Testing SCAPI

Contents

[Testing SCAPI 1](#_Toc304793375)

[Purpose: 2](#_Toc304793376)

[ Unit test 2](#_Toc304793377)

[ Wrong behavior 2](#_Toc304793378)

[ Vector test 2](#_Toc304793379)

[ Regression test 2](#_Toc304793380)

[Package: 3](#_Toc304793381)

[General design: 3](#_Toc304793382)

[Output: 3](#_Toc304793383)

[Timing: 3](#_Toc304793384)

[Checking results: 3](#_Toc304793385)

[1. Communication layer. 3](#_Toc304793386)

[Wrong behavior: 3](#_Toc304793387)

[Vector test: 4](#_Toc304793388)

[Input size: 4](#_Toc304793389)

[Number of participants: 4](#_Toc304793390)

[Performance: 5](#_Toc304793391)

[General design 5](#_Toc304793392)

[2. Primitive layer. 5](#_Toc304793393)

[Wrong behavior: 5](#_Toc304793394)

[Vector test: 6](#_Toc304793395)

[Sources: 6](#_Toc304793396)

[Platforms: 6](#_Toc304793397)

[Performance 6](#_Toc304793398)

[3. Testing general design 7](#_Toc304793399)

[4. Testing Detailed Design 7](#_Toc304793400)

[PRG 9](#_Toc304793401)

[PRF 9](#_Toc304793402)

[Target Collision resistant Hash 10](#_Toc304793403)

[Trapdoor permutation 10](#_Toc304793404)

[Perfect universal hash 11](#_Toc304793405)

[Key derivation function 11](#_Toc304793406)

[Discrete logarithm 12](#_Toc304793407)

[5. Appendix 13](#_Toc304793408)

[Table of algorithms/ sources: 13](#_Toc304793409)

[Output file format 14](#_Toc304793410)

# Purpose:

The goals of the Testing system are:

1. To check that SCAPI meets the requirements.
2. To find bugs and undesired behavior. We aim to detect software failures so that defects may be discovered and corrected.
3. To make sure that new bugs are not introduced in new versions.

To reach all these goals, testing SCAPI is composed of the stages described below:

## Unit test

The programmer is responsible of checking that the component he writes works, that is, it does not crash and performs requirements. In this test the programmer may use dummy data.

## Version test

A new version can contain additional features to the existing features, bug fixing and changes in the existing modules. When a new version is added to the API, we need to test the following things:

* Expected behavior - the programmer or a QA member checks that the component behaves as expected when passed known and standard inputs.

SCAPI will be tested with the known test vectors of NIST and IETF if exist. If we do not find test vectors from NIST or IETF we will look for a different source or make up our own expected outputs.

* Wrong behavior - the tester uses wrong data in purpose in order to check proper response. He also uses the component in various unexpected ways to assure that the program deals with these situations in an elegant manner.
* Performance test – the programmer or a QA member runs the component on a regular and standard input and take the time it took.
* Data loading – the programmer or a QA member runs the component many times or on an extra ordinary large input to check that the component does not crash and finish working in a reasonable amount of time.

## Regression test

When a new version is developed (i.e. new features, bug fixes, etc.) before going through the version test a regression test should be performed to check that previous working features were not damaged. Once this is ascertained and the version test also succeeded, we can say that the new version is working well and we can release it. We will also save the snapshot of this new version to compare in the future. The “snapshot” includes:

1. A label of the version.
2. Configuration files used in the testing module.
3. The output file contains the results of the tests.

Another usage of regression test is when only cosmetic changes were performed. A regression test should be performed to check that the changes did not damage previous working features.

Our test package contains the last two tests. The unit tests are done by the programmer after he writes each component and will not be include in the test package.

# Packages:

All testing files will be under package edu.biu.scapi.tests. Sub packages will be dedicated for each tool.

* edu.biu.scapi.tests.comm – tests classes of the communication layer
* edu.biu.scapi.tests.primitives – tests classes of the primitive's layer
* edu.biu.scapi.tests.midLayer – tests classes of the mid layer
* edu.biu.scapi.tests.configurationFiles – configuration files used in the tests classes.
* edu.biu.scapi.tests.configurationClasses – utility classes used in the tests. This will be explained later.

# Platforms:

We will check different operating systems for the use of native external libraries. We will match the dll/lib application to the tested environment.

# Performance:

We will compare similar algorithms from different sources. The decision which default implementation to supply will take into account the results of the performance tests.

# Layers detailed explanations

As mentioned above, we will test out components for four tests:

* Expected behavior
* Wrong behavior
* Performance
* Data loading

Below we explained each test in each layer.

