Federal Office
for Information Security

A study commissioned by the Federal Office for Information Security (BSI)

Project 197

Secure Implementation of a Universal Crypto Library

Test Specification

Version 1.2.0 / 2017-03-02
Botan 2.0.1-RSCS1

Summary
The objective of this project is the secure implementation of a universal crypto library which contains all common cryptographic primitives that are necessary for the wide use of cryptographic operations. These include symmetric and asymmetric encryption and signature methods, PRFs, hash functions and RNGs. Additionally, security standards such as X.509 and SSL/TLS have to be supported. The library will be provided to manufacturers of VS-NfD products which will help the Federal Office for Information Security (BSI) to evaluate these products.

This document specifies test cases implemented in the library's test suite.

Authors
René Korthaus (RK), Rohde & Schwarz Cybersecurity
Juraj Somorovsky (JSo), Hackmanit GmbH

Secure Implementation of a Universal Crypto Library

Test Specification
Botan 2.0.1-RSCS1

Changelog

Version	Authors	Comment	Date
0.1.0	RK, JSo	Initial version	2016-10-28
1.0.0	RK	Added SHA-3, AES adjustments	2016-11-11
1.1.0	RK, DN	Add Botan version Fix page numbers Add public key encryption scheme tests Add certificate store tests Add CTR mode tests Add PKCS11 Session and Slot negative tests Add 2,048 bits DH test cases Add Entropy Sources tests Update AEAD, Block Cipher, DH, KDF, MAC, Modes, Public-key Encryption, Public-key signature, RNG, X.509 tests to Botan 2.0.0	2017-01-09
1.2.0	RK	Fix CBC-CTS test vectors sizes Specify TLS cipher suites tested Add DSA to TLS handshake and policy tests Add ECDSA invalid public key tests	2017-03-02

Table of Contents

1 Introduction

This document specifies test cases implemented in the test suite.

It covers AEAD modes, block ciphers, hash functions, public key-based key agreement schemes, key derivation functions, message authentication codes, block cipher modes of operation, password-based key derivation functions, PKCS#11, public key signature schemes, random number generation, TLS and X.509 tests.

All tests are implemented in the `src/tests` directory. Test vectors are located in `src/tests/data`.

Tests of the TLS configuration using external tools such as TLS Attacker and side channel tests are not covered by this document.

2 AEAD Modes

AEAD modes are tested using known answer tests that (1) encrypt a message, (2) decrypt a message and (3) an additional test to check whether AEAD decryption correctly rejects manipulated ciphertexts and manipulated nonces. All the tests are implemented in `src/tests/test_aead.cpp`. The test cases are described in the following.

Test Case No.:	AEAD-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of AEAD encryption
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • Nonce: The nonce used to initialize the AEAD mode (varying length) • In: The test message to be encrypted (varying length) • AD: Additional data to be authenticated (varying length, optional)
Expected Output:	• Out: Ciphertext (varying length depending on the block cipher)
Steps:	1. Create a AEAD_Encryption object 2. Check that the AEAD output length matches the length of *Out* 3. Check that the AEAD mode accepts nonces of the default nonce length 4. Set a modified version of *Key* as a key on the AEAD_Encryption object 5. Set a modified version of associated data *AD* on the AEAD_Encryption object 6. Set a modified version of nonce *Nonce* on the AEAD_Encryption object 7. Pass a random plaintext value into the AEAD_Encryption object 8. Reset the AEAD_Encryption object 9. Set the key *Key* on the AEAD_Encryption object 10. Set the associated data *AD* on the AEAD_Encryption object 11. Set the nonce *Nonce* on the AEAD_Encryption object 12. Calculate the ciphertext of input value *In* and compare the result with the expected output value *Out* 13. If *In* is the empty message, Return 14. If *In* is longer than the block size of the AEAD mode, calculate the ciphertext of input value *In* by encrypting *In* in block size blocks and comparing the result with the expected output value *Out* 15. If *In* is longer than the block size of the AEAD mode, calculate the ciphertext of input value *In* by encrypting *In* in multiples of block size blocks and comparing the result with the expected output value *Out*

Test Case No.:	AEAD-2

Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of AEAD decryption
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • Nonce: The nonce used to initialize the AEAD mode (varying length) • Out: Ciphertext (varying length depending on the block cipher) • AD: Additional data to be authenticated (varying length, optional)
Expected Output:	• In: The original test message (plaintext, varying length)
Steps:	1. Create a AEAD_Decryption object 2. Check that the AEAD output length matches the length of *Out* 3. Check that the AEAD mode accepts nonces of the default nonce length 4. Set a modified version of *Key* as a key on the AEAD_Decryption object 5. Set a modified version of associated data *AD* on the AEAD_Decryption object 6. Set a modified version of nonce *Nonce* on the AEAD_Decryption object 7. Pass a random ciphertext value into the AEAD_Decryption object 8. Reset the AEAD_Decryption object 9. Set the key *Key* on the AEAD_Decryption object 10. Set the associated data *AD* on the AEAD_Decryption object 11. Set the nonce *Nonce* on the AEAD_ Decryption object 12. Calculate the plaintext of input value *Out* and compare the result with the expected output value *In* 13. If *Out* is longer than the block size of the AEAD mode, calculate the plaintext of input value *Out* by decrypting *Out* in block size blocks and comparing the result with the expected output value *In* 14. If *Out* is longer than the block size of the AEAD mode, calculate the plaintext of input value *Out* by decrypting *Out* in multiples of block size blocks and comparing the result with the expected output value *In*

Test Case No.:	AEAD-3
Type:	Negative Test
Description:	Make sure AEAD decryption correctly rejects manipulated ciphertexts and manipulated nonces
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • Nonce: The nonce used to initialize the AEAD mode (varying

	length) • Out: Ciphertext (varying length depending on the block cipher) • AD: Additional data to be authenticated (varying length, optional)
Expected Output:	Decryption shall output an error (throw an exception)
Steps:	1. Create a AEAD_Decryption object 2. Set the key *Key* on the AEAD_Decryption object 3. Set the associated data *AD* on the AEAD_Decryption object 4. Set the nonce *Nonce* on the AEAD_ Decryption object 5. Create a modified version of *Out*, by changing the length of Out or by flipping random bits in *Out* 6. Calculate the plaintext of the modified *Out*, which should throw an exception If *Nonce* is of length n > 0: 7. Create a modified version of *Nonce* by flipping random bits in *Nonce* 8. Set the modified nonce on the AEAD_Decryption object 9. Calculate the plaintext of the original ciphertext *Out*, which should throw an exception End If 10. Create a modified version of *AD*, by changing the length of *AD* or by flipping random bits in *AD* 11. Set the modified associated data on the *AEAD*_Decryption object 12. Set the nonce *Nonce* on the AEAD_ Decryption object 13. Calculate the plaintext of the original ciphertext *Out*, which should throw an exception

2.1 GCM

GCM is tested with the following constraints:

- Number of test cases: 27
- Sources: NIST CAVP, generated using OpenSSL, Project Wycheproof
- Block Cipher: AES-128 and AES-256
- Key: 128 bits, 192 bits, 256 bits
  - Extreme values: 128 bits all zero, 192 bits all zero, 256 bits all zero
- Nonce: 64 bits, 96 bits, 128 bits and 480 bits
  - Extreme values: 128 bits, 480 bits[1]
- Out: 64 bits, 128 bits, 608 bits, 640 bits
- AD: 64 bits, 128 bits, 160 bits, 192 bits, no AD

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/aead/gcm.vec`.

Test Case No.:	AEAD-GCM-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of GCM encryption
Preconditions:	None
Input Values:	Block Cipher = AES-128 Key = 0x00000000000000000000000000000000 (128 bits) Nonce = 0x000000000000000000000000 (96 bits) In = Message of length zero AD = None
Expected Output:	Out = 0x58E2FCCEFA7E3061367F1D57A4E7455A (128 bits)
Steps:	See generic description in test case *AEAD-1*
Notes:	Corresponds to NIST Test Case 1

1 These GCM nonces are not 96 bits and so are hashed with GHASH to produce the counter value. For these inputs the CTR value is very near 2^32, which exposed a bug in GCM when the counter overflowed

3 Certificate Store

The Certificate Store SQLite interface is tested using unit tests that (1) insert, search and remove certificates and keys, (2) revokes certificates and (3) looks up subjects in the store. All the tests are implemented in `src/tests/test_certstor.cpp`.

3.1 Insert, Search and Remove

These unit tests search and remove certificates and private keys stored in the store. The tests are executed with the following constraints:

- Number of test cases: 6
- Cert: X.509v3
- Key: RSA, 2048 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/certstore/`.

Test Case No.:	CERTSTOR-ISR-1
Type:	Positive Test
Description:	Look up and remove certificates and key in the store
Preconditions:	None
Input Values:	• Cert: Certificate stored in the store • Key: Corresponding private key to Cert
Expected Output:	None
Steps:	1. Look up *Cert* by subject DN 2. Look up *Cert* by subject DN and subject key ID 3. Look up *Key* by *Cert* 4. Look up *Cert* by *Key* 5. Remove *Cert* from the store 6. Look up *Cert* by subject DN and subject key ID 7. Remove *Key* from the store 8. Look up *Key* by *Cert*

3.2 Revocation

These unit tests revoke certificates and generate a CRL on certificates stored in the store. The tests are executed with the following constraints:

- Number of test cases: 1
- Cert: X.509v3
- Key: RSA, 2048 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/certstore/`.

Test Case No.:	CERTSTOR-REV-1
Type:	Positive Test
Description:	Revoke certificate and generate a CRL
Preconditions:	None
Input Values:	• Certs: Certificates stored in the store (from `src/tests/data/certstor`) • Keys: Corresponding private keys to Certs
Expected Output:	None
Steps:	1. Revoke *Certs[0]* with reason *CA Compromise* 2. Revoke *Certs[3]* with reason *CA Compromise* 3. Generate CRLs 4. Check that *Certs[0]* and *Certs[3]* are revoked 5. Reverse the revocation of *Cert[3]* 6. Check that *Certs[0]* is still revoked 7. Look up CRL for *Cert[0]* 8. Check that no CRL exists for *Cert[3]*

3.3 Subject DN Listing

These unit tests test the member function `all_subjects()`. The tests are executed with the following constraints:

- Number of test cases: 1
- Cert: X.509v3

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/certstore/`.

Test Case No.:	CERTSTOR-SDN-1
Type:	Positive Test
Description:	List subject DNs of all certificates
Preconditions:	None
Input Values:	• Certs: Certificates stored in the store (from `src/tests/data/certstor`)
Expected Output:	None
Steps:	1. List the distinguished names of all certificates in the certificate store using `all_subjects()` and compare each subject DN with the subject DN from *Certs*

4 Block Ciphers

Block ciphers are tested using (1) unit tests and known answer tests that (2) encrypt a message and (3) decrypt a message. All the tests are implemented in `src/tests/test_block.cpp`. The test cases are described in the following.

Test Case No.:	BLOCK-1
Type:	Positive Test
Description:	Unit Test that checks certain properties of the BlockCipher
Preconditions:	None
Input Values:	• Name: The block cipher name
Expected Output:	
Steps:	1. Create a block cipher object 2. Test the block cipher name 3. Test that block cipher parallelism equals or is greater than one 4. Test that block size equals or is greater than eight 5. Test that block cipher parallel bytes equals *block size*parallel bytes*

Test Case No.:	BLOCK-2
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of block cipher encryption
Preconditions:	None
Input Values:	• Key: The encryption key used for the block cipher (varying length depending on the block cipher) • In: The test message to be encrypted (varying length)
Expected Output:	• Out: Ciphertext (varying length depending on the block cipher)
Steps:	1. Create a block cipher object 2. Set a randomly generated key of length *minimum key length* bits 3. Generate a random plaintext of length *key length* bits and encrypt it 4. Reset the block cipher object 5. Set the key *Key* on the block cipher object 6. Clone the block cipher object 7. Check that cloned object points to a different memory location 8. Check that cloned object has the same block cipher name 9. Set a random key on the cloned object 10. Encrypt the input value *In* and compare the result with the expected value *Out* 11. Decrypt the result from the previous step and compare with the input value *In*

Test Case No.:	BLOCK-3

Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of block cipher decryption
Preconditions:	None
Input Values:	• Key: The decryption key used for the block cipher (varying length depending on the block cipher) • Out: Ciphertext (varying length depending on the block cipher)
Expected Output:	• In: The original test message (plaintext, varying length)
Steps:	1. Create a block cipher object 2. Set the key *Key* on the block cipher object 3. Encrypt the value *In* 4. Decrypt the result from the previous step and compare with the input value *Out*

4.1 AES

The AES tests are executed with the AES software implementation and on systems with SSSE3 support additionally with SSSE3 and on systems with support for hardware acceleration additionally with AES-NI.

AES-128 is tested with the following constraints:

- Number of test cases: 1350
- Source: NIST CAVP AESAVS
- Key: 128 bits
  - Extreme values: 128 bits all zero, only one bit set
- In: 128 bits, 1024 bits
  - Extreme values: 128 bits all zero, only one bit set, 1024 bits
- Out: 128 bits, 1024 bits

AES-192 is tested with the following constraints:

- Key: 192 bits
  - Extreme values: 192 bits all zero, only one bit set
- In: 128 bits, 896 bits
  - Extreme values: 192 bits all zero, only one bit set, 896 bits
- Out: 128 bits, 896 bits

AES-256 is tested with the following constraints:

- Key: 256 bits
  - Extreme values: 256 bits all zero, only one bit set
- In: 128 bits, 640 bits
  - Extreme values: 256 bits all zero, only one bit set, 640 bits
- Out: 128 bits, 640 bits

Note: The BlockCipher interface allows processing multiples of the cipher's block size (via encrypt_n()/decrypt_n()). In this case, processing happens blockwise and the result is concatenated.

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/block/aes.vec`.

Test Case No.:	BLOCK-AES-2
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of AES encryption
Preconditions:	None

Input Values:	Key = 0x000102030405060708090A0B0C0D0E0F (128 bits) In = 0x00112233445566778899AABBCCDDEEFF (128 bits)
Expected Output:	Out = 0x69C4E0D86A7B0430D8CDB78070B4C55A (128 bits)
Steps:	1. Create an AES object 2. Set a randomly generated key of length *minimum key length* bits 3. Generate a random plaintext of length *key length* bits and encrypt it 4. Reset the AES object 5. Set the key *Key* on the AES object 6. Encrypt the input value *In* and compare the result with the expected value *Out* 7. Decrypt the result from the previous step and compare with the input value *In*

5 Entropy Sources

Entropy sources are tested using a system test that polls the entropy source for entropy and checks that entropy was added to given random number generator's entropy pool. Additionally, the entropy returned by the entropy sources is compressed using different compression algorithms and the compressed byte size is compared to the number of entropy bytes returned by the entropy source. All entropy sources in the build-time configuration variable BOTAN_ENTROPY_DEFAULT_SOURCES are tested. In the default configuration these are "rdseed", "rdrand", "darwin_secrandom", "dev_random", "win32_cryptoapi", "proc_walk" and "system_stats". Note that some entropy sources are not available on all platforms and therefore tests are skipped on unsupported platforms.

All the tests are implemented in `src/tests/test_entropy.cpp`.

Entropy sources are tested with the following constraints:

- Number of test cases: 1
- Source: -

Test Case No.:	ENTROPY-1
Type:	Positive Test
Description:	Tests whether each enabled entropy source outputs entropy bytes
Preconditions:	None
Input Values:	None
Expected Output:	None
Steps:	1. Create an Entropy_Sources object from all entropy sources in BOTAN_ENTROPY_DEFAULT_SOURCES using Entropy_Sources::global_sources() 2. Get all sources supported by this platform and for each entropy source do: a) Poll the entropy source using a SeedCapturing_RNG test object and check that the number of entropy bytes added to the SeedCapturing_RNG pool is greater or equal to the entropy estimate returned by the entropy source b) If the entropy source added entropy to the pool in the previous step, check that it added at least one byte and check that it added entropy exactly once

6 Hash Functions

6.1 Hash Function Tests

Hash functions are tested using a (1) combined unit and known answer test that hashes a message as a whole and (2) a known answer test that hashes a message in separate chunks. All the tests are implemented in `src/tests/test_hash.cpp`. The test cases are described in the following.

Test Case No.:	HASH-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	• In: The test message to be hashed (varying length)
Expected Output:	• Out: Message digest (varying length depending on the hash function)
Steps:	1. Create a HashFunction object 2. Test the hash function's name 3. Feed the input value *In* into the hash function 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the hash function 6. Reset the hash function 7. Feed an input value of length zero into the hash function 8. Feed the input value *In* into the hash function 9. Calculate the message digest and compare with the expected output value *Out*

Test Case No.:	HASH-2
Type:	Positive Test
Description:	Known Answer Test that hashes a message in two chunks
Preconditions:	*In* must be of length n > 1 byte
Input Values:	• In: The test message to be hashed (varying length)
Expected Output:	• Out: Message digest (varying length depending on the hash function)
Steps:	1. Feed the first byte of the input value *In* into the hash function 2. Feed the bytes 2..n of the input value *In* into the hash function 3. Calculate the message digest and compare with the expected output value *Out*

6.1.1 MD-5

MD-5 is tested with the following constraints:

- Number of test cases: 76
- In: varying length
  - Range: 1 byte – 67 bytes
  - Extreme values: empty message, 1029 bytes
- Out: 128 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/md5.vec`.

Test Case No.:	HASH-MD5-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xD41D8CD98F00B204E9800998ECF8427E
Steps:	1. Create an MD5 object 2. Test MD5's name 3. Feed the input value *In* into the MD5 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the MD5 6. Reset the MD5 7. Feed an input value of length zero into the MD5 8. Feed the input value *In* into the MD5 9. Calculate the message digest and compare with the expected output value *Out*

6.1.2 SHA-1

SHA-1 is tested with the following constraints:

- Number of test cases: 76
- In: varying length
  - Range: 8 bits – 536 bits
  - Extreme values: empty message, 8232 bits
- Out: 160 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha1.vec`.

Test Case No.:	HASH-SHA1-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xDA39A3EE5E6B4B0D3255BFEF95601890AFD80709 (160 bits)
Steps:	1. Create a SHA1 object 2. Test SHA1's name 3. Feed the input value *In* into the SHA1 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value "*some discarded input*" into the SHA1 6. Reset the SHA1 7. Feed an input value of length zero into the SHA1 8. Feed the input value *In* into the SHA1 9. Calculate the message digest and compare with the expected output value *Out*

6.1.3 SHA-224

SHA-224 is tested with the following constraints:

- Number of test cases: 2
- In: varying length
  - Range: 0 bits, 8 bits
  - Extreme values: empty message, 8 bits message
- Out: 224 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha2_32.vec`.

Test Case No.:	HASH-SHA224-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xD14A028C2A3A2BC9476102BB288234C415A2B01F828EA62AC5B3E42F (224 bits)
Steps:	1. Create a SHA224 object 2. Test SHA224's name 3. Feed the input value *In* into the SHA224 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the SHA224 6. Reset the SHA224 7. Feed an input value of length zero into the SHA224 8. Feed the input value *In* into the SHA224 9. Calculate the message digest and compare with the expected output value *Out*

6.1.4 SHA-256

SHA-256 is tested with the following constraints:

- Number of test cases: 262
- In: varying length
  - Range: 8 byte – 256 bits
  - Extreme values: empty message, 640 bits, only one bit set
- Out: 256 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha2_32.vec`.

Test Case No.:	HASH-SHA256-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xE3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855 (256 bits)
Steps:	1. Create a SHA256 object 2. Test SHA256's name 3. Feed the input value *In* into the SHA256 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the SHA256 6. Reset the SHA256 7. Feed an input value of length zero into the SHA256 8. Feed the input value *In* into the SHA256 9. Calculate the message digest and compare with the expected output value *Out*

6.1.5 SHA-384

SHA-384 is tested with the following constraints:

- Number of test cases: 7
- In: varying length
  - Range: 8 bits – 640 bits
  - Extreme values: empty message, 896 bits
- Out: 384 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha2_64.vec`.

