nb_for_key=1 and **encode_key** takes the output and sets the most and least significant bits to 1;
- N_key and N_input are chosen such that the test runs for 48 hours.

[RSA-ConformanceTesting-4] The tester shall perform KATs or the methodology of §5.1.4 related to the modular exponentiation and RSA recombination.	I2
[RSA-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.4 for implementation representation analysis, as §5.1.4 may contain some specific cases.	I3

[RSA-SourceCodeAnalysis-3] The tester shall verify that p and q, outputs of the RSA key pair generator, have the same bit length, and that their product has the required modulus bit length ( ) Moreover, the following condition shall be satisfied: $\lvert p-q \rvert \geq 2^{\frac{log_2(n)}{2}-100}$	I1 or I3

Analysis: For this task, the tester shall check if the condition is implemented in source code or check the randomness of p and q. In the latter case, the probability that the condition is not satisfied is negligible .

[RSA-SourceCodeAnalysis-4] The tester shall verify that d is larger than $2^{log_2(n)/2}$.	I1 or I3

Analysis: For this task, the tester shall check if the condition is implemented in source code or check that d has been chosen after that a small public exponent e has been generated.

Sensitive data within RSA primitive implementations

This paragraph specifies sensitive data for RSA primitive implementations.

As far as this document is concerned, in mechanisms that uses the RSA primitive to perform exponentiations $a^b mod n$, a, b and/or the result of the exponentiation can be sensitive. This will be specified in the sections related to mechanisms. In CRT mode, the modulus is sensitive as well.

In addition, during an exponentiation computation, the following data may also be considered sensitive (depending on whether a, b, and/or the result are sensitive):

- Almost every temporary variables within the exponentiation. These data can be used to derive the base or the result of the exponentiation. In CRT mode, the modulus can be derived as well.
- When using the CRT mode, erroneous outputs are sensitive data because of the so called *Bellcore* attack, where only one erroneous output permits to derive the private key.
- To a lesser extent, other erroneous outputs may be sensitive as well (e.g. because of Safe-Error attacks).

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

Discrete Logarithm in Finite Fields

Discrete Logarithm is a primitive used as an alternative to RSA for signature, key exchange and encryption schemes. The security of Discrete Logarithm in Finite Fields is based on the difficulty to find x given g and $y = g^x$ where g denotes the generator of a subgroup of order q of the multiplicative group $GF(p)^\times$ with p being a prime number. Let be r the largest prime factor of q.

[FFDLOG-AgreedMechanism-1] The tester shall verify that the size of the prime number p defining the finite field is greater than 1900 bits .	I1

Analysis: This is a legacy limit and holds until 2024. The recommended limit number is 3000 bits.

[FFDLOG-AgreedMechanism-2] If the used exponential modular group is widely spread, the tester shall verify that implemented DH parameters are conformance with the standard.	I1

Analysis: Discrete logarithm algorithms involve a group related precomputation phase, which is the bottleneck in terms of complexity of the attack. As a consequence, for DL modules shared by a lot of users and applications, it is strongly recommended not to use modules of length close to the lower limit of the legacy range . Examples of widely spread modular exponential groups are the ones defined in , used in both SSH and IKE protocols.

[FFDLOG-AgreedMechanism-3] The tester shall verify that r is greater than 200 bits .	I1

Analysis: In addition, it is recommended that the size of r be greater than 250 bits.

The main computations of mechanisms based on Dicrete Logarithm in Finite Field are:

- A modular exponentiation given Diffie-Hellman group parameters and an exponent; that is the computation of $y = g^x mod p$ given g, p (in the group parameters) and the exponent x;
- The computation of $g^u y^v mod p$ given g, p (defined by the group parameters), y (generally a public key) and the two exponents u, v. It is called multi modular exponentiation. This operation is generally used for signature verifications only.

[FFDLOG-ConformanceTesting-1] The conformance tests shall be applied to all modular exponential groups used in the system.	I1 I2

Analysis: It shall be verified in the system specifications which modular exponential groups can be used in the system. All used groups shall be tested. If any kind of modular exponential groups can be used within the range $[min, max]$, then a subset of standardised groups within the range size shall be tested. provides standardized modular exponential groups.

[FFDLOG-ConformanceTesting-2] The conformance tests shall be applied to all modular exponentiation implementations used in the system.	I1 I2

Analysis: Several implementations of modular exponentiations might occur in a system. For instance, different side-channel protections may be implemented depending if the exponentiation is used for decryption or signature.

Recommendation: Even if a modular exponentiation implementation is not used in the system under evaluation, it should be tested as long as it is in the system. Indeed, it is unsafe to keep an untested unused function in a product. If the function is activated later during a system evolution, the function would not have been tested.

[FFDLOG -ConformanceTesting-3] The conformance tests shall include the MCT defined below.	I2

The generic MCT with key defined in §2.2.1 shall be considered with the following characteristics:

- f is a modular exponentiation: $a^b mod p$; a is considered as the input and b is considered as the key; p is a constant modulus defined by the used modular exponential group specification;
- nb_for_input=1 (**first_outputs** is void) and **encode_input** is the identity function;
- nb_for_key=1 and **encode_key** takes the last output where the least significant bit is set to one.

[FFDLOG-ConformanceTesting-4] The tester shall perform KATs or the methodology of §5.1.4 related to modular exponentiation.	I2
[FFDLOG-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.4 for implementation representation analysis, as §5.1.4 may contain some specific cases.	I3

[FFDLOG-ImplementationPitfall-1] The tester shall check that the system only manipulate correct values y. These values must have order divisible by r and dividing q.	I1 or I3

Particularly, points received from another entity shall be verified by the system.	

Analysis: This requirement ensures that the manipulated values do not lie in subgroups of small size (i.e. ≤ 250 bits) or outside the intended group.

If q is prime, $r = q$ and the system shall verify that manipulated values have exact order q. It is advised to pick a subgroup of prime number.

Sensitive data within Discrete Logarithm in Finite Fields primitive implementations

This paragraph specifies sensitive data for Discrete Logarithm in Finite Fields primitive implementations.

Depending on the mechanisms that uses this primitive to perform modular or multi modular exponentiations, the base(s), exponent(s) and/or the result can be considered sensitive. This will be specified in the sections related to mechanisms.

In addition, during an exponentiation computation, the following data may also be considered sensitive (depending on which operands are considered sensitive):

- Almost every temporary variables within the exponentiation. These data can be used to derive the base or the result of the exponentiation as well.
- As specified in **[FFDLOG-ImplementationPitfall-1],** if modular exponentiations are performed on unintended subgroups, the outputs are sensitive.
- If an error occurs during modular exponentiations, the computation can boil down to the computation on an unintended subgroup. The erroneous computed results are then sensitive.
- To a lesser extent, other erroneous outputs may be sensitive as well (e.g. because of Safe-Error attacks).

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

Discrete Logarithm in Elliptic Curves

The elliptic curve used in ECC protocols must be robust to offer security for the asymmetric cryptographic scheme. These parameters may be common to a group of users and may be public. The security of Discrete Logarithm in Elliptic Curves is based on the difficulty to find d given G and $P = [d]G$ where G denotes the generator of a subgroup of order q of the elliptic curve. Let be r the largest prime factor of q.

[ECDLOG-AgreedMechanism-1] The tester shall verify that the elliptic curve used by the asymmetric scheme is agreed .	I1

The following elliptic curves are agreed : P256r1, P384r1, P512r1 from the Brainpool standard, P-256, P-384, P-512 from NIST, and FRP256v1.

The main computations of ECC mechanisms are:

- The computation of $Q = [d]S$, given the scalar d and a point S lying on a curve. It is called elliptic curve scalar multiplication (EC scalar multiplication) and can be used for both private and public ECC operations.
- The computation of $R = [u]S + [v]T$ given the scalars u, v and the points S, T lying on a curve. It is called multi EC scalar multiplication. This operation is generally used for signature verifications only.