## Communication layer.

### Wrong behavior:

This test will be for incorrect behavior and how the tool handles them. The following tests will be performed.

1. An unauthorized IP/Party will try to connect.
2. One Party will not respond to connection requests and will not initiate connection to the parties it should connect to.
3. Close a socket in the middle of communication setup and after setup.
4. Close the program of one party in the middle of the setup stage and after setup.
5. Overload one station with lots of applications running taking the time of the CPU. Alternatively, use a slow computer. During setup and if setup has succeeded also after setup.
6. Put a lot of traffic on the network in order to cause a major slow down in communication for our tool. During setup and if setup has succeeded also after setup.
7. Make a listening port unavailable.
8. Cause one party (if possible in the success function) to output true and other parties to output false on the success and see how the communication is progressing.
9. Use a key that did not match the session key provided by the key exchange algorithm.
10. Ask for secure channel but do not give a key exchange protocol or a key.
11. Give different list of participants to different parties.
12. Give the same party twice in the list.
13. Give the same IP and port for different parties.
14. Give certain ID's for one party and different ID's to other parties. The IP's will be the same.

### Expected behavior:

In addition, we will check that the package is working as expected. In this test there will be 2 or 3 parties with different IP addresses and a couple of messages passed between them. This simple test will be composed of several tests which include different configurations. We will use the following configurations.

1. Use pre-defined keys and do not ask for the key exchange algorithm.
2. Key exchange – Ask for the different key exchange algorithms.
3. Naïve success function.
4. At first stage, use the TwoPartiesSuccess success algorithm. After the rest of the success functions will be coded, check them too.
5. Plain channel
6. Encrypted Channel
7. Authenticated channel
8. Secure (both) channel
9. As a unit test check different timeouts.

### Performance:

#### Input size:

Messages of different sizes will be sent. The aim is to test the ability of the tool to handle a variety of message sizes. Specifically, we will use the communication layer to send the following messages.

* One byte
* 10's to 1000's of bytes.
* A message of one megabyte
* A full time movie

The time it takes for such messages to pass will be measured.

#### Number of participants:

We want the testing to include numerous parties. This test is not a regression test and will not be performed for every version, rather periodically.

We will test the tool for 10, 100 and 1000 parties. The test will be done in two ways:

* The tests will be maintained in our lab and thus parties will share the same IP's but own different ports.
* To make the tests more realistic, we will use PlanetLab (or any other test bed that we find suitable). PlanetLab is a collection of machines distributed over the globe that can be used. The idea is to place our software on remote computers and check communication.

The time it takes for the communication layer to set up for the different number of parties will be measured.

#### Channel type:

The overhead of encrypted and authenticated channels will also be compared to simple plain channel.

### Data loading

In this test we check edge cases. We will test cases of extra ordinary large message, very large amount of participants, etc.

### Nagle – unit test

The option of nagle is by default turned on in Java. We will run performance test with both options, that is, nagle turned on and off. We consider letting the user determine this option and supplying an API accordingly.

## Primitive layer and Mid layer.

### Wrong behavior:

This test will be for incorrect arguments and how the tool handles them. We will perform the following tests. Every family will test the cases that are related to it. Detailed information of each family will be given later in this section.

1. Wrong key size
2. Wrong type of key. For example, an RSA key will be passed to the Rabin algorithm.
3. Do not call setKey. Call functions of algorithms that should be initialized with a key without initializing them.
4. Wrong block size. Call functions of an algorithm that expects a certain size of input with a different size.
5. Size not specified. Call an algorithm that expects the size as argument without specifying the size. For example, call computeBlock of prf with varying input without size.
6. Wrong offset. Pass an array and an offset. Set the offset to be higher than the size of the input array.
7. Wrong length. Pass an array and a length. Set the length to other than the array length.
8. Wrong argument. For example, an AlgorithmParametersSpec for RSA will be sent to Hmac.
9. Wrong algorithm. For example, call the prfFactory with "SHA1" as the algorithm name.
10. Wrong type of argument. For example, pass a ZpElement to ECDlogGroup in DlogGroup.
11. Wrong usage. For example, call the update function of a CBCMac without call startMac first.

Some of these tests are avoided by the compiler checks. For example, CryptoPpDlogZp accepts a ZpGroupParams in its constructor. Therefore, passing an ECGroupParams will result in compilation error.

A table with the expected result for each of the tests we perform can be found in the [appendix](#_Wrong_behavior_and).