Test Case No.:	HASH-SHA384-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0x38B060A751AC96384CD9327EB1B1E36A21FDB71114BE07434C0 CC7BF63F6E1DA274EDEBFE76F65FBD51AD2F14898B95B
Steps:	1. Create a SHA384 object 2. Test SHA384's name 3. Feed the input value *In* into the SHA384 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the SHA384 6. Reset the SHA384 7. Feed an input value of length zero into the SHA384 8. Feed the input value *In* into the SHA384 9. Calculate the message digest and compare with the expected output value *Out*

6.1.6 SHA-512

SHA-512 is tested with the following constraints:

- Number of test cases: 7
- In: varying length
  - Range: 8 bits – 640 bits
  - Extreme values: empty message, 896 bits
- Out: 512 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha2_64.vec`.

Test Case No.:	HASH-SHA512-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xCF83E1357EEFB8BDF1542850D66D8007D620E4050B5715DC83F4A921D36CE9CE47D0D13C5D85F2B0FF8318D2877EEC2F63B931BD47417A81A538327AF927DA3E (512 bits)
Steps:	1. Create a SHA512 object 2. Test SHA512's name 3. Feed the input value *In* into the SHA512 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value "*some discarded input*" into the SHA512 6. Reset the SHA512 7. Feed an input value of length zero into theSHA512 8. Feed the input value *In* into the SHA512 9. Calculate the message digest and compare with the expected output value *Out*

6.1.7 SHA-512/256

SHA-512/256 is tested with the following constraints:

- Number of test cases: 1
- In: empty message
- Out: 256 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha2_64.vec`.

Test Case No.:	HASH-SHA512-256-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xC672B8D1EF56ED28AB87C3622C5114069BDD3AD7B8F9737498D0C01ECEF0967A
Steps:	1. Create a SHA512_256 object 2. Test SHA512_256's name 3. Feed the input value *In* into the SHA512_256 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value “*some discarded input*” into the SHA512_256 6. Reset the SHA512_256 7. Feed an input value of length zero into the SHA512_256 8. Feed the input value *In* into the SHA512_256 9. Calculate the message digest and compare with the expected output value *Out*

6.1.8 SHA-3/224

SHA-3/224 is tested with the following constraints:

- In: varying length
  - Range: 8 bits – 14644 bytes
  - Extreme values: empty message, 14644 bytes
- Out: 224 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha3.vec`.

Test Case No.:	HASH-SHA3-224-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0x6b4e03423667dbb73b6e15454f0eb1abd4597f9a1b078e3f5b5a6bc7
Steps:	10. Create a SHA3_224 object 11. Test SHA3_224's name 12. Feed the input value *In* into the SHA3_224 13. Calculate the message digest and compare with the expected output value *Out* 14. Feed the string value *"some discarded input"* into the SHA3_224 15. Reset the SHA3_224 16. Feed an input value of length zero into the SHA3_224 17. Feed the input value *In* into the SHA3_224 18. Calculate the message digest and compare with the expected output value *Out*

6.1.9 SHA-3/256

SHA-3/256 is tested with the following constraints:

- In: varying length
  - Range: 8 bits – 13836 bytes
  - Extreme values: empty message, 13836 bytes
- Out: 256 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha3.vec`.

Test Case No.:	HASH-SHA3-256-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xa7ffc6f8bf1ed76651c14756a061d662f580ff4de43b49fa82d80a4b80f8434a
Steps:	1. Create a SHA3_256 object 2. Test SHA3_256's name 3. Feed the input value *In* into the SHA3_256 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value "*some discarded input*" into the SHA3_256 6. Reset the SHA3_256 7. Feed an input value of length zero into the SHA3_256 8. Feed the input value *In* into the SHA3_256 9. Calculate the message digest and compare with the expected output value *Out*

6.1.10 SHA-3/384

SHA-3/384 is tested with the following constraints:

- In: varying length
  - Range: 8 bits – 10604 bytes
  - Extreme values: empty message, 10604 bytes
- Out: 384 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha3.vec`.

Test Case No.:	HASH-SHA3-384-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0x0c63a75b845e4f7d01107d852e4c2485c51a50aaaa94fc61995e71bbee983a2ac3713831264adb47fb6bd1e058d5f004
Steps:	1. Create a SHA3_384 object 2. Test SHA3_384's name 3. Feed the input value *In* into the SHA3_384 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the SHA3_384 6. Reset the SHA3_384 7. Feed an input value of length zero into the SHA3_384 8. Feed the input value *In* into the SHA3_384 9. Calculate the message digest and compare with the expected output value *Out*

6.1.11 SHA-3/512

SHA-3/512 is tested with the following constraints:

- In: varying length
  - Range: 8 bits – 7372 bytes
  - Extreme values: empty message, 7372 bytes
- Out: 512 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hash/sha3.vec`.

Test Case No.:	HASH-SHA3-512-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and hashes a test message as a whole
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xa69f73cca23a9ac5c8b567dc185a756e97c982164fe25859e0d1dcc1475c80a615b2123af1f5f94c11e3e9402c3ac558f500199d95b6d3e301758586281dcd26
Steps:	1. Create a SHA3_512 object 2. Test SHA3_512's name 3. Feed the input value *In* into the SHA3_512 4. Calculate the message digest and compare with the expected output value *Out* 5. Feed the string value *"some discarded input"* into the SHA3_512 6. Reset the SHA3_512 7. Feed an input value of length zero into the SHA3_512 8. Feed the input value *In* into the SHA3_512 9. Calculate the message digest and compare with the expected output value *Out*

6.2 Parallel Hash Function Tests

Test Case No.:	H-PHASH-1
Type:	Positive Test
Description:	Unit test for cloning of a Parallel hash object
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xD41D8CD98F00B204E9800998ECF8427EDA39A3EE5E6B4B0D3255BFEF95601890AFD80709 (288 bits)
Steps:	1. Create a Parallel hash object with MD5 and SHA-160 2. Feed an input value of length zero into the hash function 3. Calculate the message digest and compare with the expected output value *Out* 4. Clone the parallel hash function object 5. Reset the cloned parallel hash function object 6. Feed an input value of length zero into the hash function 7. Calculate the message digest and compare with the expected output value *Out*

Test Case No.:	H-PHASH-2
Type:	Positive Test
Description:	Unit test for construction of a Parallel hash object
Preconditions:	None
Input Values:	In = Input value of length zero
Expected Output:	Out = 0xE3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855CF83E1357EEFB8BDF1542850D66D8007D620E4050B5715DC83F4A921D36CE9CE47D0D13C5D85F2B0FF8318D2877EEC2F63B931BD47417A81A538327AF927DA3E (1536 bits)
Steps:	1. Create a SHA-256 object 2. Create a SHA-512 object 3. Create a Parallel hash object with the SHA-256 and SHA-512 objects 4. Feed an input value of length zero into the hash function 5. Calculate the message digest and compare with the expected output value *Out* 6. Clone the parallel hash function object 7. Reset the cloned parallel hash function object 8. Feed an input value of length zero into the hash function

	9. Calculate the message digest and compare with the expected output value *Out*

7 Key Derivation Functions

Key derivation functions (KDFs) are tested using a known answer test that derives a key from a set of input values. The test is implemented in `src/tests/test_kdf.cpp`. The test case is described in the following.

Test Case No.:	KDF-1
Type:	Positive Test
Description:	Derives a key from the KDF
Preconditions:	None
Input Values:	• Hash Function: The underlying hash function, e.g., SHA-1 or • MAC: The underlying message authentication code, e.g., HMAC-SHA1 • Output Length: The desired output length in bytes (varying length) • Salt: A salt value (varying length, optional) • Secret: The secret input used to derive the key (varying length) • Label: A label value (varying length, optional)
Expected Output:	• Out: The derived key (length depending the desired output length)
Steps:	1. Create the KDF object 2. Input *Output Length*, *Salt (optional)*, *Secret*, and *Label (optional)* into the KDF and compare the result with the expected output value *Out* 3. Clone the KDF object and check that it points to a different memory location

7.1 KDF1 (ISO 18033-2)

KDF1 from ISO 18033-2 is tested with the following constraints:

- Number of test cases: 2
- Source: ISO 18033-2
- Hash Function: SHA-1, SHA-256
- Output Length: 160 bits, 856 bits
- Secret: 160 bits, 856 bits
- Out: 160 bits, 856 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/kdf1_iso18033.vec`.

Test Case No.:	KDF-KDF1-1
Type:	Positive Test
Description:	Derives a key from the KDF
Preconditions:	None
Input Values:	Hash Function = SHA-256 Output Length = 160 bits Secret = 0xD6E168C5F256A2DCFF7EF12FACD390F393C7A88D (160 bits)
Expected Output:	Out = 0x0742BA966813AF75536BB6149CC44FC256FD6406 (160 bits)
Steps:	1. Create the KDF1_18033 object 2. Input *Output Length* and *Secret* into KDF1_18033 and compare the result with the expected output value *Out*

7.2 NIST SP 800-108 (Counter Mode)

The NIST SP 800-108 KDF in Counter Mode is tested with the following constraints:

- Number of test cases: 240
- Source: Generated with BouncyCastle
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512, CMAC-AES128, CMAC-AES192, CMAC-AES256
- Output Length: 16 bits – 160 bits
- Salt: 80 bits – 800 bits
- Secret: 128 bits – 512 bits
- Out: 16 bits – 160 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/sp800_108_ctr.vec`.

Test Case No.:	KDF-NISTSP800-108-CTR-1
Type:	Positive Test
Description:	Derives a key from the NIST SP 800-108 KDF in Counter Mode
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 16 bits Salt = 0x876F7274958C9F920019 (80 bits) Secret = 0x4C5FFEE342D0F1D9204CE138ED131558CF364BBC (160 bits)
Expected Output:	Out = 0x5B3A (16 bits)
Steps:	1. Create the SP800_108_Counter object 2. Input *Output Length*, *Salt* and *Secret* into SP800_108_Counter and compare the result with the expected output value *Out*

7.3 NIST SP 800-108 (Feedback Mode)

The NIST SP 800-108 KDF in Feedback Mode is tested with the following constraints:

- Number of test cases: 240
- Source: Generated with BouncyCastle
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512, CMAC-AES128, CMAC-AES192, CMAC-AES256
- Output Length: 16 bits – 160 bits
- Salt: 144 bits – 1104 bits
- Secret: 128 bits – 512 bits
- Out: 16 bits – 160 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/sp800_108_fb.vec`.

Test Case No.:	KDF-NISTSP800-108-FB-1
Type:	Positive Test
Description:	Derives a key from the NIST SP 800-108 KDF in Feedback Mode
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 16 bits Salt = 0x0976FDEC7817D94D60C4E0C9091D82E38BCFC58D7FFF0829A13D1B4455B8 (240 bits) Secret = 0xE6EA4E4F7178A81230A01DA05705B9C8B902121B (160 bits)
Expected Output:	Out = 0x1092 (16 bits)
Steps:	1. Create the SP800_108_Feedback object 2. Input *Output Length*, *Salt* and *Secret* into SP800_108_Feedback and compare the result with the expected output value *Out*

7.4 NIST SP 800-108 (Pipeline Mode)

The NIST SP 800-108 KDF in Pipeline Mode is tested with the following constraints:

- Number of test cases: 240
- Source: Generated with BouncyCastle
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512, CMAC-AES128, CMAC-AES192, CMAC-AES256
- Output Length: 16 bits – 160 bits
- Salt: 80 bits – 800 bits
- Secret: 128 bits – 512 bits
- Out: 16 bits – 160 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/sp800_108_pipe.vec`.

Test Case No.:	KDF-NISTSP800-108-PI-1
Type:	Positive Test
Description:	Derives a key from the NIST SP 800-108 KDF in Pipeline Mode
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 16 bits Salt = 0xB65A30885B0849C7099B (80 bits) Secret = 0x63CB90F9CD34B95007277AE6FC17FB45A9248725 (160 bits)
Expected Output:	Out = 0x4B0D (16 bits)
Steps:	1. Create the SP800_108_Pipeline object 2. Input *Output Length*, *Salt* and *Secret* into SP800_108_Pipeline and compare the result with the expected output value *Out*

7.5 SP 800-56C

The NIST SP 800-56C KDF is tested with the following constraints:

- Number of test cases: 40
- Source: Generated with PyCryptodome
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512
- Output Length: 16 bits – 400 bits
- Salt: 80 bits – 800 bits
- Secret: 160 bits – 512 bits
- Label: 96 bits
- Out: 16 bits – 400 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/sp800_56c.vec`.

Test Case No.:	KDF-NISTSP800-56C-1
Type:	Positive Test
Description:	Derives a key from the NIST SP 800-56C KDF
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 16 bits Salt = 0x97ca00eac481e8b3556a (80 bits) Label = 0xae8cf2e46773a68098ea53b3 (96 bits) Secret = 0x52f4676023946c7307b5e8148d97f312623a6e88 (160 bits)
Expected Output:	Out = 0x1bcd (16 bits)
Steps:	1. Create the SP800_56C object 2. Input *Output Length*, *Salt*, *Secret* and *Label* into SP800_56C and compare the result with the expected output value *Out*

7.6 TLS 1.0/1.1 PRF

The PRF used in TLS 1.0/1.1 is tested with the following constraints:

- Number of test cases: 30
- MAC: HMAC-MD5, HMAC-SHA1
- Output Length: 8 bits – 256 bits
- Salt: 120 bits – 248 bits
- Secret: 152 bits, 160 bits
- Out: 8 bits – 256 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/tls_prf.vec`.

Test Case No.:	KDF-TLS1-PRF-1
Type:	Positive Test
Description:	Derives a key from the PRF used in TLS 1.0/1.1
Preconditions:	None
Input Values:	MAC = HMAC-MD5, HMAC-SHA1 Output Length = 8 bits Salt = 0xA6D455CB1B2929E43D63CCE55CE89D66F252549729C19C1511 (208 bits) Secret = 0x6C81AF87ABD86BE83C37CE981F6BFE11BD53A8 (152 bits)
Expected Output:	Out = 0xA8 (8 bits)
Steps:	1. Create the TLS_PRF object 2. Input *Output Length*, *Salt* and *Secret* into the TLS_PRF and compare the result with the expected output value *Out*

7.7 TLS 1.2 PRF

The PRF used in TLS 1.2 is tested with the following constraints:

- Number of test cases: 4
- Source: https://www.ietf.org/mail-archive/web/tls/current/msg03416.html
- Hash Function: SHA-224, SHA-256, SHA-384, SHA-512
- Output Length: 88 bits – 196 bits
- Salt: 128 bits
- Secret: 128 bits
- Label: 80 bits
- Out: 88 bits – 196 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/kdf/tls_prf.vec`.

Test Case No.:	KDF-TLS12-PRF-1
Type:	Positive Test
Description:	Derives a key from the PRF used in TLS 1.2
Preconditions:	None
Input Values:	MAC = SHA-224 Output Length = 88 bits Salt = 0xf5a3fe6d34e2e28560fdcaf6823f9091 (128 bits) Secret = 0xe18828740352b530d69b34c6597dea2e (128 bits) Label = 0x74657374206c6162656c (80 bits)
Expected Output:	Out = 0x224d8af3c0453393a9779789d21cf7da5ee62ae6b617873d489428efc8dd58d1566e7029e2ca3a5ecd355dc64d4d927e2fbd78c4233e8604b14749a77a92a70fddf614bc0df623d798604e4ca5512794d802a258e82f86cf (88 bits)
Steps:	1. Create the TLS_12_PRF object 2. Input *Output Length*, *Salt*, *Label* and *Secret* into the TLS_12_PRF and compare the result with the expected output value *Out*

8 Message Authentication Codes

Message authentication codes (MACs) are tested using a (1) combined unit and known answer test that calculates the MAC tag on a message as a whole and (2) a known answer test that calculates the MAC tag on a message in separate chunks. All the tests are implemented in `src/tests/test_mac.cpp`. The test cases are described in the following.

Test Case No.:	MAC-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and calculates the MAC tag on a test message as a whole
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 or • Hash Function: The underlying hash function, e.g., SHA-1 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • In: The test message (varying length)
Expected Output:	• Out: The MAC tag (varying length depending on the block cipher)
Steps:	1. Create the MAC object 2. Test the name of the MAC 3. Set the key *Key* 4. Input *In* into the MAC, calculate the tag and compare it with the expected output value *Out* 5. Set the key *Key* 6. Input the string “some discarded input” into the MAC 7. Reset the MAC 8. Set the key *Key* 9. Input *In* into the MAC 10. Clone the MAC object and check that the cloned object points to a different memory location 11. Check that the cloned and the original MAC object return the same MAC name 12. Set the key *Key* on the cloned object 13. Input 32 random bytes as the message into the cloned object 14. Verify the tag on the original KDF object with the expected output value *Out*

Test Case No.:	MAC-2
Type:	Positive Test
Description:	Calculates the MAC tag on a test message in chunks
Preconditions:	*In* is of length n > 1 byte
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or

	AES-256 or • Hash Function: The underlying hash function, e.g., SHA-1 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • In: The test message (varying length)
Expected Output:	• Out: The MAC tag (varying length depending on the block cipher)
Steps:	1. Create the MAC object 2. Set the key *Key* 3. Feed the first byte of the input value *In* into the MAC 4. Feed the bytes 2..n-1 of the input value *In* into the MAC 5. Feed the last byte of the input value *In* into the MAC 6. Calculate the tag and compare with the expected output value *Out* 7. Set the key *Key* 8. Feed the first byte of the input value *In* into the MAC 9. Feed the bytes 2..n-1 of the input value *In* into the MAC 10. Feed the last byte of the input value *In* into the MAC 11. Input *In* into the MAC and verify the tag with the expected output value *Out*

8.1.1 CMAC

CMAC is tested with the following constraints:

- Number of test cases: 36
- Block Cipher: AES-128, AES-192, AES-256
- Key: 128 bits, 192 bits and 256 bits
- In: varying length
  - Range: 8 bits – 960 bits
  - Extreme values: empty message, 960 bits
- Out: varying length

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/mac/cmac.vec`.

Test Case No.:	MAC-CMAC-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and calculates the CMAC tag on a test message as a whole
Preconditions:	None
Input Values:	Block Cipher = AES-128 Key = 0x2B7E151628AED2A6ABF7158809CF4F3C (128 bits) In = Input value of length zero
Expected Output:	Out = 0xBB1D6929E95937287FA37D129B756746 (128 bits)
Steps:	1. Create the CMAC object 2. Test the name of the CMAC 3. Set the key *Key* 4. Input *In* into the CMAC, calculate the tag and compare it with the expected output value *Out* 5. Set the key *Key* 6. Input the string “some discarded input” into the CMAC 7. Reset the CMAC 8. Set the key *Key* 9. Input *In* into the CMAC and verify the tag with the expected output value *Out*

8.2 HMAC

HMAC is tested with the following constraints:

- Number of test cases: 15
- Hash Function: MD5, SHA-1, SHA-256
- Key: 128 bits, 160 bits, 256 bits
- In: varying length
  - Range: 24 bits – 896 bits
  - Extreme values: 896 bits
- Out: varying length

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/mac/hmac.vec`.