[ECDLOG-ConformanceTesting-1] The conformance tests shall be applied to all elliptic curves used in the system.	I1 I2

Analysis: It shall be verified in the function specification which elliptic curves can be used in the system. All used elliptic curves shall be tested. If any elliptic curve can be used

within the range $[min, max]$, then a subset of standardised elliptic curves within the range size shall be tested. NIST and Brainpool curves are examples of standardized curves.

[ECDLOG-ConformanceTesting-2] The conformance tests shall be applied to all EC scalar multiplication and multi scalar multiplication implementations used in the system.	I1 I2

Analysis: Several implementations of elliptic curve scalar multiplication might occur in a system. For instance, different side-channel protections may be implemented depending if the exponentiation is used for decryption or signature. Also, the modular arithmetic operations are generally optimized for specific curves and therefore the implementation differ.

Recommendation: Even if an EC scalar multiplication or multi scalar multiplication implementation is not used in the system under evaluation, it should be tested as long as it is in the system. Indeed, it is unsafe to keep an untested unused function in a product. If the function is activated later during a system evolution, the function would not have been tested.

[ECDLOG-ConformanceTesting-3] The tester shall perform the MCT defined below.	I2

The generic MCT with key defined in §2.2.1 shall be considered.

To test EC scalar multiplication, f is an EC scalar multiplication with :

- f is the computation of $[d]P$; d is considered as the key and P is considered as the input;
- nb_for_input=1 (**first_outputs** is void) and **encode_input** is the identity function;
- nb_for_key=1 and **encode_key** is the following algorithm :

Require : a point on the elliptic curve

Convert the x coordinate of output as an integer and set it to $tmp0$

Convert the y coordinate of output as an integer and set it to $tmp1$

return $tmp0 + tmp1 mod q$

In some mechanisms, such as the signature generation in ECDSA, the base point is the generator G and never varies. In some optimised implementations, the EC scalar multiplication takes this generator G as the only input of the scalar multiplication. In the latter case, a specific MCT shall be performed where first_input=G and N_input=1.

To test multi EC scalar multiplication, f is an EC scalar multiplication with:

- f is the computation of $[u]S + [v]T$; the pair (u, v) is considered as the key and the pair (S, T) is considered as the input;
- nb_for_input=2 (**first_outputs** contains one point) and **encode_input** is the construction of the pair (S, T) given two outputs of f;
- nb_for_key=1 and **encode_key** is the following algorithm :

Require : a point on the elliptic curve

Convert the x coordinate of output as an integer and set it to $tmp0$

Convert the y coordinate of output as an integer and set it to $tmp1$

- **return** $(tmp0 mod q, tmp1 mod q)$

[ECDLOG-ConformanceTesting-4] The tester shall perform KATs or the methodology of §5.1.4 related to EC scalar multiplications.	I2

[ECDLOG-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.4 for implementation representation analysis, as §5.1.4 may contain some specific cases.	I3

[ECDLOG-ImplementationPitfall-1] The tester shall check that the system only manipulates points lying on the intended curve. In addition, if a curve with a non-prime order is used the points shall be in the intended subgroup. Particularly, points received from another entity shall be verified.	I1 or I3

Analysis: This task is required when points are received from another entity. In this case, the system shall verify that, before being used, the manipulated points lie on the curve and are in the intended subgroup. Otherwise, the *Invalid Curve Attack* can be possible.

If q is prime, $r = q$ and the tester shall verify that manipulated points have exact order q. It is advised to pick a subgroup of prime number.

Sensitive data within Elliptic Curve primitive implementations

This paragraph specifies sensitive data for Discrete Logarithm in Elliptic Curves primitive implementations, in the case when a secret scalar is used.

Depending on the mechanisms using this primitive to perform simple or multi EC scalar multiplication, the base point(s), scalar(s) and/or the resulting point can be considered sensitive. This will be specified in the sections related to mechanisms.

In addition, during an EC scalar multiplication computation, the following data may also be considered sensitive (depending on which operands are considered sensitive):

- Almost every temporary variables within the multiplication. These data can be used to derive the base point, the result and the scalar.
- As specified in **[ECDLOG-ImplementationPitfall-1],** if EC scalar multiplications are performed on unintended curves, the outputs are sensitive. This also applies if the elliptic curve parameters are corrupted .
- If an error occurs during EC scalar multiplications, the computation can boil down to the computation on an unintended curve. The erroneous computed results are then sensitive.
- To a lesser extent, other erroneous outputs may be sensitive as well (e.g. because of Safe-Error attacks).

The amount of information on sensitive data, accessible to an attacker through Side-Channel or Perturbation attacks, shall be restrained. Also, sensitive data shall not remain, even partially, in the volatile or non-volatile memory after usage.

3.5 Asymmetric Constructions

Asymmetric constructions rely on several primitives. For example, EC-DSA relies on:

- an EC scalar multiplication primitive;
- a hash function primitive;
- a DRG mechanism, see section 3.6.2.

[AsymmetricConstruction-ImplementationPitfall-1] All implementation pitfalls of the underlying mechanisms shall be considered.	I1 or I3

Analysis: Pitfalls related to the asymmetric primitives (§3.4), the hash functions (§3.1.3), the DRG (§3.6) and/or block ciphers (§3.1.1) shall be used.

3.5.1 Asymmetric Encryption

[AsymmetricEncryption-AgreedMechanism-1] The tester shall verify that the used Asymmetric Encryption scheme is agreed .	I1

The following Asymmetric Encryption schemes are agreed : RSA OAEP and RSA PKCS#1 v1.5 (encryption mode).

[AsymmetricEncryption-AgreedMechanism-2] The tester shall verify that the underlying primitives in the Asymmetric Encryption scheme are agreed .	I1

[AsymmetricEncryption-ConformanceTesting-1] The tester shall test the conformance of the underlying primitives.	I1 I2

[AsymmetricEncryption-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.5.1 related to Asymmetric Encryptions. The tester shall test correct and incorrect ciphertexts.	I2
[AsymmetricEncryption-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.5.1 for implementation representation analysis, as §5.1.5.1 may contain some specific cases.	I3

This conformance test or source code analysis is particularly important for RSA schemes. The padding decoding procedure, as provided in the mechanism specifications, shall be correctly implemented.

[AsymmetricEncryptionRSA-ImplementationPitfall-1] For RSA encryption schemes, the tester shall verify that the attacker cannot access to the specific error condition that can occur during the decoding of an encoded message.	I1 or I3

Analysis: If the attacker can distinguish errors, attacks such as for OAEP can be performed.

[AsymmetricEncryptionRSA-SourceCodeAnalysis-2] The tester shall verify in the source code that the random bytes used in the padding scheme are generated using an agreed DRG.	I3

Sensitive data within Asymmetric Encryption implementations

In an Asymmetric Encryption mechanism, the public operation is the encryption. The only sensitive data here is the plaintext itself, whether it is in its raw or encoded form.

The private operation is the decryption. The sensitive data in this operation are the raw or encoded plaintext, and the private key. The tester can refer to §3.4, dealing with sensitive data in RSA modular exponentiations that take place with those sensitive data.

Also, as described in task **[AsymmetricEncryptionRSA-ImplementationPitfall-1],** information regarding the success or failure of the decoding operation of deciphered ciphertexts can be used to recover the plaintext of a ciphertext. Such information shall not be accessible by an attacker either by direct error code, as stated in the task, or by Side-Channel attacks, particularly timing attacks.

RSA OAEP

For **[AsymmetricEncryption-ConformanceTesting-2]**, the KATs in Appendix contain all possible errors conditions in the decoding operations. These KATs can help for **[AsymmetricEncryptionRSA-ImplementationPitfall-1]**. In fact, those tasks correspond to the note of the provided in the mechanism specifications (Section 7.1.2 of).

Sensitive data within RSA OAEP implementations

This paragraph specifies sensitive data for RSA OAEP in addition to the ones specified for Asymmetric Encryption.