### Expected behavior:

In addition, we will test each component for proper behavior, meaning that the component does its’ job correctly. Examples of this test are:

1. In Pseudorandom Function family, check that on fixed input and key the output of the compute function is as expected.
2. In Dlog Group family, check that on fixed group element and exponent the output of the exponentiate function is as expected.
3. In Trapddor Permutation family, check that the TP element outputted by the compute function from a certain input is inverted to the same input as the original.
4. In Mac family, check that the tag given by the mac function on a fixed message is verified by the verify function.
5. In encryption family (both symmetric and asymmetric), check that the ciphertext outputted by the encrypt function from a certain plaintext is decrypted to the same plaintext as the original.
6. In Digital Signature family, check that the signature outputted by the sign function from a certain message is verified by the verify function.

# Testing general design:

As mention above, every time there is a new version there are two things that have to be checked:

1. Check that the changes of the new version do not cause any problems to modules that already worked. This is done by the regression test.
2. Check that the new features of this version are working as expected. This is done in the version test.

## Regression test

The regression test is a standalone application that has a user interface. In general, the user can choose a previous version (in most cases the user will choose the latest one) and the regression test will automatically run the test to compare the output of the current version with the output of the selected one.

The regression test can do a partial compare, meaning comparing only the expected behavior and wrong behavior; or a full compare, meaning comparing also the performance and data loading. The last two checks take a lot of running time and there are cases the user may want to skip them to save time. These cases can happen when the user added cosmetic changes and just want to be sure that there is no damage. These two checks will run mostly when there is a final version that we want to fully test.

When we run the regression test, the application takes the configuration files of the previous selected version and **uses them to test the existing version**. Then, it compares the output to the output that was saved in the previous version. If both files are the same, the regression test result is success. Otherwise, the result is failure.

## Regression test output

In both regression test results, success or failure, there will be a log file contains the output. This log file will contain for each object, the result of each version in each test function. IF there are changes between the outputs, it will be written too. Example of a log file can be found in the [Regression test output file format](#_Regression_test_output) section in the appendix.

## Version test

The version test includes functions to test expected behavior, wrong behavior, performance test and data loading test. The version test performs these tests on modules in SCAPI upon user request.

Version test can be, as the regression test, partial or full.

## Version test input

The version test tests objects from our API. We want to allow the users determine which object and which functions to test and we want that a change of the object being checked will be easy and will not cause changes in the code.

Thus, we decided that the preferred way to get the input to test is from a configuration files. There are two types of configuration files:

1. **One** configuration file that contain the modules to tests. In this file the user writes the names of the modules to check. For example:

Pseudorandom Function

Pseudorandom Generator

…

Dlog Group

1. Configuration files **for each component** that contain the objects in this component that need to be tested. The general structure of this file is as follows:

* For each object needs to be tested:
* Name of algorithm
* Name of provider
* For each test function needs to be checked:
* Number of times this function should run (in case the user want the check multiple inputs)
* For each running:
* Parameter1
* Parameter2
* …
* parameter N

Each component has a unique structure, depends on the inner structure of the test classes and the functions needs to be tested. The configuration files formats can be found in the [configuration files format](#_Configuration_files_format) in the appendix.

## Version test output

The output of the tests of each version will be written to a file in the tests folder. The file's name will be: TestResults + [Time & date].csv.

The file format can be found in the [Version test output file format](#_Version_test_output) in the appendix. It contains the output in each test function, for each tested object.

We will have a unique place in the SVN where we put our final versions. When we have a final version that we already test we will save it in this place along with the configuration files and the version test output file. Each version will have a folder includes these 3 parts:

* version code
* version configuration files
* version output file

## Test classes structure

The main components of the testing package are:

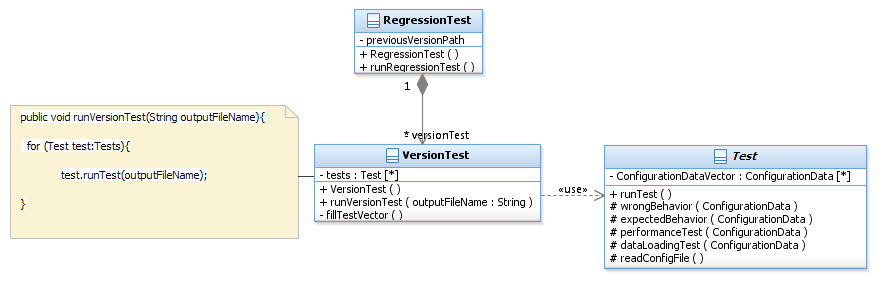
* A RegressionTest class.
* A VersionTest class.
* An abstract Test class.
* A specific class for each SCAPI component that extends the Test class and additional derived classes for some of them, as will be explained later.