Test Case No.:	MAC-HMAC-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and calculates the HMAC tag on a test message as a whole
Preconditions:	None
Input Values:	Hash Function = MD5 Key = 0x0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B0B (128 bits) In = 0x4869205468657265 (64 bits)
Expected Output:	Out = 0x9294727A3638BB1C13F48EF8158BFC9D (128 bits)
Steps:	10. Create the HMAC object 11. Test the name of the HMAC 12. Set the key *Key* 13. Input *In* into the HMAC, calculate the tag and compare it with the expected output value *Out* 14. Set the key *Key* 15. Input the string “some discarded input” into the HMAC 16. Reset the HMAC 17. Set the key *Key* 18. Input *In* into the HMAC and verify the tag with the expected output value *Out*

8.3 GMAC

GMAC is tested with the following constraints:

- Number of test cases: 15
- Source: Generated with BouncyCastle
- Cipher: AES-128, AES-192, AES-256
- Key: 128 bits, 192 bits, 256 bits
- In: varying length
  - Range: 0 bits – 400 bits
- IV: different 96 bit values, one 32 bit value
- Out: varying length

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/mac/gmac.vec`.

The test vectors were generated with Bouncy Castle Crypto 1.54.

Test Case No.:	MAC-GMAC-1
Type:	Positive Test
Description:	Combined unit and known answer test that checks that reset works correctly and calculates the GMAC tag on a test message
Preconditions:	None
Input Values:	Cipher = AES-128 IV = 0xFFFFFFFFFFFFFFFFFFFFFFFF (96 bits) Key = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF (128 bits) In = 0x00000000000000000000000000000000 (128 bits)
Expected Output:	Out = 0xB19E0699327D423B057C95D258AC3129 (128 bits)
Steps:	1. Create the GMAC object 2. Test the name of the GMAC 3. Set the key *Key* 4. Set the initialization vector *IV* 5. Input *In* into the GMAC, calculate the tag and compare it with the expected output value *Out* 6. Reset the GMAC 7. Set the key *Key* 8. Set the initialization vector *IV* 9. Split the input string *IN* into three arrays and invoke three update functions on the GMAC with these arrays. Calculate the tag and compare it with the expected output value *Out*

9 Modes of Operation

Block cipher modes of operation are tested using known answer tests that (1) encrypt a message and (2) decrypt a message. All the tests are implemented in `src/tests/test_modes.cpp`. The test cases are described in the following.

Test Case No.:	MODE-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of encryption under the mode of operation
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • Nonce: The nonce used to initialize the mode of operation (varying length) • In: The test message to be encrypted (varying length)
Expected Output:	• Out: Ciphertext (varying length depending on the block cipher)
Steps:	1. Create a Cipher_Mode encryption object 2. Set the key *Key* on the Cipher_Mode encryption object 3. Test the name of the mode 4. Test that the mode is not an authenticated mode 5. Set the key *Key* on the Cipher_Mode encryption object 6. Set the nonce *Nonce* on the Cipher_Mode encryption object 7. Calculate the ciphertext of input value *In* and compare the result with the expected output value *Out* 8. If *In* is longer than the block size of the mode, calculate the ciphertext of input value *In* by encrypting *In* in block size blocks and comparing the result with the expected output value *Out* 9. If *In* is longer than the block size of the mode, calculate the ciphertext of input value *In* by encrypting *In* in multiples of block size blocks and comparing the result with the expected output value *Out*

Test Case No.:	MODE-2
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of decryption under the mode of operation
Preconditions:	None
Input Values:	• Block Cipher: The underlying block cipher, e.g., AES-128 or AES-256 • Key: The encryption/decryption key used for the block cipher (varying length depending on the block cipher) • Nonce: The nonce used to initialize the mode of operation (varying length)

	• Out: Ciphertext (varying length depending on the block cipher)
Expected Output:	• In: The original plaintext (varying length)
Steps:	1. Create a Cipher_Mode decryption object 2. Set the key *Key* on the Cipher_Mode decryption object 3. Set the nonce *Nonce* on the Cipher_Mode decryption object 4. Calculate the plaintext of output value *In* and compare the result with the output value *In* 5. If *Out* is longer than the block size of the mode, calculate the plaintext of input value *Out* by decrypting *Out* in block size blocks and comparing the result with the expected output value *In* 6. If *Out* is longer than the block size of the mode, calculate the plaintext of input value *Out* by decrypting *Out* in multiples of block size blocks and comparing the result with the expected output value *In*

9.1 CBC

CBC is tested with the following constraints:

- Number of test cases: 3
- Block Cipher: AES-128, AES-192 and AES-256
- Key: 128 bits, 192 and 256 bits
- Nonce: 128 bits
- In: 512 bits
- Out: 512 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/modes/cbc.vec`.

Test Case No.:	MODE-CBC-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of encryption under CBC
Preconditions:	None
Input Values:	Block Cipher = AES-128 Key = 0x2B7E151628AED2A6ABF7158809CF4F3C (128 bits) Nonce = 0x000102030405060708090A0B0C0D0E0F (128 bits) In = 0x6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C3710 (512 bits)
Expected Output:	Out = 0x7649ABAC8119B246CEE98E9B12E9197D5086CB9B507219EE95DB113A917678B273BED6B8E3C1743B7116E69E222295163FF1CAA1681FAC09120ECA307586E1A7 (512 bits)
Steps:	1. Create a CBC_Encryption object 2. Set the key *Key* on the CBC_Encryption object 3. Test the name of the mode 4. Test that the mode is not an authenticated mode 5. Set the key *Key* on the CBC_Encryption object 6. Set the nonce *Nonce* on the CBC_Encryption object 7. Calculate the ciphertext of input value *In* and compare the result with the expected output value *Out*

9.2 CBC-CTS (CBC-CS3)

CBC-CTS is tested with the following constraints:

- Number of test cases: 6
- Source: RFC 3962
- Block Cipher: AES-128
- Key: 128 bits
- Nonce: 128 bits
- In: 136 bits, 248 bits, 256 bits, 376 bits, 384 bits, 512 bits
- Out: 136 bits, 248 bits, 256 bits, 376 bits, 384 bits, 512 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/modes/cbc.vec`.

Test Case No.:	MODE-CTS-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of encryption under CTS
Preconditions:	None
Input Values:	Block Cipher = AES-128 Key = 0x636869636b656e207465726979616b69 (128 bits) Nonce = 0x00000000000000000000000000000000 (128 bits) In = 0x4920776f756c64206c696b652074686520 (136 bits)
Expected Output:	Out = 0xc6353568f2bf8cb4d8a580362da7ff7f97 (136 bits)
Steps:	1. Create a CTS_Encryption object 2. Set the key *Key* on the CTS_Encryption object 3. Test the name of the mode 4. Test that the mode is not an authenticated mode 5. Set the key *Key* on the CTS_Encryption object 6. Set the nonce *Nonce* on the CTS_Encryption object 7. Calculate the ciphertext of input value *In* and compare the result with the expected output value *Out*

9.3 CTR

CTR mode is a stream cipher mode of operation in the library and thus is tested differently than other block cipher modes of operation. All the stream cipher modes of operation tests are implemented in `src/tests/test_stream.cpp`. CTR mode is tested with the following constraints:

- Number of test cases: 6
- Block Cipher: AES-128, AES-192, AES-256
- Key: 128 bits, 192 bits, 256 bits
- Nonce: 128 bits
- In: 384 bits, 512 bits, 5720 bits, 65536 bits
- Out: 384 bits, 512 bits, 5720 bits, 65536 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/stream/ctr.vec`.

Test Case No.:	MODE-CTR-1
Type:	Positive Test
Description:	Known Answer Test that verifies the correctness of encryption under CTR
Preconditions:	None
Input Values:	Block Cipher = AES-128 Key = 0x2B7E151628AED2A6ABF7158809CF4F3C (128 bits) Nonce = 0xF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF (128 bits) In = 0x6BC1BEE22E409F96E93D7E117393172AAE2D8A571E03AC9C9EB76FAC45AF8E5130C81C46A35CE411E5FBC1191A0A52EFF69F2445DF4F9B17AD2B417BE66C3710 (384 bits)
Expected Output:	Out = 0x874D6191B620E3261BEF6864990DB6CE9806F66B7970FDFF8617187BB9FFFDFF5AE4DF3EDBD5D35E5B4F09020DB03EAB1E031DDA2FBE03D1792170A0F3009CEE (384 bits)
Steps:	1. Create a StreamCipher object 2. Test the name of the mode 3. Set the key *Key* on the StreamCipher object 4. Set the IV *Nonce* on the StreamCipher object 5. Clone the StreamCipher object and check that it has a different pointer but the same name 6. Set a random key on the cloned StreamCipher object 7. Calculate the ciphertext of input value *In* on the original StreamCipher object and compare the result with the expected output value *Out*

10 Password-based Key Derivation Functions

Password-based Key derivation functions (PBKDFs) are tested using a known answer test that derives a key from a set of input values. The test is implemented in `src/tests/test_pbkdf.cpp`. The test case is described in the following.

Test Case No.:	PBKDF-1
Type:	Positive Test
Description:	Derives a key from the PBKDF
Preconditions:	None
Input Values:	• Hash Function: The underlying hash function, e.g., SHA-1 or • MAC: The underlying message authentication code, e.g., HMAC-SHA1 • Output Length: The desired output length in bytes (varying length) • Iterations: The number of iterations • Salt: A salt value (varying length) • Passphrase: The passphrase used to derive the key (varying length)
Expected Output:	• Out: The derived key (length depending the desired output length)
Steps:	1. Create the PBKDF object 2. Input *Output Length*, *Iterations*, *Salt* and *Passphrase* into the PBKDF and compare the result with the expected output value *Out*

10.1 PBKDF1

PBKDF1 from PKCS#5 v1.5 is tested with the following constraints:

- Number of test cases: 5
- Hash Function: SHA-1, SHA-256
- Salt: 160 bits, 256 bits
- Output Length: 112 bits, 152 bits, 160 bits
- Iterations: 6, 2001, 10000
- Passphrase: 12 characters, 20 characters
- Out: 112 bits, 152 bits, 160 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pbkdf/pbkdf1.vec`.

Test Case No.:	PBKDF-PBKDF1-1
Type:	Positive Test
Description:	Derives a key from the PBKDF1
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 152 bits Iterations = 6 Salt = 0x40AC5837560251C275AF5E30A6A3074E57CED38E (160 bits) Passphrase = “ftlkfbxdtbjbvllvbwiw”
Expected Output:	Out = 0x768B277DC970F912DBDD3EDAD48AD2F065D25D (160 bits)
Steps:	1. Create the PBKDF1 object 2. Input *Output Length*, *Iterations*, *Salt* and *Passphrase* into the PBKDF1 and compare the result with the expected output value *Out*

10.2 PBKDF2

PBKDF1 from PKCS#5 is tested with the following constraints:

- Number of test cases: 13
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA384, HMAC-SHA512
- Salt: 64 bits, 160 bits, 240 bits
- Output Length: 80 bits – 512 bits
- Iterations: 1 - 10000
- Passphrase: 3 – 20 characters
  - Extreme values: Empty passphrase
- Out: 80 bits – 512 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pbkdf/pbkdf2.vec`.

Test Case No.:	PBKDF-PBKDF2-1
Type:	Positive Test
Description:	Derives a key from the PBKDF2
Preconditions:	None
Input Values:	MAC = HMAC-SHA1 Output Length = 256 bits Iterations = 10000 Salt = 0x0001020304050607 (64 bits) Passphrase = Empty passphrase
Expected Output:	Out = 0x59B2B1143B4CB1059EC58D9722FB1C72471E0D85C6F7543BA5228526375B0127 (256 bits)
Steps:	1. Create the PBKDF2 object 2. Input *Output Length*, *Iterations*, *Salt* and *Passphrase* into the PBKDF2 and compare the result with the expected output value *Out*

10.3

11 PKCS#11

PKCS#11 functions are tested using a set of system tests for Modules, Slots, Sessions and Objects and system tests for RSA key generation, encryption and signature, ECDSA key generation and signature, ECDH key generation and key derivation, random generator generate and reseeding and X.509 certificate import. Last but not least, the token management functions to initialize a token and setting/changing the user PIN and Security Officer (SO) PIN are tested. All the tests are implemented in `src/tests/test_pkcs11_high_level.cpp`.

PKCS#11 functions are not executed during a regular run of the test suite, but instead must be executed manually:

```
./botan-test --pkcs11-lib=<PATH_TO_PKCS11_MODULE> pkcs11
```

This is because the test suite needs a vendor-specific PKCS#11 module to communicate with the HSM under test.

The token under test must have the User PIN set to 123456 and the SO PIN set to 12345678 prior to running the tests.

11.1 Module Tests

Module tests check that PKCS#11 modules can be loaded and unloaded successfully.

Test Case No.:	PKCS11-MODULE-1
Type:	Negative Test
Description:	Load a PKCS#11 module from a non-existing path
Preconditions:	None
Input Values:	Module Path = “/a/b/c”
Expected Output:	An exception is thrown
Steps:	1. Load a Module from a non-existing path “/a/b/c”

Test Case No.:	PKCS11-MODULE-2
Type:	Positive Test
Description:	Load a PKCS#11 module from a valid path
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib* path
Expected Output:	The module is loaded
Steps:	1. Load a Module from *Module Path*

Test Case No.:	PKCS11-MODULE-3
Type:	Positive Test
Description:	Reload a PKCS#11 module
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	The module is loaded
Steps:	1. Load a Module from *Module Path* 2. Reload the Module from *Module Path* 3. Retrieve Module information

Test Case No.:	PKCS11-MODULE-4
Type:	Negative Test
Description:	Attempt to load the same module twice
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	An error occurs
Steps:	1. Load a Module from *Module Path* 2. Load another Module from the same *Module Path*

Test Case No.:	PKCS11-MODULE-5
Type:	Negative Test
Description:	Attempt to load the same module twice
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	An error occurs
Steps:	1. Load a Module from *Module Path* 2. Load a second Module from the same *Module Path*

Test Case No.:	PKCS11-MODULE-6
Type:	Positive Test
Description:	Retrieve Module information
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Retrieve Module information and check that the Cryptoki major version is not 0

11.2 Slot Tests

Slot tests check whether slots can successfully be enumerated and slot information can be retrieved.

Test Case No.:	PKCS11-SLOT-1
Type:	Positive Test
Description:	Detect available slots
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token and check that the number is 1 or higher

Test Case No.:	PKCS11-SLOT-2
Type:	Positive Test
Description:	Load a specific slot
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token and check that the number is 1 or higher 3. Load a Slot from the first element of the available slots 4. Check that the Slot ID equals the slot id in the list from step 2

Test Case No.:	PKCS11-SLOT-3
Type:	Positive Test
Description:	Retrieve slot info
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Retrieve SlotInfo from Slot 5. Check that SlotInfo description field is not the empty string

Test Case No.:	PKCS11-SLOT-4

Type:	Negative Test
Description:	Test with invalid slot id
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	An exception is thrown
Steps:	1. Load a Module from *Module Path* 2. Get available slots with and without a connected token 3. Select a slot id that is not present in the available slots list 4. Load this Slot from the slot id 5. Retrieve slot info from this Slot

Test Case No.:	PKCS11-SLOT-5
Type:	Positive Test
Description:	Retrieve token info
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Retrieve TokenInfo from Slot 5. Check that TokenInfo label field is not the empty string

Test Case No.:	PKCS11-SLOT-6
Type:	Positive Test
Description:	Retrieve mechanism list
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Retrieve mechanism list from Slot and check that it contains at least one element

Test Case No.:	PKCS11-SLOT-7
Type:	Positive Test
Description:	Retrieve mechanism info
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Retrieve mechanism info for the `RsaPkcsKeyPairGen` mechanism from Slot

11.3 Session Tests

Session tests check whether sessions can be successfully established with a token.

Test Case No.:	PKCS11-SESSION-1
Type:	Positive Test
Description:	Open a read-only session
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-only Session using the Slot

Test Case No.:	PKCS11-SESSION-2
Type:	Negative Test
Description:	Open a read-only session using an invalid slot id
Preconditions:	None
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	An exception is thrown
Steps:	1. Load a Module from *Module Path* 2. Get available slots with and without a connected token 3. Select a slot id that is not present in the available slots list 4. Load this Slot from the slot id 5. Open a read-only Session using the Slot

Test Case No.:	PKCS11-SESSION-3
Type:	Positive Test
Description:	Open a read-write session
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-write Session using the Slot

Test Case No.:	PKCS11-SESSION-4

Type:	Positive Test
Description:	Open a read-write session using dedicated CK_FLAGS
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path* CK_FLAGS = SerialSession | RwSession
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-write Session using the Slot with *CK_FLAGS*

Test Case No.:	PKCS11-SESSION-5
Type:	Positive Test
Description:	Open two sessions in parallel
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-only Session using the Slot 5. Open a read-write Session using the same Slot

Test Case No.:	PKCS11-SESSION-6
Type:	Positive Test
Description:	Reuse the session handle in a second session
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-write Session using the Slot 5. Get the Session handle and invalidate the Session object 6. Create a new Session object and reuse the Session Handle

Test Case No.:	PKCS11-SESSION-7
Type:	Positive Test
Description:	Log into a session with the User PIN

Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-write Session using the Slot 5. Log into the Session with the User PIN 6. Log off from the Session

Test Case No.:	PKCS11-SESSION-8
Type:	Positive Test
Description:	Log into a session with the SO PIN
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Open a read-write Session using the Slot 5. Log into the Session with the SO PIN

11.4 RSA Tests

RSA tests involve key import and export, key generation, signature and verification and encryption and decryption.