RSA OAEP makes use of a hash function. Hence, the tester may refer to §3.1.3 for sensitive data related to this kind of primitive where the input and the output are sensitive.

All the intermediate values in the OAEP encoding and decoding operation are sensitive, since they may allow the recovery of the original message.

RSA PKCS#1v1.5 - Encryption

In addition to the evaluation tasks mentioned in §3.5.1, this section provides additional procedures regarding RSA PKCS#1v1.5.

The KATs of Appendix contain all possible errors in the decoding operations and can help for **[AsymmetricEncryptionRSA-ImplementationPitfall-1].**

The *PS* value when encoding the message consists of nonzero random bytes. For the task **[AsymmetricEncryptionRSA-SourceCodeAnalysis-1]**, the tester shall in particular verify in the implementation representation that *PS* is correctly generated.

[AsymmetricEncryptionRSA-ImplementationPitfall-1] are not enough if a padding oracle is available. Indeed, attacks exist such as Bleichenbacher's attack or . In addition, the following rule shall be considered.

[RSAPKCS#1v1.5Encryption-ImplementationPittfall-1] The tester shall verify that a padding oracle is not available to an attacker, or that all attacks similar to the Bleichenbacher's attack are taken care of with efficient countermeasures.	I1 or I3

Analysis: Bleichenbacher's attack is an example of an attack targeting this encryption scheme using an oracle. In the case where countermeasures are applied, the tester shall equivalently verify that the system handles in the same way:

- an incorrectly encoded ciphertext, and,
- a correctly encoded ciphertext that has been obtained from a relation with another valid ciphertext rather than the encryption operation.

For example, the authors of suggest computing the encoding message (*EM)* in PKCS#1 v1.5 as:

$$EM = 0002 \,||\, r \,||\, 00 \,||\, \text{msg} \,||\, \text{SHA(msg)}$$

instead of

$$EM = 0002 \,||\, r \,||\, 00 \,||\, \text{msg}$$

with *r* being an octet string of random non-zero values as in PKCS#1 v1.5. The system shall handle "incorrectly encoded ciphertexts" and "correctly encoded ciphertexts where the digest does not match with the one in *EM*", in the same way. The term "in the same way" means that the system shall behave in the same manner in the error returned and the time duration of checking the format of *EM*.

Another example, adopted in TLS since version 1.0, is the following. The countermeasure is applied by the server when decrypting a message during the handshake. If the padding during decryption is not PKCS#1 v1.5 valid, the server still continues the protocol but using a random value instead of the (invalid) plaintext. This value is used as a pre-master secret to derive session keys, by both the server and the client. This prevents Bleichenbacher's attack since the attacker knows neither the plaintext value (this is in fact the value he tries to gain access to, or at least gain some information on), nor the random value generated by the server. The attacker cannot make the difference between a valid and invalid PKCS#1 v1.5 padding check since he cannot derive the pre-master secret and respond to the server in any case.

Sensitive data within RSA PKCS#1v1.5 implementations

This paragraph specifies sensitive data for RSA PKCS#1v1.5 in addition to the ones specified for Asymmetric Encryption.

As described in task **[RSAPKCS#1v1.5Encryption-ImplementationPitfall-1],** information regarding the success or failure of the decoding operation of deciphered ciphertexts can be used to recover the plaintext of a ciphertext. Such information shall not be accessible by an attacker either by direct error code, as stated in the task, or by Side-Channel attacks, particularly timing attacks.

3.5.2 Digital Signature

[DigitalSignature-AgreedMechanism-1] The tester shall verify that the Digital Signature scheme is agreed .	I1

The following Digital Signature schemes are agreed in : RSA PSS, DSA based schemes (KCDSA, Shnorr, DSA, EC-KCDSA, EC-Shnorr, EC-DSA, EC-GDSA) and RSA PKCS#1v1.5 (signature mode).

[DigitalSignature-AgreedMechanism-2] The tester shall verify that the underlying primitives in the Digital Signature scheme are agreed .	I1

[DigitalSignature-ConformanceTesting-1] The tester shall test the conformance of the underlying primitives in the Digital Signature scheme.	I1 I2

[DigitalSignature-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.5.2 related to Asymmetric Encryptions. The tester shall test correct and incorrect signatures.	I2
[DigitalSignature-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.5.2 for implementation representation analysis, as §5.1.5.2 may contain some specific cases.	I3

Sensitive data within Digital Signature implementations

In a Digital Signature mechanism, the private operation is the signature generation. Obviously, the private key is sensitive. The message and the signature are generally considered as public data and therefore not sensitive. However, it may happen that the message to be signed needs to be kept secret. In this case, the raw message is sensitive as well as its digest and its fully encoded forms (especially if messages are non-random, such as small human readable texts, or very small random messages). In this case, even the signature is sensitive as the digest can be derived from it.

The public operation is the signature verification. In this operation, nothing is generally considered sensitive. As for the signature generation, the message, its digest, its encoded form and even the signature may be sensitive.

The practical impact of these sensitive data will be made explicit in the next sections, since it depends on the specific Asymmetric Atomic Primitive underlying each Digital Signature mechanism.

3.5.2.1 RSA based schemes

Sensitive data within RSA based Digital Signature implementations

Since the same remarks can be done for RSA PSS and RSA PKCS#1v1.5, this paragraph specifies the sensitive data for both mechanisms, in addition to ones specified for Digital Signature, and in relation with the RSA Asymmetric Atomic Primitive (§3.4).

The private key is always sensitive, and it appears as exponent in RSA modular exponentiations during the signature generation. Then, the message and/or the signature may be considered sensitive. During signature generation (resp. during signature verification), the former appears, in its encoded form, as base of exponentiations (resp. as result), while the latter appears as result (resp. as base). Section §3.4 can be used to determine the intermediary values during exponentiations that are consequently considered sensitive. Also, as specified in §3.4, a faulty signature computed in the CRT mode must be considered sensitive.

Finally, RSA PSS and RSA PKCS#1v1.5 both make use of a hash function. Hence, the tester may refer to §3.1.3 for sensitive data related to this kind of primitive.

RSA PSS

In addition to the evaluation tasks mentioned in §3.5.1, this section provides additional procedures regarding RSA PSS.

For the task **[DigitalSignature-ConformanceTesting-2]**, the appendix of this document lists KATs that shall be used for this mechanism.

RSA PKCS#1v1.5 - Signature

In addition to the evaluation tasks mentioned in §3.5.1, this section provides additional procedures regarding RSA PKCS#1v1.5.

During signature verification, the conformance of the format check is very important. Bleinchenbacher and May reported the first attack on this signature scheme where the format check was not correctly implemented . KATs in Appendix contain all possible errors in the format encoding. Those KATs can be used for the task **[DigitalSignature-ConformanceTesting-2]** and therefore attest that attacks such as Bleinchenbacher's and its variants are not possible.

3.5.2.2 DSA based schemes

In addition to the evaluation tasks mentioned in §3.5.2, this section provides additional procedures regarding DSA and its derived schemes: KCDSA, Schnorr, and their elliptic variants: EC-KCDSA, EC-DSA, EC-GDSA and EC-Schnorr.

The DSA scheme is probabilistic. An exponent is randomly generated. For this conformance testing, the generation of the exponent is tweaked in such a way that the exponent is the one given in the KAT.

The leakage of the secret scalar k or even a few bits, generated during signature generation, poses risks to the confidentiality of the associated long-term private key. Strong care must be taken so that no bias can be exploited, and any bit value of k shall not leak during the use of k.

[DSA-AgreedMechanism-1] The tester shall verify that the secret value k is generated using an agreed random bit generator .	I1 or I3

[DSA-SourceCodeAnalysis-1] The tester shall verify that the secret value k is either generated uniformly in the range $[1, q-1]$ using the "Testing" technique of , Appendix B.1.2, or generated using the "extra random" technique of , Appendix B.1.1.	I3

Analysis: The naive method to generate k is to generate a random number in $[1, 2^l]$, with $l = \lceil log_2(q) \rceil$, then applying a reduction mod q. This method introduces biases and attacks

can be led. This specific task can only be done with an implementation representation analysis.