Test Case No.:	PKCS11-RSA-1
Type:	Positive Test
Description:	Import an RSA private key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random 2048 bits RSA keypair 2. Set the RSA key to be a token key, to be a private token object, a decryption key and a signature key 3. Import the RSA key into the token using the read-write session 4. Destroy the RSA key in the token

Test Case No.:	PKCS11-RSA-2
Type:	Positive Test
Description:	Export an RSA private key from a token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random 2048 bits RSA keypair 2. Set the RSA key to be a token key, to be a private token object, a decryption key and a signature key, set it to be extractable and not sensitive 3. Import the RSA private key into the token using the read-write session 4. Export the key from the token and compare it with the generated private key 5. Destroy the RSA key in the token

Test Case No.:	PKCS11-RSA-3
Type:	Positive Test
Description:	Import an RSA public key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None

Steps:	1. Generate a random 2048 bits RSA keypair 2. Set the RSA key to be a token key, to not be a private token object and to be a decryption key 3. Import the RSA public key into the token using the read-write session 4. Destroy the RSA public key in the token

Test Case No.:	PKCS11-RSA-4
Type:	Positive Test
Description:	Generate an RSA private key in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an RSA keypair in the token with the following properties: ◦ length = 2048 bits ◦ token key = true ◦ private object = true ◦ signature key = true ◦ decryption key = true 2. Destroy the RSA private key in the token

Test Case No.:	PKCS11-RSA-5
Type:	Positive Test
Description:	Generate an RSA keypair in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an RSA keypair in the token with the following properties: ◦ length = 2048 bits ◦ public key label = “BOTAN_TEST_RSA_PUB_KEY” ◦ private key label = “BOTAN_TEST_RSA_PRIV_KEY” ◦ token key = true ◦ public verification key = true ◦ public key private object = false ◦ private key private object = true ◦ private signature key = true ◦ private decryption key = true 2. Destroy the RSA public key in the token 3. Destroy the RSA private key in the token

Test Case No.:	PKCS11-RSA-6

Type:	Positive Test
Description:	Encrypt and decrypt in the token with no padding
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • signature key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Plaintext = 0x000102030405060708090A0B… (2048 bits)
Expected Output:	None
Steps:	1. Encrypt *Plaintext* using the RSA public key in the token 2. Decrypt the resulting ciphertext and compare the output with the input value *Plaintext* 3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-7
Type:	Positive Test
Description:	Encrypt and decrypt in the token with PKCS#1 v1.5 padding
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • signature key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Plaintext = 0x000102030400 (48 bits)
Expected Output:	None
Steps:	1. Encrypt *Plaintext* using the RSA public key in the token 2. Decrypt the resulting ciphertext and compare the output with the input value *Plaintext*

	3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-8
Type:	Positive Test
Description:	Encrypt and decrypt in the token with OAEP padding (SHA-1)
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • signature key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Plaintext = 0x000102030400 (48 bits)
Expected Output:	None
Steps:	1. Encrypt *Plaintext* using the RSA public key in the token 2. Decrypt the resulting ciphertext and compare the output with the input value *Plaintext* 3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-9
Type:	Positive Test
Description:	Sign and verify a message in the token with no padding
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x000102030405060708090A0B… (2048 bits)

Expected Output:	None
Steps:	1. Sign the *Message* using the RSA private key in the token 2. Verify the resulting signature 3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-10
Type:	Positive Test
Description:	Sign and verify a single-part message in the token with PKCS#1 v1.5 padding (SHA-256)
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x000102030405060708090A0B… (2048 bits)
Expected Output:	None
Steps:	1. Sign the *Message* using the RSA private key in the token 2. Verify the resulting signature 3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-11
Type:	Positive Test
Description:	Sign and verify a single-part message in the token with PKCS#1 PSS padding (SHA-256)
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • public key private object = false • private key private object = true

	• private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x000102030405060708090A0B… (2048 bits)
Expected Output:	None
Steps:	1. Sign the *Message* using the RSA private key in the token 2. Verify the resulting signature 3. Destroy the token private key 4. Destroy the token public key

Test Case No.:	PKCS11-RSA-12
Type:	Positive Test
Description:	Sign and verify a multi-part message in the token with PKCS#1 v1.5 padding (SHA-256)
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x000102030405060708090A0B… (2048 bits)
Expected Output:	None
Steps:	1. Input the first 1024 bits of *Message* into the token signature function 2. Input the second 1024 bits of *Message* into the token signature function 3. Sign using the RSA private key in the token 4. Input the first 1024 bits of *Message* into the token verification function 5. Input the second 1024 bits of *Message* into the token verification function 6. Verify the resulting signature 7. Destroy the token private key 8. Destroy the token public key

Test Case No.:	PKCS11-RSA-13
Type:	Positive Test
Description:	Sign and verify a multi-part message in the token with PKCS#1 PSS padding (SHA-256)

Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An RSA keypair was generated with the following properties: • length = 2048 bits • public key label = “BOTAN_TEST_RSA_PUB_KEY” • private key label = “BOTAN_TEST_RSA_PRIV_KEY” • token key = true • public verification key = true • public key private object = false • private key private object = true • private signature key = true • private decryption key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x000102030405060708090A0B… (2048 bits)
Expected Output:	None
Steps:	1. Input the first 1024 bits of *Message* into the token signature function 2. Input the second 1024 bits of *Message* into the token signature function 3. Sign using the RSA private key in the token 4. Input the first 1024 bits of *Message* into the token verification function 5. Input the second 1024 bits of *Message* into the token verification function 6. Verify the resulting signature 7. Destroy the token private key 8. Destroy the token public key

11.5 ECDSA Tests

ECDSA tests involve key import and export, key generation, and signature and verification.

Test Case No.:	PKCS11-ECDSA-1
Type:	Positive Test
Description:	Import an ECDSA private key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDSA private key on the secp256r1 curve 2. Import the ECDSA key into the token using the read-write session and with the following properties: ◦ token key = true ◦ private object = true ◦ signature key = true ◦ label = “Botan test ecdsa key” 3. Destroy the ECDSA key in the token

Test Case No.:	PKCS11-ECDSA-2
Type:	Positive Test
Description:	Export an ECDSA private key from a token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDSA private key on the secp256r1 curve 2. Import the ECDSA key into the token using the read-write session and with the following properties: ◦ token key = true ◦ private object = true ◦ signature key = true ◦ extractable = true ◦ label = “Botan test ecdsa key” 3. Export the key from the token 4. Destroy the ECDSA key in the token

Test Case No.:	PKCS11-ECDSA-3
Type:	Positive Test
Description:	Import an ECDSA public key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN

Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDSA private key on the secp256r1 curve 2. Import the ECDSA public key into the token using the read-write session and with the following properties: ◦ token key = true ◦ verification key = true ◦ private object = false ◦ label = “Botan test ecdsa pub key” 3. Destroy the ECDSA key in the token

Test Case No.:	PKCS11-ECDSA-4
Type:	Positive Test
Description:	Export an ECDSA public key from the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDSA private key on the secp256r1 curve 2. Import the ECDSA public key into the token using the read-write session and with the following properties: ◦ token key = true ◦ verification key = true ◦ private object = false ◦ label = “Botan test ecdsa pub key” 3. Export the public key and compare it with the generated public key 4. Destroy the ECDSA key in the token

Test Case No.:	PKCS11-ECDSA-5
Type:	Positive Test
Description:	Generate an ECDSA private key in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an ECDSA private key in the token with the following properties: ◦ curve = secp256r1 ◦ token key = true ◦ private object = true ◦ signature key = true 2. Destroy the ECDSA private key in the token

Test Case No.:	PKCS11-ECDSA-6
Type:	Positive Test
Description:	Generate an ECDSA keypair in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an ECDSA keypair in the token with the following properties: ◦ curve = secp256r1 ◦ public key label = “BOTAN_TEST_ECDSA_PUB_KEY” ◦ private key label = “BOTAN_TEST_ECDSA_PRIV_KEY” ◦ token key = true ◦ public key private object = false ◦ private key private object = true ◦ public key modifiable = true ◦ private key modifiable = true ◦ private key sensitive = true ◦ private key extractable = false ◦ public verification key = true ◦ private signature key = true 2. Destroy the ECDSA public key in the token 3. Destroy the ECDSA private key in the token

Test Case No.:	PKCS11-ECDSA-7
Type:	Positive Test
Description:	Sign and verify a message in the token with no padding
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN An ECDSA keypair was generated with the following properties: • curve = secp256r1 • public key label = “BOTAN_TEST_ECDSA_PUB_KEY” • private key label = “BOTAN_TEST_ECDSA_PRIV_KEY” • token key = true • public key private object = false • private key private object = true • public key modifiable = true • private key modifiable = true • private key sensitive = true • private key extractable = false • public verification key = true • private signature key = true
Input Values:	Module Path = *--pkcs11-lib path* Message = 0x01010101010101010101010101010101010101010101010101 (160 bits)

Expected Output:	None
Steps:	1. Sign the *Message* using the ECDSA private key in the token 2. Verify the resulting signature in the token 3. Verify the resulting signature using the software implementation 4. Destroy the token private key 5. Destroy the token public key

11.6 ECDH Tests

Test Case No.:	PKCS11-ECDH-1
Type:	Positive Test
Description:	Import an ECDH private key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDH private key on the secp256r1 curve 2. Import the ECDH key into the token using the read-write session and with the following properties: ◦ token key = true ◦ private object = true ◦ derivation key = true ◦ label = “Botan test ecdh key” 3. Destroy the ECDH key in the token

Test Case No.:	PKCS11-ECDH-2
Type:	Positive Test
Description:	Export an ECDH private key from a token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDH private key on the secp256r1 curve 2. Import the ECDH key into the token using the read-write session and with the following properties: ◦ token key = true ◦ private object = true ◦ derivation key = true ◦ extractable = true ◦ label = “Botan test ecdh key” 3. Export the key from the token 4. Destroy the ECDH key in the token

Test Case No.:	PKCS11-ECDH-3
Type:	Positive Test
Description:	Import an ECDH public key into the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*

Expected Output:	None
Steps:	1. Generate a random ECDH private key on the secp256r1 curve 2. Import the ECDH public key into the token using the read-write session and with the following properties: ◦ token key = true ◦ derivation key = true ◦ private object = false ◦ label = “Botan test ecdh pub key” 3. Destroy the ECDH key in the token

Test Case No.:	PKCS11-ECDH-4
Type:	Positive Test
Description:	Export an ECDH public key from the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate a random ECDH private key on the secp256r1 curve 2. Import the ECDH public key into the token using the read-write session and with the following properties: ◦ token key = true ◦ derivation key = true ◦ private object = false ◦ label = “Botan test ecdh pub key” 3. Export the public key 4. Destroy the ECDH key in the token

Test Case No.:	PKCS11-ECDH-5
Type:	Positive Test
Description:	Generate an ECDH private key in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an ECDH private key in the token with the following properties: ◦ curve = secp256r1 ◦ token key = true ◦ private object = true ◦ derivation key = true 2. Destroy the ECDH private key in the token

Test Case No.:	PKCS11-ECDH-6
Type:	Positive Test
Description:	Generate an ECDH keypair in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate an ECDH keypair in the token with the following properties: ◦ curve = secp256r1 ◦ public key label = “Botan test ECDH key1_PUB_KEY” ◦ private key label = “Botan test ECDH key1_PRIV_KEY” ◦ token key = true ◦ public key private object = false ◦ private key private object = true ◦ public key modifiable = true ◦ private key modifiable = true ◦ private key sensitive = true ◦ private key extractable = false ◦ public derivation key = true ◦ private derivation key = true 2. Destroy the ECDH public key in the token 3. Destroy the ECDH private key in the token

Test Case No.:	PKCS11-ECDH-7
Type:	Positive Test
Description:	Derive a shared secret in the token
Preconditions:	At least one token must be connected A read-write session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path* Two ECDH keypairs were generated in the token with the following properties: • curve = secp256r1 • public key label = “Botan test ECDH key2_PUB_KEY” • private key label = “Botan test ECDH key2_PRIV_KEY” • token key = true • public key private object = false • private key private object = true • public key modifiable = true • private key modifiable = true • private key sensitive = true • private key extractable = false • public derivation key = true • private derivation key = true
Expected Output:	None
Steps:	1. Derive a 256 bit shared secret with the first ECDH key

	1. Derive a 256 bit shared secret with the second ECDH key 2. Check that both derived shared secrets are equal 3. Destroy the first ECDH key in the token 4. Destroy the second ECDH key in the token

11.7 Random Generator Tests

Test Case No.:	PKCS11-RNG-1
Type:	Positive Test
Description:	Request random bytes
Preconditions:	At least one token must be connected A read-only session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Request 20 random bytes from the token 2. Check that not all bytes are null

Test Case No.:	PKCS11-RNG-2
Type:	Positive Test
Description:	Add entropy
Preconditions:	At least one token must be connected A read-only session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Generate 20 random bytes with a software generator 2. Seed the token random generator with the 20 random bytes from step 1

Test Case No.:	PKCS11-RNG-3
Type:	Positive Test
Description:	Use PKCS#11 random generator as seed generator for HMAC_DRBG
Preconditions:	At least one token must be connected A read-only session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Create an instance of the HMAC_DRBG with the PKCS#11 random generator as seed generator 2. Check that HMAC_DRBG is not seeded 3. Request 2048 random bits from HMAC_DRBG 4. Check that HMAC_DRBG is seeded 5. Add the string "Botan PKCS#11 Tests" as additional entropy into HMAC_DRBG 6. Request 2048 random bits from HMAC_DRBG 7. Check that not all bytes are null

11.8 X.509 Tests

X.509 tests load an X.509 certificate into a token.

Test Case No.:	PKCS11-X509-1
Type:	Positive Test
Description:	Load an X.509 certificate into the token
Preconditions:	At least one token must be connected A read-only session is open with the token using the User PIN
Input Values:	Module Path = *--pkcs11-lib path* Certificate File = “src/tests/data/nist_x509/test01/end.crt”
Expected Output:	None
Steps:	1. Load certificate from *Certificate File* 2. Import the certificate into the token with the following properties: ◦ label = “Botan PKCS#11 test certificate” ◦ private object = false ◦ token object = true 3. Create a copy of the certificate using the object handle and compare both certificates 4. Destroy the certificate in the token

11.9 Token Management

Token management tests initialize a token and set and change User PIN and SO PIN.

Test Case No.:	PKCS11-MGMT-1
Type:	Positive Test
Description:	Set the User PIN with the SO PIN
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Set the User PIN to 654321 using the SO PIN 12345678 5. Set the User PIN to 123456 using the SO PIN 12345678

Test Case No.:	PKCS11-MGMT-2
Type:	Positive Test
Description:	Initialize a token
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Initialize the token and set the User PIN to 123456 and the the SO PIN to 12345678

Test Case No.:	PKCS11-MGMT-3
Type:	Positive Test
Description:	Change User PIN with the User PIN
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Set the User PIN to 654321 using the User PIN 123456 5. Set the User PIN to 123456 using the User PIN 654321

Test Case No.:	PKCS11-MGMT-4
Type:	Positive Test
Description:	Change SO PIN with the SO PIN
Preconditions:	At least one token must be connected
Input Values:	Module Path = *--pkcs11-lib path*
Expected Output:	None
Steps:	1. Load a Module from *Module Path* 2. Get available slots with a connected token 3. Load a Slot from the first element of the available slots 4. Set the SO PIN to 87654321 using the SO PIN 12345678 5. Set the SO PIN to 12345678 using the SO PIN 87654321

12 Public Key-based Encryption Algorithms

Public Key-based Encryption Algorithms are divided into hybrid encryption schemes and public key encryption schemes. Some public key-based encryption algorithms use test classes implemented in `src/tests/test_pubkey.cpp`.

12.1 Hybrid Encryption Schemes

12.1.1 DLIES

The Discrete Logarithm Integrated Encryption Scheme (DLIES) is tested with the following constraints:

- Number of test cases: 37
- Source: Generated with BouncyCastle
- KDF: KDF1-18033
- Hash Function: SHA-1, SHA-256, SHA-512
- MAC: HMAC-SHA1, HMAC-SHA256, HMAC-SHA512
- IV: 128 bits
- X1: 232 bits
- X2: 232 bits
- Group (P, Q, G): 2048 bits (MODP Group, RFC 3526)
- Cipher: XOR, AES-256/GCM
- Msg: 256 bits
- Ciphertext: 2432 bits - 2944 bits

All the tests are implemented in `src/tests/test_dlies.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/dlies.vec`.

Test Case No.:	PKENC-DLIES-1
Type:	Positive Test
Description:	Encrypt and decrypt a secret
Preconditions:	None
Input Values:	KDF= KDF1-18033(SHA-512) MAC = HMAC(SHA-512) Group = modp/ietf/2048 IV = 0x00112233445566778899aabbccddeeff X1 = 0x4316760088048858173826993660634587631078362099236037980378049883427191 X2 = 0x3824157470039532100357278938102046076290169354062923298804711018976423 Msg = 0x75dad921764736e389c4224daf7b278ec291e682044742e2e9c7a025b54dd62f

Expected Output:	Ciphertext = 0x57DFAFA0D81AC3AACA2570AD13CCCD127239F4EE04843BB738234588F0DAEA53CCD8AF65A5A00ED19FBB6F2EB57779FF2E38E3D5D27986253A1193DABF14D2402E1A33527866FA21F23F7ABBEE5F454AAD762FC90139C8377BF6CC77AF7F982404BAEA5CA4831DD8ED28BABF2D43B1F65EFF42167B82F020DFD4928D8E96DCB7845ECF8F560FBBF5646FAE5BC4EDA6D978E5FB333843A1F4525CFBDDE756842A1E353F4DE1503738EEC6C9D901A78CDEFEDF8DAAA49631DA674B44CAB2193C778BF29766730A656B42E96F84698F77913C718067048263034CF2A2F34572AB662E4B1C5B04CD71183433C591ABD5613820544D46F7462BEA57E44F23AB06E0FB9A0B0CAB5C285FB0CB1F788213B6B82A2C2E485C1D514BAEF7FC241D57DB031D9E80361C55B562232759A660C89E0DE0E11BB8C807142C1C98C07C9BD08BFC7A3D9977133AD07DDED60728B46D668444A74BC001CFBFB8E8FE0BACF6A4078DD4212DC7CDC3291CB3F02AC0B7CDF6E65D
Steps:	1. Create a DH_PrivateKey object *P1* from *P, Q, G* and *X1* 2. Create a DH_PrivateKey object *P2* from *P, Q, G* and *X2* 3. Use P1, P2, the *KDF*, *MAC* and *IV* to encrypt the *Msg* and compare with *Ciphertext* 4. Use P2, P1, the *KDF*, *MAC* and *IV* to decrypt the *Ciphertext* and compare with *Msg*

Test Case No.:	PKENC-DLIES-2
Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	KDF = KDF1-18033(SHA-512) MAC = HMAC(SHA-512) Group = modp/ietf/2048 IV = 0x00112233445566778899aabbccddeeff X1 = 0x4316760088048858173826993660634587631078362099236037980378049883427191 X2 = 0x3824157470039532100357278938102046076290169354062923298804711018976423 Msg = 0x75dad921764736e389c4224daf7b278ec291e682044742e2e9c7a025b54dd62f
Expected Output:	Invalid ciphertexts should not decrypt correctly
Steps:	1. Create a DH_PrivateKey object *P1* from *P, Q, G* and *X1* 2. Create a DH_PrivateKey object *P2* from *P, Q, G* and *X2* 3. Use P2, P1, the *KDF*, *MAC* and *IV* to decrypt the *Ciphertext* and compare with *Msg*

12.1.2 ECIES

The Elliptic Curve Integrated Encryption Scheme (ECIES) is tested with the following constraints:

- Number of test vectors: 2
- Source: ISO/IEC 18033-2:2006
- Format: uncompressed, compressed
- P: 192 bits
- A: 192 bits
- B: 191 bits
- MU: 192 bits (order)
- NU: 8 bits (cofactor)
- Gx: 189 bits (base point x)
- Gy: 187 bits (base point y)
- Hx: 189 bits (x of public point of bob)
- Hy: 191 bits (y of public point of bob)
- X: 192 bits (private key of bob)
- R: 188 bits (ephemeral private key of alice)
- C0: 200 bits, 392 bits (expected encoded ephemeral public key)
- K: 1024 bits (expected derived secret)
- Cofactor Mode: enabled, disabled
- Old Cofactor Mode: enabled, disabled
- Check Mode: enabled, disabled
- Single Hash Mode: enabled, disabled
- Kdf: KDF2(SHA-1)
- Cipher: AES-256/CBC (cipher used to encrypt data)
- CipherKeyLen: 256 bits
- Mac: HMAC(SHA-1) (MAC used to authenticate data)
- MacKeyLen: 160 bits

All the tests are implemented in `src/tests/test_ecies.cpp`. All test vectors are listed in `src/tests/data/pubkey/ecies-18033.vec`. It contains only two test vectors, but all combinations of cofactor mode, single hash mode, old cofactor mode, check mode and compression mode are tested with these two test vectors, so all in all, 96 test cases are executed, 48 tests for each

test vector. As only one of the modes cofactor mode, old cofactor mode and check mode can be enabled at a time, the test cases where two or more of these modes are enabled do not encrypt/decrypt, but instead only check that the combination of these modes lead to an exception (negative test). In the following one positive and one negative test is shown.

Test Case No.:	PKENC-ECIES-1
Type:	Positive Test
Description:	Derive a shared secret and encrypt/decrypt
Preconditions:	None
Input Values:	P = 0xffffffffffffffffffffffffffffffeffffffffffffffff A = 0xffffffffffffffffffffffffffffffeffffffffffffffc B = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1 MU = 0xffffffffffffffffffffffff99def836146bc9b1b4d22831 NU = 0x01 Gx = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012 Gy = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811 Hx = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130 Hy = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2 X = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3 Format = uncompressed Cofactor Mode = enabled Old Cofactor Mode = disabled Single Hash Mode = disabled Check Mode = disabled Kdf = KDF2(SHA-1) Cipher = AES-256/CBC CipherKeyLen = 256 bits Mac = HMAC(SHA-1) MacKeyLen = 160 bits Plaintext = 0x010203
Expected Output:	K = 0x9a709adeb6c7590ccfc7d594670dd2d74fcdda3f8622f2dbcf0f0c02966d5d9002db578c989bf4a5cc896d2a11d74e0c51efc1f8ee784897ab9b865a7232b5661b7cac87cf4150bdf23b015d7b525b797cf6d533e9f6ad49a4c6de5e7089724c9cadf0adf13ee51b41be6713653fc1cb2c95a1d1b771cc7429189861d7a829f3
Steps:	1. Create an ECDH_PrivateKey object *PR1* from *P, A, B, Gx, Gy, MU, NU, X* 2. Create an ECDH_PublicKey object *PU1* P, A, B, Hx, Hy 3. Create an ECDH_PrivateKey object *PR2* from *P, A, B, Gx, Gy, MU, NU, R* 4. Encode the public point of *PR2* using *Format* and compare with expected output *C0* 5. Use PR1 and PU1 to derive a shared secret of 128 bytes using KDF1-18033(SHA-1) and *Format* and compare with expected output *K* 6. Create an ECIES_System_Params object ESP from *P*, *A*, *B*, *Kdf*, *Cipher*, *CipherKeyLen*, *Mac*, *MacKeyLen*, *Format* and *Cofactor Mode*, *Old Cofactor Mode*, *Single Hash Mode* and *Check Mode*

	7. Create an ECIES_Encryptor from PR1 and ESP 8. Set the public point of PR2 as the public key of the other party on the ECIES_Encryptor 9. Create an ECIES_Decryptor from PR2 and ESP 10. Set the public point of PR2 as the public key of the other party on the ECIES_Decryptor 11. Set the IV on the ECIES_Encryptor to 16 zero bytes 12. Set the IV on the ECIES_Decryptor to 16 zero bytes 13. Encrypt the *Plaintext* using the ECIES_Encryptor 14. Decrypt the ciphertext generated by the previous step using the ECIES_Decryptor and compare the output with the *Plaintext* 15. Negate the last byte of the previously generated ciphertext and check that decryption using the ECIES_Decryptor throws an exception

Test Case No.:	PKENC-ECIES-2
Type:	Negative Test
Description:	Derive a shared secret test that encrypt/decrypt is not possible using the combination of cofactor mode, old cofactor mode and check mode
Preconditions:	None
Input Values:	P = 0xffffffffffffffffffffffffffffffeffffffffffffffff A = 0xffffffffffffffffffffffffffffffeffffffffffffffc B = 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1 MU = 0xfffffffffffffffffffffff99def836146bc9b1b4d22831 NU = 0x01 Gx = 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012 Gy = 0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811 Hx = 0x1cbc74a41b4e84a1509f935e2328a0bb06104d8dbb8d2130 Hy = 0x7b2ab1f10d76fde1ea046a4ad5fb903734190151bb30cec2 X = 0xb67048c28d2d26a73f713d5ebb994ac92588464e7fe7d3f3 Format = uncompressed Cofactor Mode = enabled Old Cofactor Mode = enabled Single Hash Mode = disabled Check Mode = disabled Kdf = KDF2(SHA-1) Cipher = AES-256/CBC CipherKeyLen = 256 bits Mac = HMAC(SHA-1) MacKeyLen = 160 bits Plaintext = 0x010203
Expected Output:	K = 0x9a709adeb6c7590ccfc7d594670dd2d74fcdda3f8622f2dbcf0f0c02966d5d9002db578c989bf4a5cc896d2a11d74e0c51efc1f8ee784897ab9b865a7232b5661b7cac87cf4150bdf23b015d7b525b797cf6d533e9f6ad49a4c6de5e7089724c9cadf0adf13ee51b41be6713653fc1cb2c95a1d1b771cc7429189861d7a829f3
Steps:	1. Create an ECDH_PrivateKey object *PR1* from *P, A, B, Gx, Gy,*

	MU, NU, X 2. Create an ECDH_PublicKey object *PU1* P, A, B, Hx, Hy 3. Create an ECDH_PrivateKey object *PR2* from *P, A, B, Gx, Gy, MU, NU, R* 4. Encode the public point of *PR2* using *Format* and compare with expected output *C0* 5. Use PR1 and PU1 to derive a shared secret of 128 bytes using KDF1-18033(SHA-1) and *Format* and compare with expected output *K* 6. Create an ECIES_System_Params ESP object from *P*, *A*, *B*, *Kdf*, *Cipher*, *CipherKeyLen*, *Mac*, *MacKeyLen*, *Format* and *Cofactor Mode*, *Old Cofactor Mode*, *Single Hash Mode* and *Check Mode* and check that it throws an exception

12.1.3 RSA-KEM

The RSA Key Encapsulation Mechanism (RSA-KEM) is tested with the following constraints:

- Number of test cases: 3
- Source: Generated with BouncyCastle
- KDF: KDF1-18033
- Hash Function: SHA-1, SHA-256, SHA-512
- E: 17
- P: 1024 bits
- Q: 1024 bits
- C0: 512 bits, 2048 bits
- K: 2432 bits - 2944 bits

All the tests are implemented in `src/tests/test_rsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/rsa_kem.vec`.

Test Case No.:	PKENC-RSAKEM-1
Type:	Positive Test
Description:	Derive a shared secret
Preconditions:	None
Input Values:	KDF= KDF1-18033 Hash Function = SHA-1 E = 17 P = 164595018656847388234196458295155176106758058516345827114376462850563872821063372112958430530617671033587308745561238441006073716102223570442822100777454385735694646754229560608162424597515812243913409386743169797403795135840467301322375842101624289696215748957306098326616232554693866253333994954431119962 69 Q = 154815693339461674971201202928063553732348769555838450004553011845712199598612461913292296568174793540787763943903927157071815682359748526650950854481712029197298601776364230444684691118479599447186381098181319184319389074673921642098571884038579323293539363273392989580933234215294363547330708372978868708523
Expected Output:	K = 0x2879A51427541B4CDAC3AD823C75FB2B4CF895BFC8F08DF4F1355CCE27C5A544B3701E91D4E6A8FB9FA7762168974202D6719DA117AB506386F6BAED09F1F8FB84620684AE4C962C05CE130D6BA770F1A54CA8C68CCEA59702DE33DDF456B0F34813CC8BFE6999C

	6086B5EE96122669EAF85FD427D6EC80250FB86D39AAEA752A57 EDE4AD5802B709B536A42F1C9285BAA73884DA2E22204C0D6040 4DE70E24D03BBA5ED3A453782D0B49800EDCE562FE2793B6C9A A59881FB29992BDA65C67BF2625EBCBC66EE87F734C95DDFEC8 08EF6D44DD9682801F26D0F91F60F85F01A1A3D197CD13DFC2B1 74F4BE14CBB14A5946F8E22E9AC492472707DB684B85E0E 0x57DFAFA0D81AC3AACA2570AD13CCCD127239F4EE04843BB73 8234588F0DAEA53CCD8AF65A5A00ED19FBB6F2EB57779FF2E38 E3D5D27986253A1193DABF14D2402E1A33527866FA21F23F7ABB EE5F454AAD762FC90139C8377BF6CC77AF7F982404BAEA5CA483 1DD8ED28BABF2D43B1F65EFF42167B82F020DFD4928D8E96DCB 7845ECF8F560FBBF5646FAE5BC4EDA6D978E5FB333843A1F4525 CFBDDE756842A1E353F4DE1503738EEC6C9D901A78CDEFEDF8D AAA49631DA674B44CAB2193C778BF29766730A656B42E96F84698 F77913C718067048263034CF2A2F34572AB662E4B1C5B04CD71183 433C591ABD5613820544D46F7462BEA57E44F23AB06E0FB9A0B0 CAB5C285FB0CB1F788213B6B82A2C2E485C1D514BAEF7FC241D 57DB031D9E80361C55B562232759A660C89E0DE0E11BB8C807142 C1C98C07C9BD08BFC7A3D9977133AD07DDED60728B46D668444 A74BC001CFBFB8E8FE0BACF6A4078DD4212DC7CDC3291CB3F0 2AC0B7CDF6E65D C0 = 0xC03666B82F2E0076C9CF78056F3BE5549A2BD03349D0D52160C 3D9C1C2B46FB4E65642B340EE73EE73D301CE8DB75A5CDF5B972 011490758A1E0314E0E7E4B952A546FBA6EE8AA7370B6773D6E59 1D2561148FD049E571A5D8AEAF2BE9EA90F15FFE2736D62AC13B B6C2BA0FC993E7CD72FA890E50DBF27554D3BF7F1B913107F201 C6D9EA3E56C53E5683C763C0E7E23F1CD416CBCAD7A6A688AB4 00CBC5D87B1D6DD3612E2615C87B398AE42B43FD5CEAF762033 AC3860C38E96CEF3E5B1180C0EB5DE5D33138131A78D12B4E826 ACE6BE2F1954CD56716D3BD7FE23C7187EE40E34BF5CD0F01B0F 9A6DE390830EC71CB9021ADBCE5AE761E6A1439E157E01
Steps:	1. Create a Private_Key object from *P, Q, G* 2. Use the Private_Key and the *KDF* to derive a shared secret, compare the shared secret to expected output *K* and the encapsulated key to expected output *C0* 3. Use the Private_Key and the *KDF* to decrypt the input value *C0* and compare the output to expected output *K*

12.2 Public Key Encryption Algorithms

12.2.1 RSA

RSA encryption and decryption are tested with the following constraints:

- Number of test cases: 123
- E: 3 - 2147483647
- P: 256 bits – 1024 bits
- Q: 256 bits – 1024 bits
- Msg: 32 bits – 1024 bits
- Nonce: 88 - 904 bits (optional)
- Padding: Raw, EME1(SHA-1), EME-PKCS1-v1_5(SHA-1)
- Ciphertext: 512 bits – 2048 bits

All the tests are implemented in `src/tests/test_rsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/rsaes.vec`.

Test Case No.:	PKENC-RSAES-1
Type:	Positive Test
Description:	Encrypt and decrypt
Preconditions:	None
Input Values:	E = 0x3ED19 P = 0xD987D71CC924C479D30CD88570A626E15F0862A9A138874F7016684216984215 Q = 0xC5660F33AB35E41CB10A30D3A58354ADB5CC3243342C22E1A5BCCB79C391A533 Msg = 0x098825DEC8B4DAB5765348CEE92C4C6A527A172E4A4311399B0B02914E75822F1789B583180ADEADE98C200B7B7670D7B9FBA19946F3D8A7FC8322F80CF67C Padding = Raw
Expected Output:	Ciphertext = 0xA54A45C5F534A6C727212802CD4B2A0B9D0069EFE32B1D239D3B13958BC49711E1CA5BB499FBF7402B6006E654C719C5FB7614C7C00699866B38445228EC7663
Steps:	1. Create the *Private_Key* object from *P, Q, E* 2. Decrypt the *Ciphertext* with the Private_Key object and compare with the *Msg* 3. Encrypt the *Msg* with the Public_Key object and compare with the *Ciphertext*

	4. Decrypt the generated ciphertext from the previous step and compare with the *Msg*

Test Case No.:	PKENC-RSAES-2
Type:	Negative Test
Description:	Invalid ciphertexts should not decrypt correctly
Preconditions:	None
Input Values:	E = 0x3ED19 P = 0xD987D71CC924C479D30CD88570A626E15F0862A9A138874F7016684216984215 Q = 0xC5660F33AB35E41CB10A30D3A58354ADB5CC3243342C22E1A5BCCB79C391A533 Msg = 0x098825DEC8B4DAB5765348CEE92C4C6A527A172E4A4311399B0B02914E75822F1789B583180ADEADE98C200B7B7670D7B9FBA19946F3D8A7FC8322F80CF67C Ciphertext = 0xA54A45C5F534A6C727212802CD4B2A0B9D0069EFE32B1D239D3B13958BC49711E1CA5BB499FBF7402B6006E654C719C5FB7614C7C00699866B38445228EC7663 Padding = Raw
Expected Output:	
Steps:	1. Create the Private_Key object from *P, Q, E* 2. Create a modified version of the *Ciphertext* by changing the length of it or by flipping random bits in it 3. Decrypt the modified *Ciphertext* compare it to the *Msg*

13 Public Key-based Key Agreement Schemes

Public-key based Key Agreement Schemes are tested using a known answer test that derives a key from a set of input values and tests that generate and unit test keys. However, the input values differ for the tested algorithms Diffie Hellman and Elliptic Curve Diffie Hellman such that these test cases are described separately for each algorithm.

Additionally, for each scheme unit tests make sure that encoding and decoding private and public keys works correctly. These tests are implemented in `src/tests/test_pubkey.cpp`. These test cases are described here in the following.

Test Case No.:	KA-KEY-1
Type:	Positive Test
Description:	Encode and decode a key agreement public key as PEM
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group*/*Curve* 2. Encode the public key as PEM-encoded string 3. Create a Public_Key object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid[2]

Test Case No.:	KA-KEY-2
Type:	Positive Test
Description:	Encode and decode a key agreement public key as BER
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group*/*Curve* 2. Encode the public key as BER-encoded byte array 3. Create a Public_Key object from the BER-encoded byte array, decoding the BER-encoded key 4. Check that the key object is valid[1] 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid

2 The exact mechanism depends on the key type and is explained in the corresponding key agreement section

Test Case No.:	KA-KEY-3
Type:	Positive Test
Description:	Encode and decode a key agreement private key as PEM
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group*/*Curve* 2. Encode the private key as PEM-encoded string 3. Create a Private_Key object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid (see KA-KEY-1)

Test Case No.:	KA-KEY-4
Type:	Positive Test
Description:	Encode and decode a key agreement private key as BER
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group*/*Curve* 2. Encode the private key as BER-encoded byte array 3. Create a Private_Key object from the BER-encoded byte array, decoding the BER-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid (see KA-KEY-1)

Test Case No.:	KA-KEY-5
Type:	Positive Test
Description:	Encode and decode a key agreement private key as PEM, protected with a password
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1

Expected Output:	None
Steps:	1. Generate a random password string of length between 1-32 characters 2. Generate a random keypair on the *Group*/*Curve* 3. Encode the private key as PEM-encoded string, protected with the password 4. Create a Private_Key object from the PEM-encoded string, decoding the PEM-encoded key 5. Check that the key object is valid 6. Check that the key object algorithm name equals that of the generated keypair 7. Check that the key is valid (see KA-KEY-1)

Test Case No.:	KA-KEY-6
Type:	Positive Test
Description:	Encode and decode a key agreement private key as BER, protected with a password
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1
Expected Output:	None
Steps:	1. Generate a random password string of length between 1-32 characters 2. Generate a random keypair on the *Group*/*Curve* 3. Encode the private key as BER-encoded byte array, protected with the password 4. Create a Private_Key object from the BER-encoded byte array, decoding the BER-encoded key 5. Check that the key object is valid 6. Check that the key object algorithm name equals that of the generated keypair 7. Check that the key is valid (see KA-KEY-1)

13.1 Diffie-Hellman

The Diffie-Hellman key agreement scheme is tested with a known answer test as follows. The test is mplemented in `src/tests/test_dh.cpp`.

Test Case No.:	KA-DH-1
Type:	Positive Test
Description:	Derives a shared key from the Diffie Hellman Key Agreement Scheme
Preconditions:	None
Input Values:	• P: The prime p (varying length) • G: The base g (varying length) • X: The key's secret value (varying length) • Y: The other party's public value (varying length) • KDF: The underlying key derivation function, e.g., KDF2(SHA-1) (optional) • Output Length: The desired length of the derived shared secret (optional, only used when a KDF is used; otherwise the full output of DH is used)
Expected Output:	• K: The derived shared secret (length depending on the desired output length)
Steps:	1. Create the DH object (input *P*, *G*, *X*) 2. Input *Output Length* (optional) and *P*, *G*, *Y* into the DH and compare the result with the expected output value *K*

Diffie-Hellman key agreement is tested with the following constraints:

- Number of test cases: 40
- Sources: NIST CAVP file 20.1, other
- P: 512 bits, 768 bits, 1024 bits, 1536 bits, 2048 bits
- G: 2, 3, 5 (Zahlenwerte), 2045 bits, 2048 bits
- X: 119 bits – 1535 bits
- Y: 254 bits – 2048 bits
- KDF: None
- Output Length: None, 40 bits, 128 bits, 152 bits, 264 bits
- K: 40 bits, 128 bits, 152 bits, 256 bits, 264 bits, 512 bits, 1024 bits, 1536 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/dh.vec`.

Test Case No.:	KA-DH-1
Type:	Positive Test

Description:	Derives a shared key from the Diffie Hellman Key Agreement Scheme
Preconditions:	None
Input Values:	P = 584580020955360946586837552585233629614212007514394561597561641914945762794 67 G = 2 X = 462056630935896126687461638608709639122263791311908121635193 4948482914728987 48 Y = 268214005722980743583750739227154984032735833676174027819467773132088456286733 KDF = None
Expected Output:	K = 0x5D9A64F9E54B011381308CF462C207CB0DB7630EAB026E06E5B893041207DBD8
Steps:	1. Create the DH object (input *P*, *G*, *X*) 2. Input *Output Length* (optional) and *P*, *G*, *Y* into the DH and compare the result with the expected output value *K*

Additional two unit tests check that DH only accept public key values 1 <= Y <= P-1.

Test Case No.:	KA-DH-2
Type:	Negative Test
Description:	Makes sure Diffie Hellman Key Agreement Scheme does not accept a public key value Y > P-1
Preconditions:	None
Input Values:	P = 584580020955360946586837552585233629614212007514394561597561641914945762794 67 G = 2 X = 462056630935896126687461638608709639122263791311908121635193 4948482914728987 48 Y = 584580020955360946586837552585233629614212007514394561597561641914945762794 672 Output Length = 128 bits KDF = None
Expected Output:	DH outputs an error
Steps:	1. Create the DH object (input *P*, *G*, *X*) 2. Input *Output Length* and *P*, *G*, *Y* into the DH and compute the shared secret
Test Case No.:	KA-DH-3
Type:	Negative Test

Description:	Makes sure Diffie Hellman Key Agreement Scheme does not accept a public key value Y <= 1
Preconditions:	None
Input Values:	P = 584580020955360946586837552585233629614212007514394561597561641914945762794 67 G = 2 X = 462056630935896126687461638608709639122263791311908121635193498484829147289874 8 Y = 1 Output Length = 128 bits KDF = None
Expected Output:	DH outputs an error
Steps:	1. Create the DH object (input *P*, *G*, *X*) 2. Input *Output Length* and *P*, *G*, *Y* into the DH and compute the shared secret

The following example shows a DH-specific KA-KEY-1 test case. The constraints for this test case are:

- Group: modp/ietf/1024, modp/ietf/2048

Test Case No.:	KA-KEY-DH-1
Type:	Positive Test
Description:	Encode and decode a DH key agreement public key as PEM
Preconditions:	None
Input Values:	Group = modp/ietf/1024
Expected Output:	None

Steps:	1. Generate a random keypair on the DH *Group*
	2. Encode the public key as PEM-encoded string
	3. Create a DH_Public_Key object from the PEM-encoded string, decoding the PEM-encoded key
	4. Check that the key object is valid
	5. Check that the key object algorithm name equals that of the generated keypair
	6. Check that the key is valid by checking that:
	  1. $1 < Y < P$
	  2. $G >= 2$
	  3. $P >= 3$
	  4. If Q is given:
	  a) $(P – 1) \% Q = 0$
	  b) $G^Q \bmod P = 1$
	  c) Q is prime using a Miller-Rabin test with 50 rounds
	  5. P is prime using a Miller-Rabin test with 50 rounds

Additional tests are executed for invalid public keys failing the key checks. These tests are executed with the following constraints:

- Number of test cases: 7
- Source: NIST CAVP (NIST CAVS file 20.1)
- P: 2,048 bits
- Q: 224 bits
- G: 2,045 bits
- InvalidKey: 2,043 bits – 2,047 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/dh_invalid.vec`.