Sensitive data within DSA-based Digital Signature implementations

This paragraph specifies the sensitive data for DSA based Digital Signatures, in addition to the ones specified for Digital Signatures mechanisms. It is valid for all DSA based schemes, and none of them exhibit any additional sensitive data other than those presented here.

DSA based signatures are powered either by the Finite Fields or Elliptic Curve Discrete Logarithm primitive, both described in §3.4. One of the role of the present paragraph is to clarify which sensitive data may be appear as base (or base point), exponent (or scalar), or result of a modular exponentiation in DSA schemes; and section §3.4 can then be used to determine the intermediary values during exponentiations (or EC multiplication) that are consequently considered sensitive.

During the signature generation, there are two elements that are always sensitive: the private key, and the random per-message value k. In all DSA based signatures, the latter is involved as exponent in a modular exponentiation (or as scalar in EC scalar multiplication). Any information of such ephemeral value k, even a single bit of it for several signatures, is enough to derive the private static key [AFG+.

Also, both the privatekey and the value k are involved in the computation of the second component of DSA signature (denoted s). Thus, all intermediary results for the computation of s are sensitive.

On the other hand, during signature verification, neither the private key nor the value k are (directly) manipulated, and there is no sensitive data manipulation in this operation in the general case.

When the message and/or the signature are themselves be considered as sensitive, then all intermediary results of both the signature generation and verification are sensitive.

Finally, all DSA based schemes make use of a hash function. Hence, the tester may refer to §3.1.3 for sensitive data related to this kind of primitive.

KCDSA

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

Schnorr

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

DSA

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

EC-KCDSA

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

EC-DSA

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

EC-GDSA

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

EC-Schnorr

The tester shall refer to the evaluation tasks mentioned in §3.5.2.2.

3.5.3 Asymmetric Authentication

[AsymmetricAuthentication-AgreedMechanism-1] The tester shall verify that the used Asymmetric Authentication scheme is agreed .	I1

No dedicated asymmetric authentication scheme is currently agreed in . In practice, agreed mechanisms therefore consist in using a signature scheme in a random challenge response protocol.

The tester shall refer to evaluation tasks of the corresponding signature scheme and random generator.

[AsymmetricAuthentication-ImplementationPitfall-1] The tester shall verify that the private key is not used for different purposes (signature and authentication) .	I1

3.5.4 Key Establishment

[KeyEstablishment-AgreedMechanism-1] The tester shall verify that the Key Establishment scheme is agreed .	I1

The following Key Establishment schemes are agreed : DH, DLIES-KEM and elliptic variants: EC-DH and ECIES-KEM.

[KeyEstablishment-AgreedMechanism-2] The tester shall verify that the underlying primitives in the Key Establishment scheme is agreed .	I1

[KeyEstablishment-AgreedMechanism-3] The tester shall verify that all information is authenticated (other party, data exchanged during the key establishment).	I1

Analysis: As an example, Diffie-Hellman is an unauthenticated key establishment that may fall to man-in-the-middle attacks. In order to ensure security, it is necessary to authenticate DH parameters. This can be achieve in various ways. For instance, the DH exchange can be encrypted with a Pre-Shared Key, or signed with private keys.

[KeyEstablishment-AgreedMechanism-4] The tester shall verify that a key established through key establishment isl not used directly.	I1

***Analysis*:** The key established shall be derived into other keys using an agreed key derivation function.

[KeyEstablishment-ConformanceTesting-1] The tester shall test the conformance of the underlying primitives and mechanisms.	I1 I2

[KeyEstablishment-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.5.3 related to Key Establishment.	I2
[KeyEstablishment-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.5.3 for implementation representation analysis, as §5.1.5.3 may contain some specific cases.	I3

[KeyEstablishment-ImplementationPitfall-1] When a secret is generated from a preexisting secret, the tester shall verify that the preexisting secret has at least 100 bits of entropy.	I1

Analysis: Ephemeral keys generated using the Diffie-Hellman scheme and master secret used in TLS record protocol are examples of such preexisting secrets. In accordance with , entropy of such preexisting secret s, e.g. of each ephemeral Diffie-Hellman private key in

the former example, must be at least 100 bits for legacy mechanisms and 125 bits for recommended mechanisms.

Sensitive data within Key Establishment implementations

Agreed Key Establishment mechanisms rely on either the Finite Fields or the Elliptic Curve Discrete Logarithm primitive. The sensitive data in those primitive is described in §3.4.

In DH and EC-DH, the parties' random secret values are sensitive. They are used as exponent (resp. scalar) in a modular exponentiation (resp. EC scalar multiplication) to produce the shared secret. Of course, the result of this operation is itself sensitive, as well as the data derived from this base secret.

In DLIES-KEM and ECIES-KEM, the parties' secret keys are sensitive. They are used as exponent (resp. scalar) in the key generation operation to obtain the corresponding public key. The receiving party's secret key is also used in the decapsulation operation, as exponent (resp. scalar) to the value sent by the other party, to obtain the shared secret.

In the encapsulation operation, the sensitive data is the random secret value. It is also used as exponent (resp. scalar), applied to the other party's public key in order to obtain the shared secret. Here also, this shared secret is sensitive, as well as the data derived it. Finally, both DLIES-KEM and ECIES-KEM make use of a Key Derivation Function. The reader may refer to §3.3.7 for sensitive data related to KDF.

3.5.4.1 Schemes bases on modular exponential groups

In addition to the evaluation tasks mentioned in §3.5.4, this section provides additional procedures regarding Key Establishment scheme based on modular exponential groups.

[FFKeyEstablishment-ImplementationPitfall-1] The evaluation task **[FFDLOG-ImplementationPitfall-1]** shall be considered: the manipulated values have order divisible by r and dividing q.	I1 or I3

DH

The tester shall refer to the evaluation tasks mentioned in §3.5.4.

DLIES-KEM

The tester shall refer to the evaluation tasks mentioned in §3.5.4.

3.5.4.2 Schemes bases on elliptic curves

In addition to the evaluation tasks mentioned in §3.5.4, this section provides additional procedures regarding Key Establishment scheme based on elliptic curves.

[ECKeyEstablishment-ImplementationPitfall-1] The evaluation task **[ECDLOG-ImplementationPitfall-1]** shall be considered: the point of the other entity shall lie on the curve and in be in the intended subgroup.	I1 or I3

EC-DH

The tester shall refer to the evaluation tasks mentioned in §3.5.4 and 3.5.2.2.

ECIES-KEM

The tester shall refer to the evaluation tasks mentioned in §3.5.4 and 3.5.2.2.

3.6Random Generator

The quality of random numbers (keys, random elements of cryptographic primitives) is a crucial element for the security of a system.

3.6.1 Random Source

[RandomSource-AgreedMechanism-1] The tester shall verify that the output produced by a random source is postprocessed using a Deterministic Random Bit Generator.	I1

Analysis: A mere random source, e.g. a physical random number generator, is not considered agreed as is in .

3.6.2 Deterministic Random Bit Generator

A Deterministic Random Generator (DRG) is built around an internal state, that can be seeded and refreshed by output of a random source.

[DRG-AgreedMechanism-1] The tester shall verify that the used DRG is agreed .	I1

The agreed mechanisms are: HMAC-DRBG, Hash-DRBG and CTR-DRBG. These mechanisms are specified in and .

[DRG-AgreedMechanism-2] The tester shall verify that the underlying primitives in the DRG scheme are agreed .	I1

Analysis: The security of the DRG depends on the underlying function's behaviour when processing a series of sequential input blocks.

[DRG-AgreedMechanism-3] The tester shall verify that the DRG is backtracking resistant.	I1

Analysis: If the random number generator is backtracking resistant and is used to produce ephemeral keys of an agreed key exchange, then an attacker who has recovered the current state of the random number generator will not be able to break the *Perfect Forward Secrecy* property.