Test Case No.:	KA-KEY-DH-INVALID-1
Type:	Positive Test
Description:	Load a public key and perform the key checks
Preconditions:	None
Input Values:	P = 0xa25cb1199622be09d9f473695114963cbb3b109f92df6da1b1dcab5e8511e9a117e2881f30a78f04d6a3472b8064eb6416cdfd7bb8b9891ae5b5a1f1ee1da0cace11dab3ac7a50236b22e105dbeef9e45b53e0384c45c3078acb6ee1ca983511795801da3d14fa9ed82142ec47ea25c0c0b7e86647d41e9f55955b8c469e7e298ea30d88feacf43ade05841008373605808a2f8f8910b195f174bd8af5770e7cd85380d198f4ed2a0c3a2f373436ae6ce9567846a79275765ef829abbc6171718f7746ebd167d406e2546acdea7299194a613660d5ef721cd77e7722095c4ca42b29db3d4436325b47f850af05d411c7a95ccc54555c193384a6eeebb47e6f0f Q = 0xa944d488de8c89567b602bae44478632604f8bf7cb4deb851cf6e22d G = 0x1e2b67448a1869df1ce57517dc5e797b62c5d2c832e23f954bef8bcca74489db6caed2ea496b52a52cb664a168374cb176ddc4bc0068c6eef3a746e561f8dc65195fdaf12b363e90cfffdac18ab3ffefa4b2ad1904b45dd9f6b76b477ef8816802c7bd7cb0c0ab25d378098f5625e7ff737341af63f67cbd00509efbc6470ec38c17b7878a463cebda80053f36558a308923e6b41f465385a4f24fdb303c37fb998fc1e49e3c09ce345ff7cea18e9cd1457eb93daa87dba8a31508fa5695c32ce485962eb1834144413b41ef936db71b79d6fe985c018ac396e3af25054dbbc95e56ab5d4d4b7b61a70670e789c336b46b9f7be43cf6eb0e68b40e33a55d55cc
Expected Output:	Public key fails key checks

Steps:	1. Create a DH_Public_Key object from P, Q, G 2. Check that the key object is valid 3. Check that the key is invalid by checking that at least one of the following does not hold: 1. $1 < Y < P$ 2. $G >= 2$ 3. $P >= 3$ 4. $(P - 1) \% Q = 0$ 5. $G^Q \bmod P = 1$ 6. Q is prime using a Miller-Rabin test with 50 rounds 7. P is prime using a Miller-Rabin test with 50 rounds

13.2 Elliptic Curve Diffie Hellman

The Elliptic Curve Diffie-Hellman key agreement scheme is tested with a known answer test as follows. The test is implemented in `src/tests/test_ecdh.cpp`.

Test Case No.:	KA-ECDH-1
Type:	Positive Test
Description:	Derives a shared key from the Elliptic Curve Diffie Hellman Key Agreement Scheme
Preconditions:	None
Input Values:	• Curve: The elliptic curve, e.g., secp192r1 • Secret: The key's secret value (varying length) • CounterKey: The other party's public value (varying length)
Expected Output:	• K: The derived shared secret (varying length)
Steps:	1. Create the ECDH_KA object (input *Curve*, *Secret*) 2. Input *CounterKey* into the ECDH and compare the result with the expected output value *K*

Elliptic Curve Diffie-Hellman key agreement is tested with the following constraints:

- Number of test cases: 150
- Source: NIST CAVS file 14.1
- Curve: secp192r1, secp224r1, secp256r1, secp384r1, secp521r1, frp256v1
- Secret: 190 bits – 521 bits
- CounterKey: 192 bits, 224 bits, 256 bits, 384 bits, 521 bits
- K: 192 bits, 224 bits, 256 bits, 384 bits, 521 bits

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/ecdh.vec`.

Test Case No.:	KA-ECDH-1
Type:	Positive Test
Description:	Derives a shared key from the Elliptic Curve Diffie Hellman Key Agreement Scheme
Preconditions:	None
Input Values:	Curve = secp192r1 Secret = 0xf17d3fea367b74d340851ca4270dcb24c271f445bed9d527 (192 bits) CounterKey = 0x0442ea6dd9969dd2a61fea1aac7f8e98edcc896c6e55857cc0dfbe5d7c61fac88b11811bde328e8a0d12bf01a9d204b523 (192 bits)

Expected Output:	K = 0x803d8ab2e5b6e6fca715737c3a82f7ce3c783124f6d51cd0 (192 bits)
Steps:	1. Create the ECDH_KA object (input *Curve*, *Secret*) 2. Input *CounterKey* into the ECDH and compare the result with the expected output value *K*

The following example shows an ECDH-specific KA-KEY-1 test case. The constraints for all the key-related test cases are:

- Curve: secp256r1, secp384r1, secp521r1, brainpool256r1, brainpool384r1, frp256v1

Test Case No.:	KA-KEY-ECDH-1
Type:	Positive Test
Description:	Encode and decode an ECDH key agreement public key as PEM
Preconditions:	None
Input Values:	Curve = secp256r1
Expected Output:	None
Steps:	1. Generate a random keypair on the *Curve* 2. Encode the public key as PEM-encoded string 3. Create an ECDH_Public_Key object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the public key is valid by checking that the public point is on the *Curve*

14 Public Key-based Signature Algorithms

Public Key-based Signature Algorithms are tested using (1) a known answer test that generates a signature on a test message and (2) checks that a manipulated signature does not verify. Some algorithms also contain a third test (3), a known answer test that checks that an invalid signature does not verify. Additional tests may be implemented for specific algorithms, e.g., for public key validation. All public key-based signature algorithms use test classes implemented in `src/tests/test_pubkey.cpp`.

Test Case No.:	PKSIG-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None

Input Values:	• Hash Function: The hash function used for hashing the message, e.g., SHA-1
	• Public Parameters:
	  ◦ Group: The DL group, e.g., modp/ietf/1024 or
	  ◦ Curve: The elliptic curve, e.g., secp192r1 or
	  ◦ P, Q, E: DSA/RSA parameters
	• Private Parameters: Algorithm-specific Private Key Parameters
	• Msg: The test message (varying length)
	• Padding: The padding scheme used (optional)

Expected Output:	• Signature: The expected signature (varying length depending on the algorithm)
Steps:	1. Create the PrivateKey object from *Group/Curve/P,Q,E* and *Private Parameters* 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-2
Type:	Negative Test
Description:	Manipulated signature should not verify
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters • Private Parameters: Algorithm-specific Private Key Parameters • Msg: The test message (varying length) • Padding: The padding scheme
Expected Output:	
Steps:	1. Create the PrivateKey object from *Group/Curve/P,Q,E* and

	Private Parameters 2. Sign the *Msg* with the PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

Test Case No.:	PKSIG-3
Type:	Negative Test
Description:	Invalid signature should not verify
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters • Private Parameters: Algorithm-specific Private Key Parameters • Msg: The test message (varying length) • Padding: The padding scheme • InvalidSignature: The invalid signature
Expected Output:	
Steps:	1. Create the PrivateKey object from *Group/Curve/P,Q,E* and *Private Parameters* 2. Check that the signature *InvalidSignature* does not verify

Additionally, for each algorithm unit tests make sure that encoding and decoding private and public keys works correctly. These test cases are described here in the following.

Test Case No.:	PKSIG-KEY-1
Type:	Positive Test
Description:	Encode and decode a public key as PEM
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group*/*Curve/RSA parameters* 2. Encode the public key as PEM-encoded string 3. Create a PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair

	6. Check that the key is valid[3]

Test Case No.:	PKSIG-KEY-2
Type:	Positive Test
Description:	Encode and decode a public key as BER
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group/Curve/RSA parameters* 2. Encode the public key as BER-encoded byte array 3. Create a PublicKey object from the BER-encoded byte array, decoding the BER-encoded key 4. Check that the key object is valid[1] 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid (see PKSIG-KEY-1)

Test Case No.:	PKSIG-KEY-3
Type:	Positive Test
Description:	Encode and decode a private key as PEM
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group/Curve/RSA parameters* 2. Encode the private key as PEM-encoded string 3. Create a PrivateKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid (see PKSIG-KEY-1)

Test Case No.:	PKSIG-KEY-4
Type:	Positive Test
Description:	Encode and decode a private key as BER

3 The exact mechanism depends on the key type and is explained in the corresponding public key signature scheme section

Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None
Steps:	1. Generate a random keypair on the *Group/Curve/RSA parameters* 2. Encode the private key as BER-encoded byte array 3. Create a PrivateKey object from the BER-encoded byte array, decoding the BER-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid (see PKSIG-KEY-1)

Test Case No.:	PKSIG-KEY-5
Type:	Positive Test
Description:	Encode and decode a private key as PEM, protected with a password
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None
Steps:	1. Generate a random password string of length between 1-32 characters 2. Generate a random keypair on the *Group/Curve/RSA parameters* 3. Encode the private key as PEM-encoded string, protected with the password 4. Create a PrivateKey object from the PEM-encoded string, decoding the PEM-encoded key 5. Check that the key object is valid 6. Check that the key object algorithm name equals that of the generated keypair 7. Check that the key is valid (see PKSIG-KEY-1)

Test Case No.:	PKSIG-KEY-6
Type:	Positive Test
Description:	Encode and decode a private key as BER, protected with a password
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters
Expected Output:	None

Steps:	1. Generate a random password string of length between 1-32 characters 2. Generate a random keypair on the *Group/Curve/RSA parameters* 3. Encode the private key as BER-encoded byte array, protected with the password 4. Create a PrivateKey object from the BER-encoded byte array, decoding the BER-encoded key 5. Check that the key object is valid (see PKSIG-KEY-1) 6. Check that the key object algorithm name equals that of the generated keypair 7. Check that the key is valid (see PKSIG-KEY-1)

14.1 DSA

The Digital Signature Algorithm (DSA) is tested with the following constraints:

- Number of test cases: 304
- Source: NIST CAVP (NIST CAVS file 11.2)
- Hash Function: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
- Group (P, Q, G): 1024 bits, 2048 bits, 3072 bits
- Msg: 1024 bits
- Signature: 1024 bits, 2048 bits, 3072 bits

All the tests are implemented in `src/tests/test_dsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/dsa_prob.vec`.

Test Case No.:	PKSIG-DSA-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	P = 0xa8f9cd201e5e35d892f85f80e4db2599a5676a3b1d4f190330ed3256b26d0e80a0e49a8fffaaad2a24f472d2573241d4d6d6c7480c80b4c67bb4479c15ada7ea8424d2502fa01472e760241713dab025ae1b02e1703a1435f62ddf4ee4c1b664066eb22f2e3bf28bb70a2a76e4fd5ebe2d1229681b5b06439ac9c7e9d8bde283 Q = 0xf85f0f83ac4df7ea0cdf8f469bfeeaea14156495 G = 0x2b3152ff6c62f14622b8f48e59f8af46883b38e79b8c74deeae9df131f8b856e3ad6c8455dab87cc0da8ac973417ce4f7878557d6cdf40b35b4a0ca3eb310c6a95d68ce284ad4e25ea28591611ee08b8444bd64b25f3f7c572410ddfb39cc728b9c936f85f419129869929cdb909a6a3a99bbe089216368171bd0ba81de4fe33 Private Parameters: X= 0xc53eae6d45323164c7d07af5715703744a63fc3a Msg = empty message Nonce = 0x98cbcc4969d845e2461b5f66383dd503712bbcfa
Expected Output:	Signature = 0x50ed0e810e3f1c7cb6ac62332058448bd8b284c0c6aded17216b46b7e4b6f2a97c1ad7cc3da83fde
Steps:	1. Create the DSA_PrivateKey object from *P, Q, G* and *X* 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the DSA_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-DSA-2
Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	P = 0xa8f9cd201e5e35d892f85f80e4db2599a5676a3b1d4f190330ed3256b26d0e80a0e49a8fffaaad2a24f472d2573241d4d6d6c7480c80b4c67bb4479c15ada7ea8424d2502fa01472e760241713dab025ae1b02e1703a1435f62ddf4ee4c1b664066eb22f2e3bf28bb70a2a76e4fd5ebe2d1229681b5b06439ac9c7e9d8bde283 Q = 0x0xf85f0f83ac4df7ea0cdf8f469bfeeaea14156495 G = 0x2b3152ff6c62f14622b8f48e59f8af46883b38e79b8c74deeae9df131f8b856e3ad6c8455dab87cc0da8ac973417ce4f7878557d6cdf40b35b4a0ca3eb310c6a95d68ce284ad4e25ea28591611ee08b8444bd64b25f3f7c572410ddfb39cc728b9c936f85f419129869929cdb909a6a3a99bbe089216368171bd0ba81de4fe33 Private Parameters: X= 0xc53eae6d45323164c7d07af5715703744a63fc3a Msg = empty message Nonce = 0x98cbcc4969d845e2461b5f66383dd503712bbcfa
Expected Output:	Signatures do not verify
Steps:	1. Create the DSA_PrivateKey object from *P*, *Q*, *G* and *X* 2. Sign the *Msg* with the DSA_PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

The following example shows a DSA-specific PKSIG-KEY-1 test case. The constraints for this test case are:

- Group: dsa/jce/1024, dsa/botan/2048

Test Case No.:	PKSIG-KEY-DSA-1
Type:	Positive Test
Description:	Encode and decode a DSA public key as PEM
Preconditions:	None
Input Values:	Group = dsa/jce/1024
Expected Output:	None
Steps:	1. Generate a random keypair on the DSA *Group* 2. Encode the public key as PEM-encoded string 3. Create a DSA_PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid

	5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the key is valid by checking that: 1. $1 < Y < P$ 2. $G >= 2$ 3. $P >= 3$ 4. If Q is given: a) $(P - 1) \% Q = 0$ b) $G^Q \bmod P = 1$ c) Q is prime using a Miller-Rabin test with 50 rounds 5. P is prime using a Miller-Rabin test with 50 rounds

14.2 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is tested with the following constraints:

- Number of test cases: 183
- Source: NIST CAVP (NIST CAVS file 11.2)
- Hash Function: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
- Curve: secp224r1, secp256r1, secp384r1
- Msg: 1024 bits
- Signature: 448 bits, 512 bits, 568 bits

All the tests are implemented in `src/tests/test_ecdsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/ecdsa_prob.vec`.

Test Case No.:	PKSIG-ECDSA-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x16797b5c0c7ed5461e2ff1b88e6eafa03c0f46bf072000dfc830d615 Msg = 0x699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a5669201912064c64e700c139248cdc19495df081c3fc60245b9f25fc9e301b845b3d703a694986e4641ae3c7e5a19e6d6edbf1d61e535f49a8fad5f4ac26397cfec682f161a5fcd32c5e780668b0181a91955157635536a22367308036e2070f544ad4fff3d5122c76fad5d Nonce = 0xd9a5a7328117f48b4b8dd8c17dae722e756b3ff64bd29a527137eec0
Expected Output:	Signature = 0x2fc2cff8cdd4866b1d74e45b07d333af46b7af0888049d0fdbc7b0d68d9cc4c8ea93e0fd9d6431b9a1fd99b88f281793396321b11dac41eb
Steps:	1. Create the ECDSA_PrivateKey object from *Curve*, *X* 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the ECDSA_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-ECDSA-2
Type:	Negative Test
Description:	Invalid signatures should not verify

Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x16797b5c0c7ed5461e2ff1b88e6eafa03c0f46bf072000dfc830d615 Msg = 0x699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a5669201912064c64e700c139248cdc19495df081c3fc60245b9f25fc9e301b845b3d703a694986e4641ae3c7e5a19e6d6edbf1d61e535f49a8fad5f4ac26397cfec682f161a5fcd32c5e780668b0181a91955157635536a22367308036e2070f544ad4fff3d5122c76fad5d Nonce = 0xd9a5a7328117f48b4b8dd8c17dae722e756b3ff64bd29a527137eec0
Expected Output:	Signatures do not verify
Steps:	1. Create the ECDSA_PrivateKey object from *Curve*, *X* 2. Sign the *Msg* with the ECDSA_PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

The following example shows an ECDSA-specific PKSIG-KEY-1 test case. The constraints for this test case are:

- Curve: secp256r1, secp384r1, secp521r1

Test Case No.:	PKSIG-KEY-ECDSA-1
Type:	Positive Test
Description:	Encode and decode an ECDSA public key as PEM
Preconditions:	None
Input Values:	Curve = secp256r1
Expected Output:	None
Steps:	1. Generate a random keypair on the ECDSA *Curve* 2. Encode the public key as PEM-encoded string 3. Create a ECDSA_PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the public key is valid by performing the checks from AIS 46

Additional tests check that public keys are validated correctly. Test vectors are taken from NIST CAVS file 11.0 for FIPS 186-2 and FIPS 186-4.

Test Case No.:	PKSIG-PUBKEY-VAL-ECDSA-1
Type:	Negative Test
Description:	Validate an ECDSA public key
Preconditions:	None
Input Values:	Curve = secp256r1 InvalidKeyX = 0xd2b419e62dc101b395401208b9868a3b3fd007ad92adb18921c068d416aa22e7 (256 bits) InvalidKeyY = 0x17952007e021b46a2ab12f14115aafb70608a37f0c3366e7e3921414b904d395a (256 bits)
Expected Output:	None
Steps:	1. Generate a random keypair on the ECDSA *Curve* 2. Encode the public key as PEM-encoded string 3. Create a ECDSA_PublicKey object on the curve *Curve* with the public point x coordinate InvalidKeyX and the y coordinate InvalidKeyY 4. Check that the public key is valid by performing the checks from AIS 46

14.3 ECGDSA

The Elliptic Curve German Digital Signature Algorithm (ECGDSA) is tested with the following constraints:

- Number of test cases: 9
- Source: “The Digital Signature Scheme ECGDSA”, Erwin Hess, Marcus Schafheutle, and Pascale Serf, Siemens AG, October 24, 2006
- Hash Function: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
- Curve: brainpool192r1, brainpool256r1, brainpool320r1, brainpool384r1, brainpool512r1
- Msg: 368 bits, 384 bits, 408 bits
- Signature: 384 bits, 512 bits, 640 bits, 768 bits, 1024 bits

All the tests are implemented in `src/tests/test_ecgdsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/ecgdsa.vec`.

Test Case No.:	PKSIG-ECGDSA-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x16797b5c0c7ed5461e2ff1b88e6eafa03c0f46bf072000dfc830d615 Msg = 0x699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a5669201912064c64e700c139248cdc19495df081c3fc60245b9f25fc9e301b845b3d703a694986e4641ae3c7e5a19e6d6edbf1d61e535f49a8fad5f4ac26397cfec682f161a5fcd32c5e780668b0181a91955157635536a22367308036e2070f544ad4fff3d5122c76fad5d Nonce = 0xd9a5a7328117f48b4b8dd8c17dae722e756b3ff64bd29a527137eec0
Expected Output:	Signature = 0x2fc2cff8cdd4866b1d74e45b07d333af46b7af0888049d0fdbc7b0d68d9cc4c8ea93e0fd9d6431b9a1fd99b88f281793396321b11dac41eb
Steps:	1. Create the ECGDSA_PrivateKey object from *Curve, X* 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the ECGDSA_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-ECGDSA-2

Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x16797b5c0c7ed5461e2ff1b88e6eafa03c0f46bf072000dfc830d615 Msg = 0x699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a5669201912064c64e700c139248cdc19495df081c3fc60245b9f25fc9e301b845b3d703a694986e4641ae3c7e5a19e6d6edbf1d61e535f49a8fad5f4ac26397cfec682f161a5fcd32c5e780668b0181a91955157635536a22367308036e2070f544ad4fff3d5122c76fad5d Nonce = 0xd9a5a7328117f48b4b8dd8c17dae722e756b3ff64bd29a527137eec0
Expected Output:	Signatures do not verify
Steps:	1. Create the ECGDSA_PrivateKey object from *Curve*, *X* 2. Sign the *Msg* with the ECGDSA_PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

The following example shows an ECGDSA-specific PKSIG-KEY-1 test case. The constraints for this test case are:

- Curve: secp256r1, secp384r1, secp521r1

Test Case No.:	PKSIG-KEY-ECGDSA-1
Type:	Positive Test
Description:	Encode and decode an ECGDSA public key as PEM
Preconditions:	None
Input Values:	Curve = secp256r1
Expected Output:	None
Steps:	1. Generate a random keypair on the ECGDSA *Curve* 2. Encode the public key as PEM-encoded string 3. Create a ECGDSA_PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the public key is valid by performing the checks from AIS 46

14.4 ECKCDSA

The Elliptic Curve Korean Certificate Digital Signature Algorithm (ECKCDSA) is tested with the following constraints:

- Number of test cases: 3
- Source: TTAK.KO-12.0015/R2 "Digital Signature Mechanism with Appendix - Part 3: Korean Certificate-based Digitial Signature Algorithm using Elliptic Curves (EC-KCDSA)"
- Hash Function: SHA-1, SHA-224, SHA-256
- Curve: secp192r1, secp224r1, secp256r1
- Msg: 24 bits, 512 bits,
- Signature: 352 bits, 448 bits, 512 bits

All the tests are implemented in `src/tests/test_eckcdsa.cpp`. The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/pubkey/eckcdsa.vec`.