[DRG-ConformanceTesting-1] The tester shall perform conformance tests of the underlying primitives used in the DRG.	I1 I2

[DRG-ConformanceTesting-2] The tester shall perform KATs or the methodology of §5.1.6 related to DRG.	I2
[DRG-SourceCodeAnalysis-1] The tester shall verify in the source code that the implementation is compliant with the specifications. The tester shall still use §5.1.6 for implementation representation analysis, as §5.1.6 may contain some specific cases.	I3

[DRG-ImplementationPitfall-1] The tester shall identify the random sources used to seed for instantiating and reseed for updating the internal state of the DRG. The tester also shall analyse their random quality: the tester shall verify that the entropy of the seed is at least 125 bits long.	I1 I2 I3

Analysis: A very bad way to seed a DRG would be to use the current time only. The test consists to check if the entropy provided by the identified random sources is sufficient and cannot be reduced depending on the context. In particular, the seed used for reseeding shall be different than the seed used for the DRG instantiation . The DRG must be seeded with sufficient entropy to provide the required for the security strength. The min-entropy length 125 is from .

[DRG-ImplementationPitfall-2] The tester shall verify that the internal state of the DRG is periodically reseeded with random source to ensure enough entropy in case of generation of large random quantity.	I1 or I3

Analysis: In some implementations (as smartcards), reseeding may not be possible and an alternative to reseeding shall be to create an entirely new instantiation (to obtain a new seed and guarantee enough entropy in the DRG internal state).

Sensitive data within DRG implementations

The sensitive data are the seed(s) used to initialise or re-seed the DRG, and the internal state of the DRG. The random bits output by the mechanisms are also often sensitive (e.g. if used to generate a secret key). Note that the seed(s) and the DRG internal state should be protected at least as well as these generated random bits.

HMAC-DRBG

The tester shall refer to the evaluation tasks mentioned in §3.6.1 and.3.6.2.

Sensitive data within HMAC-DRBG implementations

This paragraph specifies the sensitive data for a HMAC-DRBG mechanism, in addition to ones specified for DRGs.

Since HMAC-DRBG is based on a HMAC, the reader may refer to the paragraph on HMAC in §3.3.3 for the sensitive data related to this mechanism where the key, the message and the generated MAC are sensitive.

Hash-DRBG

The tester shall refer to the evaluation tasks mentioned in §3.6.1 and.3.6.2.

Sensitive data within Hash-DRBG implementations

This paragraph specifies the sensitive data for a Hash-DRBG mechanism, in addition to ones specified for Deterministic Random Bit Generator.

Since Hash-DRBG is based on a hash function, the reader may refer to §3.1.3 for the sensitive data related to this kind of primitive, where the input and the output are sensitive.

CTR-DRBG

The tester shall refer to the evaluation tasks mentioned in §3.6.1 and.3.6.2.

In addition to these evaluation tasks, this section provides additional procedures regarding CTR-DRG.

[CTRDBRG-AgreedMechanism-1] The tester shall verify that a single invocation of the CTR-DRBG generate function is not done for the generation of several different keys, if the property of perfect forward secrecy is required in the system.	I1 or I3

Analysis: The backtracking resistance of CTR-DRBG is only given for the transition function between different invocations.

Sensitive data within CTR-DRBG implementations

This paragraph specifies the sensitive data for a CTR-DRBG mechanism, in addition to ones specified for Deterministic Random Bit Generator.

Since CTR-DRBG is based on a use of a block cipher in CTR mode, the reader may refer to paragraphs §3.1.1 on block ciphers, and to the paragraph on the CTR mode in §3.3.1 for the sensitive data related to these kinds of primitives and mechanisms, where the key, the input and the output of the block cipher primitive are sensitive.

3.6.3 Random Number Generator with Specific

Distribution

For specific usage, the DRBG as described previously cannot be used directly because random numbers need to follow specific distributions. Asymmetric cryptographic mechanisms are example for which a random number generator needs to provide some random prime integers with specific characteristics.

[KeyPairGenerator-ImplementationPitfall-1] The tester shall verify that the probability that the number generated by the random number generator is composite is lower than $1/(2^{125})$.	I1 or I3

Analysis: If the generation process does not prove the primality of the output, it shall be analysed by the tester. For example, if the Miller-Rabin algorithm is used, 4 iterations are enough to generate prime numbers of size 1536 bits with a probability error of $1 / (2^{128})$.

Sensitive data within RNG with Specific Distribution implementations

Assuming that the final random number is considered sensitive, the sensitive data are the random bits obtained from the DRG, and all the intermediary values during the computations leading to the random number.

3.7 Key Management

Note 53 of states that "*the management of the keys by the product should not enable a potential attacker to recover any information about secret and private keys used to protect user information, nor to alter or inject public keys used to protect identities*".

This principle is refined in the different life cycles of the keys:

- Key Generation;
- Key Storage and Transport;
- Key Use;
- Key Destruction.

[KeyManagement-SourceCodeAnalysis-1] The tester shall list all the keys used in the system with a clear identification of their generation, storage and transport, use, and destruction.	I1

Analysis: This task is done using the design and implementation representation. The main goal of this task is to have a clear understanding of the different keys used. This will help for the tasks of the sections below which refine the main concept of "Key Management".

3.7.1 Key Generation

Depending on the cryptosystem, the key can be generated using the following mechanisms:

- *Deterministic random bit generator*;
- *Key establishment mechanism*;
- *Key derivation function*.

[KeyGeneration-AgreedMechanism-1] Depending on the used generation process, the tester shall evaluate the key generation mechanism according to the following sections of this document: - 3.6.2 ""; - 3.5.4 ""; - 3.3.7 "".	I1 I2 I3

Analysis: In a system, many random variables might be generated for different purposes, sometimes unrelated to security. Non-agreed DRGs might be used for these variables generation. The tester shall verify that any generation of a variable using a non-agreed DRG is not a cryptographic secret, or it is not used for a cryptographic secret

generation. For example, in C language, the tester can search for calls to the weak DRG `rand` and verify that is it used for variables unrelated to any cryptographic secret.

3.7.2 Key Storage and Transport

[KeyStorage-AgreedMechanism-1] The tester shall verify that the key stored are protected in authenticity and integrity for public keys, secret keys and private keys. In addition, secret and private keys shall be protected in confidentiality.	I1 or I3

[KeyTransport-ImplementationPitFall-1] The tester shall verify that the key distribution channel is protected in authenticity and integrity for public keys and secret keys. In addition, secret keys shall be protected in confidentiality.	I1 or I3

Analysis: Some cryptographic keys must be distributed between identified users, allowing them to access to sensitive data protected by the related cryptosystems. So, protection of the key distribution channel is crucial. Also, if the system allows users to load their secret key generated outside the system, the tester shall check that the secret key has not been modified during import.

[KeyStorage-ConformanceTesting-1] The tester shall perform the conformance tests of the underlying mechanisms used for key storage and transport.	I1 I2

3.7.3 Key Usage

[KeyUsage-ImplementationPitfall-1] The tester shall verify that each key has only one usage.	I1 or I3

Analysis: A key must not be used with different mechanisms or contexts, otherwise an attacker could exploit a source of errors of usage of the key. This does not forbid to derive various keys from a master key.

[KeyManagement-ImplementationPitfall-1] The tester shall verify that the variable memory allocated for a key is locked when it is manipulated by the system.	I3

3.7.4 Key Destruction

[KeyDestruction-SourceCodeAnalysis-1] The tester shall verify that secret keys are securely erased after usage.	I3

Analysis: This task needs a source code analysis. The erasure process must be adapted to the environment and takes remanence issues into account.

4 Traceability of the evaluation tasks

In summary, the following evaluation tasks shall be considered for EC-DSA.

Procedure	Description
[DigitalSignature-AgreedMechanism-2]	Those tasks are related to the mechanism.
[DigitalSignature-ConformanceTesting-2] or **[DigitalSignature-SourceCodeAnalysis-1]**	
[DSA-AgreedMechanism-1]	
[DSA-ImplementationPitfall-1]	
[ECDLOG-AgreedMechanism-1]	Those tasks are related to the underlying elliptic curve.
[ECDLOG-ConformanceTesting-1]	
[ECDLOG-ConformanceTesting-2]	
[ECDLOG-ConformanceTesting-3]	
[ECDLOG-ConformanceTesting-4] or **[ECDLOG-SourceCodeAnalysis-1]**	
[ECDLOG-ImplementationPitfall-1]	
[HashFunctions-AgreedMechanism-1]	Those tasks are related to the underlying hash function.
[HashFunctions-ConformanceTesting-1]	
[HashFunctions-ConformanceTesting-2]	
[HashFunctions-ConformanceTesting-3] or **[HashFunctions-SourceCodeAnalysis-1]**	

[TODO for each mechanism.]

5 Appendix

5.1 KATs or similar methodology

This section provides KATs or a similar methodology for each cryptographic mechanism if required for conformance testing. All provided KATs shall be tested, or the entire cited methodology shall be followed.

5.1.1 Symmetric Atomic Primitives

AES

No KAT required to test this primitive.

TRIPLE-DES

No KAT required to test this primitive.

SHA-2

The tester shall use the KATs provided in the files named after the hash function. Following the methodology of NIST[8], for each test, a text in hexadecimal value and a *repeat* number are provided as inputs. The output consists in hashing the text repeated *repeat* time.

Only byte-oriented tests are provided. For bit-oriented tests, the tester shall use the public KATs provided in [NESSIE] or the methodology given in (except the Monte Carlo Test).

SHA-3

The methodology defined in shall be used to test the primitive (except the Monte Carlo Test).

SHAMIR'S SECRET SHARING

[KATs to find or define, including tests with different number of shares]

5.1.2 Multiplication in $GF(2^{128})$

No KAT required to test this primitive.

5.1.3 Symmetric Mechanisms

5.1.3.1 Symmetric Encryption (Confidentiality Only)

CTR

KATs are provided in files named "ctr-no_padding-correct_ciphertext-*block_primitve*.json" with:

- *block_primitive* being either "3des112", "3des168", "aes128", "aes192" or "aes256" for the corresponding block primitive.

The tests are displayed as **decryption tests**: the output is the plaintext. However, they can be used for encryption as well. In this case, the tester takes the "correct_ciphertext", has to switch the "plaintext" and "ciphertext" objects.

[8] https://csrc.nist.gov/CSRC/media/Projects/Hash-Functions/documents/SHA3-KATMCT1.pdf

The following test cases are given:

Test description	Test number
Empty message to encrypt.	001
i (*i* varying from 1 to $8 \times blocksize - 1$) bytes long message.	002 to $8 \times blocksize$

OFB

KATs are provided in files named "ofb-no_padding-correct_ciphertext-*block_primitve*.json" with:

- *block_primitive* being either "3des112", "3des168", "aes128", "aes192" or "aes256" for the corresponding block primitive.

The tests are displayed as **decryption tests**: the output is the plaintext. However, they can be used for encryption as well. In this case, the tester takes the "correct_ciphertext", has to switch the "plaintext" and "ciphertext" objects.

The following test cases are given for correct ciphertexts:

Test description	Test number
Empty message to encrypt.	001
i (*i* varying from 1 to $8 \times blocksize - 1$) bytes long message to encrypt.	002 to $8 \times blocksize$

CBC

KATs are provided in files named "cbc-no_padding-correct_ciphertext-*block_primitve*.json" with:

- *padding_scheme* being either "pkcs7", "iso7816" or "no_padding";
- *ciphertext_correctness* being either "correct_ciphertext" or "incorrect_ciphertext";
- *block_primitive* being either "3des112", "3des168", "aes128", "aes192" or "aes256" for the corresponding block primitive.

The tests are displayed as **decryption tests**: the output is the plaintext if the ciphertext is valid and "incorrect ciphertext" otherwise. However, they can be used for encryption as well. In this case, the tester takes the "correct_ciphertext", has to switch the "plaintext" and "ciphertext" objects.

For correct ciphertexts with paddings, the following test cases are given:

Test description	Test number
Empty message to encrypt. Need a full block of padding.	001
i (*i* varying from 1 to $8 \times blocksize - 1$) bytes long message to encrypt. The padding needs to be adjusted accordingly.	002 to $8 \times blocksize$

For incorrect ciphertexts with ISO-7816 padding, the following test cases are given:

Test description	Test number
The byte number *i* (*i* varying from 0 to $blocksize - 1$) (starting from the left) of the padding is incorrect.	001 to $blocksize$
The decrypted blocks are formed of zero bytes. This is to test a possible overflow when all bytes are tested against the byte 0x80 or a byte different from 0x00.	$blocksize + 1$

For incorrect ciphertexts with PKCS#7 padding, the following test cases are given:

Test description	Test number
The byte number *i* (*i* varying from 0 to $msglen - 1$) (starting from the left) for a message of $msglen$ bytes of the padding is incorrect.	001 to $blocksize * (blocksize + 1)$
Each byte of the decrypted block is larger than the block size. This is to test a possible overflow when the end of the decrypted blocks buffer is reached.	$blocksize * (blocksize + 1) + 1$
The padding length is equal to one block size plus one, as well as the padding length included in each byte of the padding.	$blocksize * (blocksize + 1) + 2$

CBC is occasionally used without paddings if plaintexts have constant sizes. It is used generally to encrypt other keys. KATs are provided with plaintexts of size multiple of the block size. The following test cases are given:

Test description	Test number
i (*i* varying from 1 to 8) blocks long message to encrypt.	001 to 008

CBC-CS

[KATs to be defined.]

CFB

KATs are provided in files named "cfb-no_padding-correct_ciphertext-*block_primitve*.json" with:

- *block_primitive* being either "3des112", "3des168", "aes128", "aes192" or "aes256" for the corresponding block primitive.

The tests are displayed as **decryption tests**: the output is the plaintext. However, they can be used for encryption as well. In this case, the tester takes the "correct_ciphertext", has to switch the "plaintext" and "ciphertext" objects.

The following test cases are given for correct ciphertexts:

Test description	Test number
Empty message to encrypt.	001
i (*i* varying from 1 to 127) bytes long message to encrypt.	002 to 128

5.1.3.2 Symmetric Disk Encryption

XTS

The tester shall use the methodology defined in .

CBC-ESSIV

[KATs to find or define.]

5.1.3.3 MAC

CMAC

The tester shall use the methodology defined in .

CBC-MAC

The tester shall use KATs in [NESSIE].

HMAC

KATs are provided in files named "hmac-*sha*.json" with:

- *sha* being the hash function used.

The following test cases are given:

Test description	Test number
The message is empty.	001
The message is exacly *hash_blocksize* (which is the block size of the inner hash functions).	002
Test representing a regular use case (the key is 16 bytes long, and the message is 21 bytes long)	003

GMAC

The tester shall use the methodology defined in with zero-length plaintext and MAC length 128 bits.

5.1.3.4 AE

Encrypt-then-MAC

[KATs to find or define.]

MAC-then-Encrypt

[KATs to find or define.]

Encrypt-and-MAC

[KATs to find or define. AMOSSYS is thinking in combining KATs provided for symmetric encryption (confidentiality only) and MAC.]

CCM

The tester shall use KATs available in and NIST .

GCM

The tester shall use KATs available in the *Wycheproof* project[9].

EAX

The tester shall use which provides 10 test vectors (indicated as not verified in the paper) in its Appendix E for EAX-AES128. Vectors were provided by Jack Lloyd and later verified by Brian Gladman .

5.1.3.5 Key Protection

SIV

The tester shall use KATs in if the underlying block cipher is AES.

9 https://github.com/google/wycheproof/blob/master/testvectors/aes_gcm_test.json

AES-Keywrap

The tester shall use test vectors available in and .