Test Case No.:	PKSIG-ECKCDSA-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x9051A275AA4D98439EDDED13FA1C6CBBCCE775D8CC9433DEE69C59848B3594DF Msg = 0x546869732069732612073616D706C65206D65737361676520666F722045432D4B4344534120696D706C656D656E746174696F6E2076616C69646174696F6E2E Nonce = 0x76A0AFC18646D1B620A079FB223865A7BCB447F3C03A35D878EA4CDA
Expected Output:	Signature = 0xEEA58C91E0CDCEB5799B00D2412D928FDD23122A1C2BDF43C2F8DAFAAEBAB53C7A44A8B22F35FDB9DE265F23B89F65A69A8B7BD4061911A6
Steps:	1. Create the ECKCDSA_PrivateKey object from *Curve*, *X* 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the ECKCDSA_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-ECKCDSA-2
Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	Hash Function = SHA-224 Curve = secp224r1 Private Parameters: X= 0x9051A275AA4D98439EDDED13FA1C6CBBCCE775D8CC9433DEE69C59848B3594DF Msg = 0x546869732069732612073616D706C65206D65737361676520666F722045432D4B4344534120696D706C656D656E746174696F6E2076616C69646174696F6E2E Nonce = 0x76A0AFC18646D1B620A079FB223865A7BCB447F3C03A35D878EA4CDA
Expected Output:	Signatures do not verify
Steps:	1. Create the ECKCDSA_PrivateKey object from *Curve*, *X* 2. Sign the *Msg* with the ECKCDSA_PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

The following example shows an ECKCDSA-specific PKSIG-KEY-1 test case. The constraints for this test case are:

- Curve: secp256r1, secp384r1, secp521r1

Test Case No.:	PKSIG-KEY-ECDSA-1
Type:	Positive Test
Description:	Encode and decode an ECKCDSA public key as PEM
Preconditions:	None
Input Values:	Curve = secp256r1
Expected Output:	None
Steps:	1. Generate a random keypair on the ECDSA *Curve* 2. Encode the public key as PEM-encoded string 3. Create a ECKCDSA_PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the public key is valid by performing the checks from AIS 46

14.5 RSA

The RSA algorithm is tested with the following constraints:

- Number of test cases: 77
- Source: ISO 9796-2:2010, Project Wycheproof, others
- Hash Function: SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
- E: 3, 5, 7, 17, 79, 28609, 29115, 65537
- P: 192 bits, 256 bits, 384 bits, 512 bits, 768 bits, 1024 bits, 1536 bits, 2048 bits
- Q: 192 bits, 256 bits, 384 bits, 512 bits, 768 bits, 1024 bits, 1536 bits, 2048 bits
- Msg: 0 bits – 1864 bits
- Padding: EMSA1(SHA-1), EMSA2(SHA-1), EMSA2(SHA-224), EMSA2(SHA-256), EMSA2(SHA-384), EMSA2(SHA-512), EMSA3(Raw), EMSA3(SHA-1), EMSA3(SHA-224), EMSA3(SHA-256), EMSA3(SHA-384), EMSA3(SHA-512), EMSA4(SHA-1), EMSA4(SHA-1), ISO 9796-2 DS2(SHA-1), ISO 9797-2 DS3(SHA-1)
- Signature: 384 bits – 2048 bits

All the tests are implemented in `src/tests/test_rsa.cpp`. The following table shows an example test case with one test vector. Test vectors for test cases PKSIG-RSA-1 and PKSIG-RSA-2 are listed in `src/tests/data/pubkey/rsa_sig.vec`. Test vectors for test case PKSIG-RSA-3 are listed in `src/tests/data/pubkey/rsa_invalid.vec`.

Test Case No.:	PKSIG-RSA-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	Hash Function = SHA-1 E = 5 P = 293259716013945534358765451778610158671593705962025657480327152248550535748883352950641185952331578788506447464760535308882211829358443758608966414537457415767576889158645019 Q = 363407261169858107495845562737495903466588000383866197681553088822118293584437586089664145374574157675768891586450195 Msg = 0x4161436445664768496A4B
Expected Output:	Signature = 0x3A3B7502D85F05128CFB7460820503133975 3DA50D0DB7E268C3951F04A1981EDE22613BFC38DB9FFEBE183A4F11B0B0F8D7BEB668F7C1C385A801C2DDD7C08CB2E56082F80AD1105E930ED96DB6A0309639A51F5379B682C7F75C601BD4ADE5
Steps:	1. Create the RSA_PrivateKey object from *P, Q, E* 2. Verify the signature *Signature* on the *Msg*

	3. Sign the *Msg* with the RSA_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

Test Case No.:	PKSIG-RSA-2
Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	Hash Function = SHA-1 E = 5 P = 29325971601394553435876545177861015867159370596202565748032715224855053574888335295064118595233157878850644746476053 Q = 36340726116985810749584556273749590346658800038386619768155308882211829358443758608966414537457415767576889158645019 Msg = 0x416143644566476849 6A4B
Expected Output:	Signatures do not verify
Steps:	1. Create the RSA_PrivateKey object from *P, Q, E* 2. Sign the *Msg* with the RSA_PrivateKey object 3. Check that a signature with all zeros (of the length of that of the generated signature) does not verify? 4. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 5. Check that this modified signature does not verify

Test Case No.:	PKSIG-3
Type:	Negative Test
Description:	Invalid signature should not verify
Preconditions:	None
Input Values:	• Group: The DL group, e.g., modp/ietf/1024 or • Curve: The elliptic curve, e.g., secp192r1 or • P, Q, E: RSA parameters • Private Parameters: Algorithm-specific Private Key Parameters • Msg: The test message (varying length) • Padding: The padding scheme • InvalidSignature: The invalid signature
Expected Output:	
Steps:	1. Create the RSA_PrivateKey object from *P, Q, E* 2. Check that the signature *InvalidSignature* does not verify

The following example shows an RSA-specific PKSIG-KEY-1 test case. The constraints for this test case are:

- Key Length: 1024 bits, 1280 bits

Test Case No.:	PKSIG-KEY-RSA-1
Type:	Positive Test
Description:	Encode and decode an RSA public key as PEM
Preconditions:	None
Input Values:	Key Length = 1024 bits
Expected Output:	None
Steps:	1. Generate a random *Key Length* bits RSA keypair with E = 65537 2. Encode the public key as PEM-encoded string 3. Create a RSA_PublicKey object from the PEM-encoded string, decoding the PEM-encoded key 4. Check that the key object is valid 5. Check that the key object algorithm name equals that of the generated keypair 6. Check that the public key is valid by checking that: 1. N >= 35 2. N is uneven 3. E >= 2

14.6 Extended Hash-Based Signatures (XMSS)

14.6.1 Signature Generation

The XMSS signature generation algorithm [XMSS] is tested with the following constraints:

- Hash Function: SHA-256, SHA-512
- w: 16
- h: 10
- Msg: 0 bits – 400 bits
- Signature: 20032 bits - 72768 bits

All the tests are implemented in `src/tests/test_xmss.cpp`. All test vectors are listed in `src/tests/data/pubkey/xmss_sig.vec`. Currently 4 test vectors are tested. Optional additional test vectors are present within `xmss_sig.vec` but commented out by default to reduce the test bench run time. The hash function and algorithm parameters “w”, “h” are provided through the algorithm oid, which is part of the private key. The following table shows an example test case with one test vector.

Test Case No.:	PKSIG-XMSS-1
Type:	Positive Test
Description:	Sign a test message
Preconditions:	None
Input Values:	Hash Function = SHA-256 h = 10 PrivateKey = 0x01000001A020196CDE3A20C13477CE56DE3A7A4381821EA50BF07F0670048A0E1736D22876575FA4F5404B393828F74776A9B9C73B0962069652B088432242E12CF75E170000000000000000CE1994BC37AEDD7E21851001EC0F4296ECC3D389263E4E720D05EFFD60A20A41B90B7E2CC1647319B4B143CEDDADADFB3E571BE68F36ACC8D6C0A0ADD41266F2 Msg = 0x078A87923DEC59CE843149F5E642A3F921E2E78543132F88BA637A09DF0C16552A3037E3EEB3A30FDA5DF73AE2E0DD3821D1
Expected Output:	Signature = 0x0000000000000000D3A842202DB1812F8DC93387EA6A78D01211D00911D37678CAD55CBC228B2DA495C0B88593D505696EF3BE99A6742B75A12555BBEDE5F788D4F4B7DAE4E6C7DA82FAA2D7E60F836673BC0BAE8CB75A6A94480970C90A412E49AE7B0CFA63025C1444A746C5BDCF9D8618CECE33549043A98D05CBA7673FB7E4F835E624B482E85B3B2AFF7613CD58F1C8FF2B0E6011E02F5A3387708E8E99970EB0288AFED53454DF7804B583DE58BB7B93041D6C395360D84A9B4C9744395C1F2B05ACB932C7AA8279B6012E06347559A89E945BB140119D074C19AE0608CF10D622ED32234D3F739B2A8520288BD1AFF00EFC87B14F294837644EEF5AFE6C938229E9EF39

	C35032E57E36FDF82CA625F4F78E796B6E19EA0825333BF9BAB8B 280BCA94B6858CDE824DB884F808F3DDC450290C3441EA4D3243 0FA701FFF0D7E51C1AB829C1A67FBBC0776CC47E288E7BA53934 9741F9EBFF591F40F47180A4438C998A73EDD087E57325B4E308C2 E1EA8097C9718F3547A0876789D0A808E1941FA5CDE1523934B08 D014EA974FC867EFEC161CA1591F29B4E34276FE045FBB8AA1FB B69A8693C438D47903B63CB6C9D15988C5025B0D84E1BCDEEEF4 66F4B30373EEDDDB216BF1AC20E068DB89A201706CD1F0B97819 44889A71CA92F8B9A86D086ED63EB71B6412F0672A549268418B1 7F408723FACC10A640D3977756F41B2934D3A76A64FE1FFB29456 E9634D7B839E3B66A744EF0D4BEEF472A2817C5E0F10A91119371 DF30DE7DC394219C95CCAD24116991D53CD5B2059F48D6FFAC08 A5869C94866D7932CA97760D55AE5AF8A978F91A934E21922F0B9 BE5227BB9E557D20F6D139D71160F29FBE23C0757E4E4EAE8524A 38B00043DADE86AAAE2F3BCFF6E65F74410863B7F5E585443039C FA12DC17243049A4F9CC6B68AC0874A62F821519F027D5B658C3E 8CB724FE469001C6DE151A407CC48CB966FD5DF819AEA78F3C1B 291A453490EAD24767A97506D635029D5A1182F05372E5A7CA3B7 440446733B500F26E3137EF5E01F145989EBBCE7E72681CF3213B59 9B3D44E73B8FB2B06B63A12D59BC33820AF834C9B48E1FFDA087 27A1A0D092900C91F3DF1C2F74264271DDE4546CDDF687E5A3D7 170DB4F7DB3FF913D96F3EB45F5CACD5400128838625E3C8F852F 2B1071D53A6F73FDC7FC67A9FB007519250BF50D3CEEC30184CB E4B4348B9D04EDA3CCA4DE611AA9D9E21179F7057D5229FCB8C 7599594FDD17B0DAD86DF88B1D92F29218F7AEC8478980B58256 D0B3F22A0A738F2A45BD1845AC44E20F18F5619D828E4874E430C 0C5D36EB80F9DDD1766782C4E0EADF20C971941999CF3365E2802 2D13DDEC97D9F6C7AE8E04A6E2B50711A6087EF607A14ED1A12 45CDD07BEBF086F65CA32CD258D4B9B9F93E914A1C493F6D9EC A4470EF6655139E3B15D41486FD80E755379442827CE2F73BD1471 523E0103AC6564E185F5F81DAD524BFA91A311D919B86B4585A17 1AE240191EAA78320D44B4062BB1BD13F807DB23FC2F9849CFDE B6E1023E234A07E88318F1AF60A75E7F167BE568ED0EBDA558B5 007A85C3154B6DAB837FC9FD3015D4B262502B4518F16621E945D 7FE2B5591326E4D94AF0F3EF7C289027C8ACA22C9AB658017FDF 31610B8994C042C501C25F3C84D609D1A4A6C122FEE6B63F735E2 8DCFE66640AA98C9B88594A8DAECD724BF1BBAF847764347216F FAAE0AC41EEBA945DC74AF17C463DCFAC75279899DEEBE762F8 F0858B2FA2A4E0C5C2DE0D20658F0321AB6ECB6DB62C67EEBE9 D3D04D53A987639A70F142250CC1565301F809E35DECDCAF34480 F139A07781F6B39D4DABE11DAFD7C4BAED73AE540394CF223CE D39092382C26CD968E21A97BB374B466DC34C7B25A93B876B7667 28F385145D610A3E793D005C88BBC697090523B280B382255762205 71E1E7615656583ECEC3677627A6A1E298BF4377DC9196F6659AD 6F3731D6AB1A7C8E6BFA4CF50550955EBDD47F0E42BE07EF4A78 8669ACB8F403BE85EAB1789000184BAD8C2DA4011C3FE77ADAC 3BFCCABD892BC2A4A08969BD0D01620CBF2A8664D656819CBE3 0C8D4F71EFE0DD9185B9705E82846466DA99D06184FB6B8023AE3 1CC2F1B3E9A967F787645204AA414DF00B7AE9026FB9E28BA8479 EE2B46DFBBB39CA86B5C360C9F5C512E33188ED2770CB8B03959 288BED59011D63534F9094DEF769F85E9328FC11522FCBCCA648C CAB654850C34F245F157349BA460F621FEE2BFE0B4E6D89E88AB6

	845E9BB0B6056E653FAC558EDA8BDD5A3E8C44649CDE9B5389E DF7C10A2114CA7C6DD2DABEE1F4D9A695303A79B4F72C27FC82 AF6F0C26E9551EF3A489E4F5E2CD9647B75BA75413C41B61121FD 7411099FD5EC5FBD87D2B4998AF3E484B4B2909A881CF9989F89A E190704DAD08DDADFD6687CC273E56A3B13395A942DEA8FE26E 3D7EC8B0880DC8C51FC850354AACB05BD175542080D0C87CEA9 9081ADF901920EA6327B761DEA28B61951EAEC23BC9DC30D32D D0ED4FCFE39F575803F874D72D71D48CE8F26D47B0CC74881C54 F80F41DB4718EC04FAAADFD93AF8B8A258527024658FB28D4F69 83DAA01558F85BF8C6120D355388C302516D1FDA5480961799AC8 B5E9B485BC579675F03CE604A103DF21CD31ADD951AD0A3AE1A D1788444997EB12F78BA96E909C74543EB6D0DCAFAE60796632E6 888E3B3D2EB6D6B733AA53C455C04473C2213494570F6C8AE04FE F4307419A7D84C87EF8A9CA8DC62177D2BC09FB1362ECF7A6E87 9B51B0B27B5358356689289D09BAEC2F204ADBA0A20C05A5E7C5 9F10D4C9F0C349ED71B2D08CFAFC96CB97DE01FBC0484B2A05E 93FF0FFC2C7BA974933E10AEFAFBF440C75CDA179B6DC09AFE8 1AE36080510621E77D526D677749B50250CD83EBA1C7D9F8B594D 711402B10430A22FB83FAA2372C1D6C88787BF82007BA5FAE0F3F 17836F9B2D9311366E3506395F9A0AB17731F6F792C3FB7127BBCB CAD9AB39A6E59CB7F0A2EE36E66644D41B84F5DC57D27A69CB D9BEB840E5D646E4D13AF0286E7D31D9CE93FA896889BAC3F124 DFA696AF3D60737F71FDEB9C09F3D0FEAA1FA698291A74749193 88B7C014F022D239DCCC6C760180BB34078DE1F7BAB07C46D7D9 244CBA43C3BBC2C4753868887C129CFDF2857A8E07D18EF99309 B85FB08980F4258806D7A618502E3D9C2DE7F33C3E267A53B7AC B084797C9F346B33A04E32716FD0E85B13EF7796BDAA3DC46ED5 8A93AB61A516990C04BE612F1C7E341C6267CE8B9326EFD200B01 5D2F0C50B8D9CC0217F758659DF95D86F2D3372CDBBCFE0A0E1 A8719A46E041B9BED9D194B98DD74F3308A7F9C0A068BF554861 083DAC5B6A594FE83F9111547737FB2D4D712E3AA1BAB6820FFF 37175B28F1A81B06619C99B5A89B0F2C155174C8137A63B49CE948 01F0500E0EA53F26313E3DD203B74D409DCB46C2D4CC0C08DCF EEFB89AC66D19C95A14FEB1BEE4A646FD911B
Steps:	1. Create the XMSS_PrivateKey object from the byte sequence provided through input value “PrivateKey” 2. Verify the signature *Signature* on the *Msg* 3. Sign the *Msg* with the XMSS_PrivateKey object and compare with the expected output *Signature* 4. Verify the generated signature on the *Msg*

14.6.2 Signature Verification

The XMSS signature verification is tested with the following constraints:

- Hash Function: SHA-256, SHA-512
- w: 16
- h: 10, 16, 20
- Msg: 0 bits – 2640 bits

- Signature: 20032 bits - 77888 bits

The hash function and algorithm parameters “w”, “h” are provided through the algorithm oid, which is part of the private key. All test vectors are listed in `src/tests/data/pubkey/xmss_verify.vec`, the following table shows an example test case with one test vector.

Test Case No.:	PKSIG-XMSS-2
Type:	Negative Test
Description:	Invalid signatures should not verify
Preconditions:	None
Input Values:	Hash Function = SHA-512 h=10 PublicKey = 0x04000004E0489566FE62275CF1BE38B809F0F959717848A76D26B2392793BC6523FC57AA78B3EBBEB74462990EAF2E2FB89F988B804EF9A3155641347124F7728040C1EF60BF55B84746D9B9232F0221A3EF11728BF25E797985607C06432EA5B4122574923583E7127424B4304D01F90DE74E2C81ACA71E6721805B70E9C77FA19C5C0F Msg = 0x426E562AB69A03A893F56910A2AED2A0618DA1E365167749E78BEB4997D36DC054F34225797478A5153037D4154A90C88836EAB69A7F6783237143FDEDBDB6FBA8AEDFD98D3AF16FA293660640163C0936AE072C0D38772013B0BBF97CF44B64C44ACB62803A7B2B374DA627E47A1135782F09537E873AAF5BB54676BB5195AADDF73B64FB9B32
Expected Output:	Signatures do not verify
Steps:	1. Create the XMSS_PublicKey object from the byte sequence provided through input value “Public” 2. Check that a signature with all zeros (of the length of that of the generated signature) does not verify 3. Create a modified version of the generated signature by changing the length of it or by flipping random bits in it 4. Check that this modified signature does not verify

15 Random Number Generators

Random number generators (RNGs) are tested using positive tests which compare the resulting output of the seeded random number generator with the data from test vectors (`hmac_drbg`). In addition to these tests, a unit test for HMAC-DRBG defines positive and negative tests which validates the correctness of the HMAC-DRBG random number generator (`hmac_drbg_unit`).