5.1.3.6 Key Derivation

NIST SP800-56 ABC

The tester shall use public KATs available in , , .

ANSI-X9.63-KDF

[KATs to find or define.]

PBKDF2

[KATs to find or define.]

5.1.4 Asymmetric Primitives

Exponentiation

[KATs to find or define.]

RSA recombination

[KATs to find or define.]

EC scalar multiplication

[KATs to find or define. In particular special cases with known bug attacks such as CVE-2017-8932 and CVE-2017-10176.]

Multi EC scalar multiplication

[KATs to find or define.]

5.1.5 Asymmetric Constructions

5.1.5.1 Asymmetric Encryption

For each RSA encryption mechanism, KATs are provided in files named "*encryption_mechanism-size-ciphertext_correctness-hash_function*.json" with:

- *encryption_mechanism* being either "rsa_oaep" or "rsa_pkcs1v1_5" for the corresponding mechanisms;
- *ciphertext_correctness* being either "correct_ciphertext" or "incorrect_ciphertext";
- *size* being either 2048, 3072 or 4096 for the modulus bit length;
- *hash_function* being the used hash function in the mechanism (for OAEP only).

The tests are displayed as **decryption tests**: the output is the plaintext if the ciphertext is valid and "incorrect ciphertext" otherwise. However, they can be used for encryption as well. In this case, the tester takes the "correct_ciphertext", has to switch the "plaintext" and "ciphertext" objects and use the public key given in the "debug_information" object.

The private key is provided in both extended and simple private key parameters, to test both exponentiation functions.

RSA OAEP

Test cases are given following the specification of the mechanism in .

The hash function in the file name provides the hash function used for both computation of $lHash$ and in MGF.

For correct ciphertexts, the following test cases are given (with *hLen* being the hash digest length):

Test description	Test number
'Normal' encryption, with a label.	001
'Normal' encryption, without a label.	002
Empty plaintext, with a label.	003
Empty plaintext, without a label.	004
The plaintext is k-2hLen-2 bytes long. PS is then empty.	005

For incorrect ciphertexts, the following test cases are given (with *hLen* being the hash digest length, *psLen* the length of PS):

Test description	Test number
The ciphertext is longer than n. The test corresponds to a failure at Step 1.b of RSASES-OAEP-Decryption Specification.	001
The ciphertext is equal to n. The test corresponds to a failure at Step 2.b of RSASES-OAEP-Decryption Specification.	002
The ciphertext is equal to a valid ciphertext (denoted c) plus n. c has been chosen such that c+n has the same byte length than c and n. The test corresponds to a failure at Step 2.b of RSASES-OAEP-Decryption Specification.	003
DB is set to DB = lHash || 00…00. This can lead to a buffer overflow when trying to reach the end of PS (the first non-zero byte). The test corresponds to a failure at Step 3.g of RSASES-OAEP-Decryption Specification.	004
Byte number *i* (*i* varying from 1 to *hLen*) of lHash is incorrect (XORed with 0x01) in DB. The test corresponds to a failure at Step 3.g of RSASES-OAEP-Decryption Specification.	005 to 005+*hLen*
The first byte of EM is 0x01 instead of 0x00. Checking this value allow to mitigate Manger's attack. The test corresponds to a failure at Step 3.g of RSASES-OAEP-Decryption Specification.	006+*hLen*
The byte number *i* (*i* varying from 1 to *hLen*-1) of PS is set to 0xFF instead of 0x00 in DB.	007+*hLen* to 007+*hLen*+*psLen*-1
The ciphertext is equal to one.	007+*hLen*+*psLen*
The ciphertext is equal to zero.	008+*hLen*+*psLen*
The ciphertext is equal to n-1.	009+*hLen*+*psLen*

RSA PKCS#1v1.5 – Encryption

Test cases are given following the specification of the mechanism in .

For correct ciphertexts, the following test cases are given:

Test description	Test number
'Normal' encryption.	001
Empty plaintext.	002

For incorrect ciphertexts, the following test cases are given:

Test description	Test number
The ciphertext is longer than n. The test corresponds to a failure at Step 1.b of RSASES-OAEP-Decryption Specification.	001
The ciphertext is equal to a valid ciphertext (denoted c) plus n. c has been chosen such that c+n has the same byte length than c and n. The test corresponds to a failure at Step 2.b of RSASES-PKCS1-v1_5-Decryption Specification.	002

Test description	Test number
The first byte of EM is 0x01 instead of 0x00. The test corresponds to a failure at Step 3 (part 1) of RSASES-PKCS1-v1_5-Decryption Specification.	003
EM is encoded using PKCS#1v1.5 signature padding instead of encryption. The test corresponds to a failure at Step 3 (part 2) of RSASES-PKCS1-v1_5-Decryption Specification.	004
The byte to separate PS from M is 0x01 instead of 0x00. This test can lead to a buffer overflow if the decryption is not correctly implemented. The test corresponds to a failure at Step 3 (part 3) of RSASES-PKCS1-v1_5-Decryption Specification.	005
PS contains 7 bytes (the minimum size is 8). M has been appended with leading zero bytes. The test corresponds to a failure at Step 3 (part 4) of RSASES-PKCS1-v1_5-Decryption Specification.	006
The ciphertext is equal to zero.	007
The ciphertext is equal to one.	008
The ciphertext is equal to n-1.	009

5.1.5.2 Digital Signature

5.1.5.2.1 RSA based schemes

For each RSA signature mechanism, KATs are provided in files named "*signature_mechanism-size-signature_correctness-hash_function*.json" with:

- *signature_mechanism* being either "rsa_oaep" or "rsa_pkcs1v1_5" for the corresponding mechanisms;
- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *size* being either 2048, 3072 or 4096 for the modulus bit length;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The private key is provided in both extended and simple private key parameters, to test both exponentiation functions.

RSA PSS

Test cases are given following the specification of the mechanism in .

The KATs are limited to salt length being equal the size of the digest output.

For correct signatures, the following test cases are given:

Test description	Test number
'Normal' signature.	001
Empty plaintext.	002

For incorrect signatures, the following test cases are given (*k* being the modulus length in bytes, *hLen* being the hash digest length):

Test description	Test number
The signature is longer than n. The test corresponds to a failure at Step 1 of RSASSA-PSS-Verify Specification.	001
The signature is equal to n. The test corresponds to a failure at Step 2.b of RSASSA-PSS-Verify Specification.	002

Test description	Test number
The signature is equal to a valid signature (denoted s) plus n. s has been chosen such that s+n has the same byte length than s and n. The test corresponds to a failure at Step 2.b of RSASSA-PSS-Verify Specification.	003
The salt length is too large. Note that the signature is valid in the sense that it has been correctly generated with the correct intermediate variables in debug_information. However, the salt length as input of the verification function is incorrect and larger than the limit. The test corresponds to a failure at Step 3 of RSASSA-PSS-Verify Specification, Step 3 of EMSA-PSS-Verify Specification.	004
End byte 0xbc of EM has been replaced by 0x00. The test corresponds to a failure at Step 3 of RSASSA-PSS-Verify Specification, Step 4 of EMSA-PSS-Verify Specification.	005
Byte number *i* (*i* varying from 1 to *k*-2*hLen*-2) of DB is different to 0. The test corresponds to a failure at Step 3 of RSASSA-PSS-Verify Specification, Step 10 of EMSA-PSS-Verify Specification (first part).	006 to 006+*k*-2*hLen*-2-1
Byte 0x01 of DB has been changed to 0x00. The test corresponds to a failure at Step 3 of RSASSA-PSS-Verify Specification, Step 10 of EMSA-PSS-Verify Specification (second part).	004+*k*-2*hLen*
Byte number *i* (*i* varying from 1 to *hLen*) of H has been changed (the correct byte has been XORed to 0x01). The test corresponds to a failure at Step 3 of RSASSA-PSS-Verify Specification, Step 14 of EMSA-PSS-Verify Specification.	005+*k*-2*hLen* to 005+*k*-*hLen*
The signature is equal to zero.	006+*k*-*hLen*
The signature is equal to one.	007+*k*-*hLen*
The signature is equal to n-1.	008+*k*-*hLen*

RSA PKCS#1v1.5 – Signature

Test cases are given following the specification of the mechanism in .