All the tests are implemented in `src/tests/test_rng.cpp`.

15.1.1 HMAC-DRBG

HMAC-DRBG RNG is tested with the following constraints:

- Number of test cases: 3360
- Source: NIST CAVP (NIST CAVS file 14.3)
- EntropyInput: initial entropy input
- EntropyInputReseed: entropy input used to reseed the RNG
- AdditionalInput1: optional randomization input
- AdditionalInput2: optional randomization input
- Out: RNG output (80-256 bytes)

The tests are executed for HMAC-DRBG with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, and SHA-512-256.

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/hmac_drbg.vec`.

Test Case No.:	RNG-HMAC-DRBG-1
Type:	Positive Test
Description:	A known answer test that checks the correct RNG output
Preconditions:	None
Input Values:	EntropyInput = 0x29C62AFA3C52208A3FDECB43FA613F156C9EB59AC3C2D48B EntropyInputReseed = 0xBD87BE99D184165412314140D4027141433DDAF259D14BCF897630CCAA27338C AdditionalInput1 = 0x141146D404F284C2D02B6A10156E3382 AdditionalInput2 = 0xEDC343DBFFE71AB4114AC3639D445B65
Expected Output:	Out = 0x8C730F0526694D5A9A45DBAB057A1975357D65AFD3EFF303320BD14061F9AD38759102B6C60116F6DB7A6E8E7AB94C05500B4D1E357DF8E957AC8937B05FB3D080A0F90674D44DE1BD6F94D295C4519D
Steps:	1. Create an HMAC_DRBG object 2. Seed the RNG with *EntropyInput* 3. Reseed the RNG with *EntropyInputReseed* 4. Add additional randomization input *AdditionalInput1* and *AdditionalInput2* 5. Compare the result with the output value *Out*

15.1.2 Unit Test for HMAC-DRBG

The unit tests for HMAC-DRBG (**`hmac_drbg_unit`**) extend the **`hmac_drbg`** test suite with negative tests. The following additional properties of HMAC-DRBG are tested:

- test_reseed: Tests the reseed interval.
- test_broken_entropy_input: Tests whether the RNG throws exceptions if it is provided with insufficient entropy.
- test_check_nonce: Tests whether the nonce provided to the RNG has at least one half of the security bit strength. Otherwise, the RNG has to throw an exception (for HMAC-SHA-256, the nonce has to be at least 16 bytes long).
- test_prediction_resistance: Tests with a reseed interval set to 1.
- test_fork_safety: Tests whether a forked process has a different RNG output than its parent process.
- test_randomize_with_ts_input: Tests the function `randomize_with_ts_input`.

16 AutoSeeded_RNG

The AutoSeeded_RNG random number generator is tested using a unit test for initialization, seeding and reseeding.

Test Case No.:	RNG-AUTO-RNG-1
Type:	Positive Test
Description:	A unit test that makes sure initialization, seeding and reseeding work correctly
Preconditions:	None
Input Values:	None
Expected Output:	None
Steps:	1. Create an AutoSeeded_RNG object with an empty set of entropy sources and check that it throws a PRNG_Unseeded exception 2. Create an AutoSeeded_RNG object with a Null_RNG as the entropy source and check that it throws a PRNG_Unseeded exception 3. Create an AutoSeeded_RNG object with a an empty set of entropy sources and a Null_RNG as the entropy source and check that it throws a PRNG_Unseeded exception 4. Create an AutoSeeded_RNG object with the default constructor 5. Check that the name is HMAC_DRBG plus the HMAC specified in BOTAN_AUTO_RNG_HMAC 6. Check that the AutoSeeded_RNG is seeded 7. Extract 16 random bytes from the AutoSeeded_RNG 8. Reset the AutoSeeded_RNG 9. Check that the AutoSeeded_RNG is not seeded 10. Extract 16 random bytes from the AutoSeeded_RNG, forcing an automatic reseed 11. Check that the AutoSeeded_RNG is seeded 12. Check that the AutoSeeded_RNG is seeded 13. Extract 16 random bytes from the AutoSeeded_RNG 14. Reset the AutoSeeded_RNG 15. Check that the AutoSeeded_RNG is not seeded 16. Attempt to reseed the AutoSeeded_RNG with 256 bits from an empty set of entropy sources and check that the returned entropy estimation is zero 17. Check that the AutoSeeded_RNG is not seeded 18. Extract 16 random bytes from the AutoSeeded_RNG, forcing an automatic reseed 19. Check that the AutoSeeded_RNG is seeded

17 TLS Protocol Execution

TLS client and server are tested with positive tests by performing TLS handshakes. In these tests basic credentials with TLS certificates and TLS policy are first created. Afterwards, the client and the server attempt to execute a TLS handshake with a specific TLS/DTLS protocol version, key exchange method, and cipher algorithm.

The test is implemented in `src/tests/unit_tls.cpp`.

The following TLS handshake tests are executed:

- TLS handshake with the following cipher suites, each once with and once without Encrypt-then-MAC (for TLS 1.0, TLS 1.1, TLS 1.2, DTLS 1.0, DTLS 1.2):
  - RSA_WITH_AES_128_CBC_SHA
  - RSA_WITH_AES_128_CBC_SHA256
  - ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  - ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
  - RSA_WITH_AES_256_CBC_SHA
  - ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS handshake with the following cipher suites (for TLS 1.2, DTLS 1.2):
  - DHE_RSA_WITH_AES_128_CBC_SHA256
  - DHE_DSS_WITH_AES_128_CBC_SHA256
- TLS handshake with the `Strict_Policy`
- TLS handshake with the `NSA_Suite_B_128` policy
- TLS handshake with the following GCM cipher suites (for TLS 1.2, DTLS 1.2):
  - ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
  - ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
  - ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - DHE_DSS_WITH_AES_128_GCM_SHA256
  - DHE_DSS_WITH_AES_256_GCM_SHA384
- TLS handshake using ECC point compression with the following cipher suites (for TLS 1.2, DTLS 1.2)
  - ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS handshake using the specific curve secp521r1 with the following cipher suites (for TLS 1.2, DTLS 1.2)

18 TLS Policy Verification

TLS policy is used to validate correct cryptographic algorithms, protocol versions, or cipher suites. Many of these properties are already tested in the TLS handshake execution test described in the previous section. We extended the test suite with positive and negative tests validating correct certificate handling.

The test is implemented in `src/tests/unit_tls_policy.cpp`.

In the test different certificates with different key lengths are created and tested against the default TLS policy. Only certificates with appropriate key lengths can be accepted. Certificates with insufficient key lengths must be rejected.

In the test the following certificates are tested:

- RSA (1024 / 2048 bits)
- ECDSA (192 / 256 bits)
- DSA (1024 / 2048 bits)

19 TLS Protocol Message Parsing

TLS message parsing is tested using byte sequences resulting in valid and invalid messages. The test case is described in the following.

Test Case No.:	TLS-message-1
Type:	Positive Test
Description:	Parses a byte sequence into a TLS protocol message
Preconditions:	None
Input Values:	• Buffer: byte sequence • Protocol: TLS protocol version • Ciphersuite: cipher suite included in the protocol message • AdditionalData: additional data used in the test case, for example a TLS extension bytes • Exception: exception thrown when parsing invalid byte sequence
Expected Output:	• Out: parsed message
Steps:	1. Parse the byte sequence into a TLS protocol message 2. Check the protocol message properties or the exception that has been thrown during parsing

In the following we give examples of positive and negative tests for TLS protocol messages.

The messages were generated with OpenSSL and TLS-Attacker.

19.1 ClientHello

The ClientHello message contains several fields. The following fields are checked:

- Protocol Version
- Extensions

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/client_hello.vec`.

Test Case No.:	TLS-ClientHello-1
Type:	Positive Test
Description:	Parses a ClientHello message without any extension
Preconditions:	None
Input Values:	Buffer = 030320f3dc33f90be6509e6133a1819f2b80fe6ccc6268d9195ca4ead7504ffe7e2a0000aac030c02cc028c024c014c00a00a500a300a1009f006b006a006900680039003800370036008800870086008 5c032c02ec02ac026c00fc005009d003d00350084c02fc02bc027c023c013c00900a400a200a0009e00670040003f003e0033003200310030009a009900980097004500440043004 2c031c02dc029c025c00ec004009c003c002f00960041c011c007c00cc00200050004c012c008001600130010000dc00dc003000a00ff0100000 0 Protocol = 0303 AdditionalData = FF01 Exception =
Expected Output:	The message can be successfully parsed. By default an empty renegotiation is generated inside of the ClientHello message (0xFF01)
Steps:	1. Parse the message bytes. 2. Verify successful processing, protocol version, and the extension being generated.

Test Case No.:	TLS-ClientHello-2
Type:	Negative Test
Description:	Parses a ClientHello message with insufficient bytes
Preconditions:	None
Input Values:	Buffer = 00 Protocol = 0303 Exception = Invalid argument Decoding error: Client_Hello: Packet corrupted
Expected Output:	The message cannot be parsed and the processing results into a “Packet corrupted” exception.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

19.2 ServerHello

The ServerHello message contains several fields. The following fields are checked:

- Protocol Version
- Cipher suite
- Extensions

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/server_hello.vec`.

Test Case No.:	TLS-ServerHello-1
Type:	Positive Test
Description:	Parses a ServerHello message with session ticket, extended master secret, and renegotiation info
Preconditions:	None
Input Values:	Buffer = 03019f9cafa88664d9095f85dd64a39e5dd5c09f5a4a5362938af3718ee4e818af6a00c03000001aff01000100000b000403000102002300000000f00010100170000 Protocol = 0301 Ciphersuite = C030 AdditionalData = 00170023FF01 Exception =
Expected Output:	The message can be successfully parsed. The message contains the session ticket, extended master secret, and renegotiation info extensions.
Steps:	1. Parse the message bytes. 2. Verify successful processing, protocol version, and the extensions.

Test Case No.:	TLS-ServerHello-2
Type:	Negative Test
Description:	Parses a ServerHello message with invalid extension length
Preconditions:	None
Input Values:	Buffer = 03039f9cafa88664d9095f85dd64a39e5dd5c09f5a4a5362938af3718ee4e818af6a00c03000001cff01000100000b000403000102002300000000f00010100170000 Protocol = 0303 Ciphersuite = C030 AdditionalData = 00170023FF01 Exception = Invalid argument Decoding error: Bad extension size
Expected Output:	The message cannot be parsed correctly and the processing results into a “Bad extension size” exception.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

19.3 CertificateVerify

The CertificateVerify message contains the following fields:

- Signature and Hash algorithm (only in TLS 1.2)
- Certificate length
- Certificate

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/cert_verify.vec`.

Test Case No.:	TLS-CertVerify-1
Type:	Positive Test
Description:	Parses a correct CertificateVerify message in TLS 1.2.
Preconditions:	None
Input Values:	Buffer = 06010080266481066a8431582157a9a591150d418b63d46154c4cd85bffcfdba8c7f6396f0ceb0402c2142c526a19659d58cd4111bf45f57a56e97d16eeecd350f6e9dc93662e4361053666e5a53c74fe11bd6cf86a9cf7a2488704c5121915820973280ed6afa3e8b79dfb799bddffb52caa2d1a0a895a0e7505d841a882bdd92ec9141 Protocol = 0303 Exception =
Expected Output:	The message can be successfully parsed.
Steps:	1. Parse the message bytes. 2. Verify successful processing.

Test Case No.:	TLS-CertVerify-2
Type:	Negative Test
Description:	Parses a correct CertificateVerify message with an incomplete Signature and Hash algorithm.
Preconditions:	None
Input Values:	Buffer = 06 Protocol = 0303 Exception = Invalid argument Decoding error: Invalid CertificateVerify: Expected 1 bytes remaining, only 0 left
Expected Output:	The message cannot be parsed correctly and the processing results into an exception: “Invalid CertificateVerify: Expected 1 bytes remaining, only 0 left”.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

19.4 Hello Request

The HelloRequest message does not contain any data.

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/hello_request.vec`.

Test Case No.:	TLS-HelloRequest-1
Type:	Positive Test
Description:	Parses a correct HelloRequest message.
Preconditions:	None
Input Values:	Buffer = Exception =
Expected Output:	The message can be successfully parsed.
Steps:	1. Parse the message bytes. 2. Verify successful processing.

Test Case No.:	TLS-HelloRequest-2
Type:	Negative Test
Description:	Parses a correct HelloRequest message with a non-zero size.
Preconditions:	None
Input Values:	Buffer = 01 Exception = Invalid argument Decoding error: Bad Hello_Request, has non-zero size
Expected Output:	The message cannot be parsed correctly and the processing results into an exception: “Bad Hello_Request, has non-zero size”.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

19.5 HelloVerify

The HelloVerify message contains the following fields:

- Protocol version
- Cookie length
- Cookie

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/hello_verify.vec`.

Test Case No.:	TLS-HelloVerify-1
Type:	Positive Test
Description:	Parses a correct HelloVerify message.
Preconditions:	None
Input Values:	Buffer = feff14925523e7539a13d9782af6d771b97d0032c61800 Exception =
Expected Output:	The message can be successfully parsed.
Steps:	1. Parse the message bytes. 2. Verify successful processing.

Test Case No.:	TLS-HelloVerify-2
Type:	Negative Test
Description:	Parses a correct CertificateVerify message with an incomplete cookie.
Preconditions:	None
Input Values:	Buffer = FEFD0500 Exception = Invalid argument Decoding error: Bad length in hello verify request
Expected Output:	The message cannot be parsed correctly and the processing results into an exception: “Invalid CertificateVerify: Bad length in hello verify request”.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

19.6 NewSessionTicket

The NewSessionTicket message contains the following fields:

- Lifetime (4 bytes)
- Length (2 bytes)
- Session ticket

The following table shows an example test case with one test vector. All test vectors are listed in `src/tests/data/tls/new_session_ticket.vec`.

Test Case No.:	TLS-NewSessionTicket-1
Type:	Positive Test
Description:	Parses a correct NewSessionTicket message.
Preconditions:	None
Input Values:	Buffer = 000000000051122334455 Exception =
Expected Output:	The message can be successfully parsed.
Steps:	1. Parse the message bytes. 2. Verify successful processing.

Test Case No.:	TLS-NewSessionTicket-2
Type:	Negative Test
Description:	Parses a correct NewSessionTicket message with an incomplete session ticket.
Preconditions:	None
Input Values:	Buffer = 00010203000500 Exception = Invalid argument Decoding error: Invalid SessionTicket: Expected 5 bytes remaining, only 1 left
Expected Output:	The message cannot be parsed correctly and the processing results into an exception: “Invalid SessionTicket: Expected 5 bytes remaining, only 1 left”.
Steps:	1. Parse the message bytes. 2. Verify the resulting exception content.

20 X.509 Certificates

Botan X.509 certificate tests validate X.509 format processing and correct certificate verification. The tests are divided into three independent test suites:

- X.509 unit tests (in `src/tests/unit_x509.cpp`) performs tests with dynamically generated valid and invalid X.509 certificates, validates processing of certificate extensions or expired certificates.
- X.509 tests (in `src/tests/x509_path_nist.cpp`) performs extended tests with valid and invalid certificates generated with the tool x509test [x509test].
- Extended X.509 name constraints tests (in `src/tests/x509_path_name_constraint.cpp`) performs an extended test with different named constraints used in CA certificates.
- OCSP tests (in src/tests/test_ocsp.cpp) perform tests for parsing OCSP requests and responses, validating responses and testing online OCSP checks.

In the following, we describe these tests in more detail.

20.1 X.509 Unit Test

X.509 unit test performs tests with dynamically generated valid and invalid X.509 certificates and validates their processing in Botan. The test validates key usage extension, expiration dates, or processing of self-signed certificates and certificate issuer properties.

The test is implemented in `src/tests/unit_x509.cpp`.

The following X.509 certificate tests are executed:

- Validity period: tests with valid and expired certificates
- Issuer information storage: tests storage and access to issuer data in certificates
- Certificate revocation
- Detection of self-signed certificates
- Key usage constraints for different cryptographic algorithms: DH, ECDH, RSA, ElGamal, DSA, ECDSA, ECGDSA, ECKCDSA
- X.509v3 extension handling, including writing and reading custom X.509v3 extensions

The X.509 unit test runs 339 tests.

20.2 X.509 Test with Certificate Files

Botan X.509 certificate validation is tested with a set of valid and invalid certificates generated with the tool x509test [x509test]. The following certificate properties and certificates are tested with the generated certificates:

- Key usage and CA key usage extension
- CA flag availability
- CA certificates constructed to contain a loop during validation
- Self-signed certificates
- Subject name
- Alternative names
- Name constraints with DNS names
- Wildcard certificates
- Validity period

The following tables shows an example test case with one test vector. All test vectors are included as certificates in `src/tests/data/x509test`.

Test Case No.:	X509-test-1
Type:	Negative Test
Description:	Certificate authority flag validation
Preconditions:	None
Input Values:	A certificate chain with a certificate, which sets basic constraint *Certificate Authority* to "*No*"
Expected Output:	Out = certificate not allowed to issue certs
1. Steps:	1. Import the root certificate 2. Read the provided certificate chain 3. Validate the certificate chain 4. Check the result of Botan certificate path validation
Notes:	The following file is used for this test: InvalidIntCAFlag.pem The test results are included in the file expected.txt and used for validation.

20.3 Extended X.509 Name Constraints Test

The name constraints extension is an extension used in CA certificates. It indicates a name space within which all subject names of the issued certificates must be located. For example, it indicates the IP addresses of the issued certificates or their domain names.

This test extends the previous tests with further further name constraints:

- Domain names
- DNS name
- email address
- IP address

The following tables show example test cases with one valid and one invalid test vector. All test vectors are included as certificates in `src/tests/data/name_constraint`.

Test Case No.:	X509-name-constraint-1
Type:	Positive Test
Description:	Tests the IP name constraint
Preconditions:	None
Input Values:	Root certificate with the following name constraint extension: Permitted: IP:192.168.0.0/255.255.0.0 Leaf certificate with the following X509v3 Subject Alternative Name: IP Address:192.168.1.1
Expected Output:	Out = Verified
Steps:	1. Import the root certificate 2. Read the leaf certificate 3. Validate the leaf certificate 4. Check the result of Botan certificate path validation
Notes:	The following files are used for this test: • Root_IP_Name_Constraint.crt • Valid_IP_Name_Constraint.crt

Test Case No.:	X509-name-constraint-2
Type:	Negative Test
Description:	Tests the IP name constraint
Preconditions:	None
Input Values:	Root certificate with the following name constraint extension: Permitted: IP:192.168.0.0/255.255.0.0 Leaf certificate with the following X509v3 Subject Alternative Name: IP Address:10.0.1.3
Expected Output:	Out = Certificate does not pass name constraint
Steps:	1. Import the root certificate

	2. Read the leaf certificate 3. Validate the leaf certificate 4. Check the result of Botan certificate path validation
Notes:	The following files are used for this test: • Root_IP_Name_Constraint.crt • Invalid_IP_Name_Constraint.crt

21 OCSP Tests

Botan's OCSP code is tested using different tests that parse OCSP requests and OCSP responses, validate OCSP responses (in terms of signature validation) and also using online tests for randombit.net. Online tests are only executed if BOTAN_HAS_ONLINE_REVOCATION_CHECKS is set. The tests are implemented in `src/tests/test_ocsp.cpp`. All test data can be found in `src/tests/data/ocsp`.