For correct signatures, the following test cases are given:

Test description	Test number
'Normal' signature.	001
Empty plaintext.	002

For incorrect signatures, the following test cases are given (*k* is the modulus length in bytes):

Test description	Test number
The signature is equal to n. The test corresponds to a failure at Step 2.b of RSASSA-PKCS1-v1_5-Verify Specification.	001
The signature is equal to a valid signature (denoted s) plus n. s has been chosen such that s+n has the same byte length than s and n. The test corresponds to a failure at Step 2.b of RSASSA-PKCS1-v1_5-Verify Specification.	002
Byte number *i* (*i* varying from 1 to *k*) of EM has been changed (the correct byte has been XORed to 0x01). The test corresponds to a failure at Step 10 of RSASSA-PKCS1-v1_5-Verify Specification.	003 to 003+*k*
The signature is equal to zero.	004+*k*
The signature is equal to one.	005+*k*
The signature is equal to n-1.	006+*k*
S is less than eight bytes long. The correct bytes have been replaced by 0x00. The test corresponds to a failure at Step 3 of EMSA-PKCS1-v1_5 Specification.	007+*k*
There is no 0x00 byte after PS. This can lead to an overflow if the implementation tries to reach a zero byte in EM.	008+*k*

Test description	Test number
There is no prefix identifying the hash function. PS has been extended consequently.	009+*k*

5.1.5.2.2DSA based schemes (in finite fields)

KCDSA

KATs are provided in files named "kcdsa-*signature_correctness-group-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *group* corresponding to the agreed group modulo a prime number;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to s. The signature is invalid because s > q.	002
The signature is invalid because s = q.	003
The signature is invalid because s = 0.	004
The signature is invalid because the intermediate value g^(z/s) * y^(r/s) = 1 (mod p) (r = -z/x mod q)	005

Schnorr

KATs are provided in files named "sdsa-*signature_correctness-group-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *group* corresponding to the agreed group modulo a prime number;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to r. The signature is invalid because r > q.	002
The signature is invalid because r = q.	003
The signature is invalid because r = 0.	004
q has been added to s. The signature is invalid because s > q.	005
The signature is invalid because s = q.	006
The signature is invalid because s = 0.	007
The signature is invalid because the intermediate value g^s * y^(-r) = 1 (mod p) (s = r * x mod q).	008

DSA

KATs are provided in files named "dsa-*signature_correctness*-*group*-*hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *group* corresponding to the agreed group modulo a prime number;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to r. The signature is invalid because r > q.	002
The signature is invalid because r = q.	003
The signature is invalid because r = 0.	004
q has been added to s. The signature is invalid because s > q.	005
The signature is invalid because s = q.	006
The signature is invalid because s = 0.	007
The signature is invalid because the intermediate value g^(z/s) * y^(r/s) = 1 (mod p) (r = -z/x mod q).	008

5.1.5.2.3DSA based schemes (EC variants)

EC-KCDSA

KATs are provided in files named "eckcdsa-*signature_correctness-curve-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *curve* corresponding to the agreed elliptic curve;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to s. The signature is invalid because s > q.	002
The signature is invalid because s = q.	003
The signature is invalid because s = 0.	004
The signature is invalid because r = 2^h, with h the underlying digest size.	005
The signature is invalid because the public point is the point at infinity.	006
The signature is invalid because the public point is not on the curve.	007
s = -w*d mod q. Therefore, the intermediate point is the point at infinity.	008

EC-DSA

KATs are provided in files named "ecdsa-*signature_correctness-curve-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *curve* corresponding to the agreed elliptic curve;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to r. The signature is invalid because r > q.	002
The signature is invalid because r = q.	003
The signature is invalid because r = 0.	004
q has been added to s. The signature is invalid because s > q.	005
The signature is invalid because s = q.	006
The signature is invalid because s = 0.	007
The signature is invalid because the public point is the point at infinity.	008
The signature is invalid because the public point is not on the curve.	009
r = -H(m)/d mod q. Therefore, the intermediate point is the point at infinity.	010

EC-GDSA

KATs are provided in files named "ecgdsa-*signature_correctness*-*curve*-*hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *curve* corresponding to the agreed elliptic curve;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to r. The signature is invalid because r > q.	002
The signature is invalid because r = q.	003
The signature is invalid because r = 0.	004
q has been added to s. The signature is invalid because s > q.	005
The signature is invalid because s = q.	006
The signature is invalid because s = 0.	007
The signature is invalid because the public point is the point at infinity.	008
The signature is invalid because the public point is not on the curve.	009

Test description	Test number
s = -H(m) * d mod q. Therefore, the intermediate point is the point at infinity.	010

EC-SDSA (Schnorr Signature Scheme)

KATs are provided in files named "ecsdsa-*signature_correctness-curve-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *curve* corresponding to the agreed elliptic curve;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to r. The signature is invalid because r > q.	002
The signature is invalid because r = q.	003
The signature is invalid because r = 0.	004
q has been added to s. The signature is invalid because s > q.	005
The signature is invalid because s = q.	006
The signature is invalid because s = 0.	007
The signature is invalid because the public point is the point at infinity.	008
The signature is invalid because the public point is not on the curve.	009
s = r * d mod q. Therefore, the intermediate point is the point at infinity.	010

EC-FSDSA (Full Schnorr Signature Scheme)

KATs are provided in files named "ecfsdsa-*signature_correctness-curve-hash_function*.json" with:

- *signature_correctness* being either "correct_signature" or "incorrect_signature";
- *curve* corresponding to the agreed elliptic curve;
- *hash_function* being the used hash function in the mechanism.

The tests are displayed as **verification tests**: the inputs are the plaintext and the signature, and the output is either "correct signature" or "incorrect signature". However, they can be used for testing signature generation as well. In this case, the tester takes the "correct_signature", the output is the signature and use the public key (and possibly the random *seed*) given in the "debug_information" object. For testing the signature function, the DRG used is tweaked: the random seed returned is the one provided in the KAT.

The correct signatures are conform with , especially the truncation method of the digest (Step 5 of Section 4.1.3).

For correct signatures, the following test cases are given:

Test description	Test number
The message to sign is empty.	001
Tests representing regular use cases.	002 to 010

For incorrect signatures, the following test cases are given:

Test description	Test number
The signature is invalid because (r, s) has been randomly drawn.	001
q has been added to s. The signature is invalid because s > q.	002
The signature is invalid because s = q.	003
The signature is invalid because s = 0.	004
The signature is invalid because r does not correspond to a point on the curve.	005
The signature is invalid because r corresponds to the point at infinity.	006
The signature is invalid because the public point is the point at infinity.	007
The signature is invalid because the public point is not on the curve.	008
s = r * d mod q. Therefore, the intermediate point is the point at infinity.	009

5.1.5.3 Key Establishment

5.1.5.3.1 Key Establishment (modular exponential groups)

DH

Diffie-Hellman Modular Exponential Groups are largely used. They are defined in . The tester shall use KATs available in the *Pyca* project[10].

DLIES-KEM

[KATs to find or define.]

5.1.5.3.2 Key Establishment (EC variants)

EC-DH

[KATs to find or define.]

ECIES-KEM

The tester shall use KATs available in *ISO/IEC 18033-2*[11].

5.1.6 Deterministic Random Bit Generator

HMAC-DRBG

The tester shall use the methodology defined in .

10 https://github.com/pyca/cryptography/blob/master/vectors/cryptography_vectors/asymmetric/DH/rfc3526.txt

11 http://shoup.net/iso/std4.pdf

Hash-DRBG

The tester shall use the methodology defined in .

CTR-DRBG

The tester shall use the methodology defined in .

5.2 MCT

[MCT samples to create when MCT will be validated for each primitive.]