In general, we will have an abstract class Test (with a function runTest) that all tests must implement. The function runTest tests a specific version for wrong behavior, expected behavior, data loading and performance test.

We will have a VersionTest class that holds a vector of Tests and will perform runTest on each instance in the vector.

RegressionTest class holds an instance of the regression test and the path of the selected previous version.

The following diagram explained the relationships between the tests classes:



Each primitive family will have a class that implements the Test class. These classes will implement the functions wrongBehavior(), expectedBehavior(), dataLoadingTest() and performanceTest() according to their requirements. For example, prfTest class will implement the expected behavior function to test the compute function of the Prf interface.

As mentioned before, the inputs to check will be given in a configuration file. Every concrete test class will have a known configuration file that the class reads in the construction time. The configuration file contains the objects to check and a list of checks to run on them.

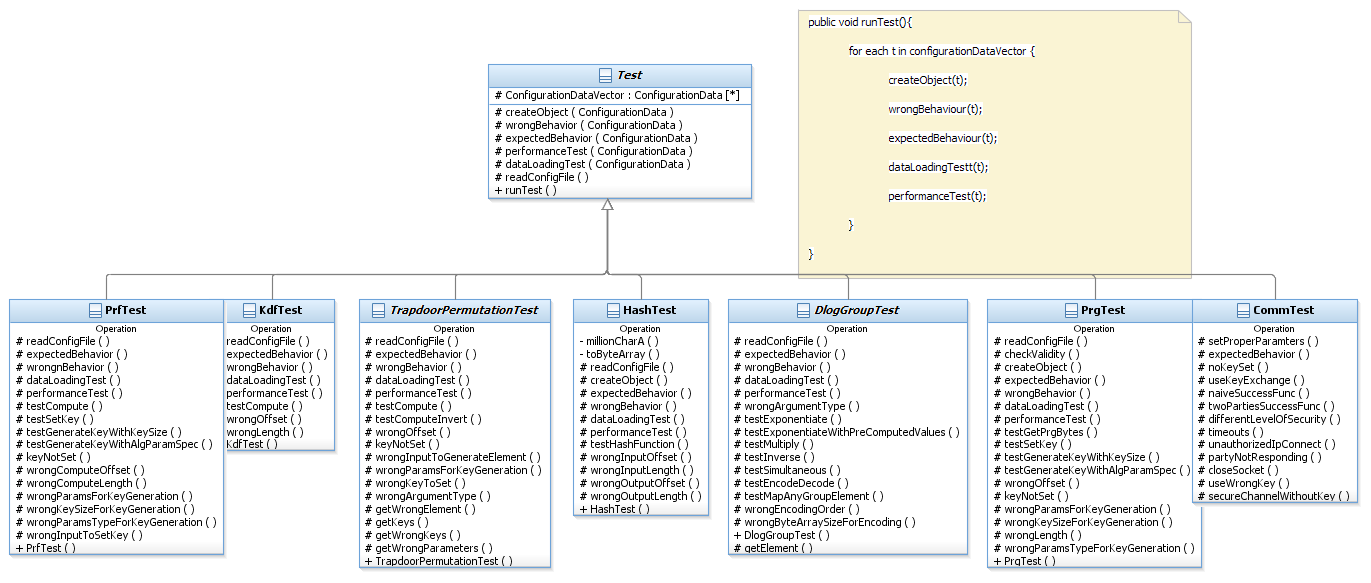
Respectively, each concrete test class will have a utility class contains the data to check. Each object in the configuration file is converted to an instance of this utility class.

In the constructor of a concrete family test class there will be a call to a readConfigFile function. This function reads the configuration file of this family and performs the following algorithm:

* Converts each object mentioned in the file to an instance of the utility class.
* Inserts the created instance to the configurationDataVector.

As a result, in the end of readConfigFile function the test class will have a vector of instances of the utility class that match this test.

Below is a diagram containing the Test abstract class and the primitive families’ classes. It also contains the commTest.



# Testing detailed design

As mentioned above, the testing module contains an abstract Test class and family test classes (PrfTest, PrgTest and so on) that extend it.

In addition, we have the same structure for utility classes used in the test classes to hold the parameters for the tests. There is an abstract class called ConfigurationData and a utility class for every family, called [familyName]ConfigurationData, that extends he abstract class. The abstract utility class holds an algorithm name and provider name, while the concrete utility classes hold the parameters for the checks that the family’s test class is testing.

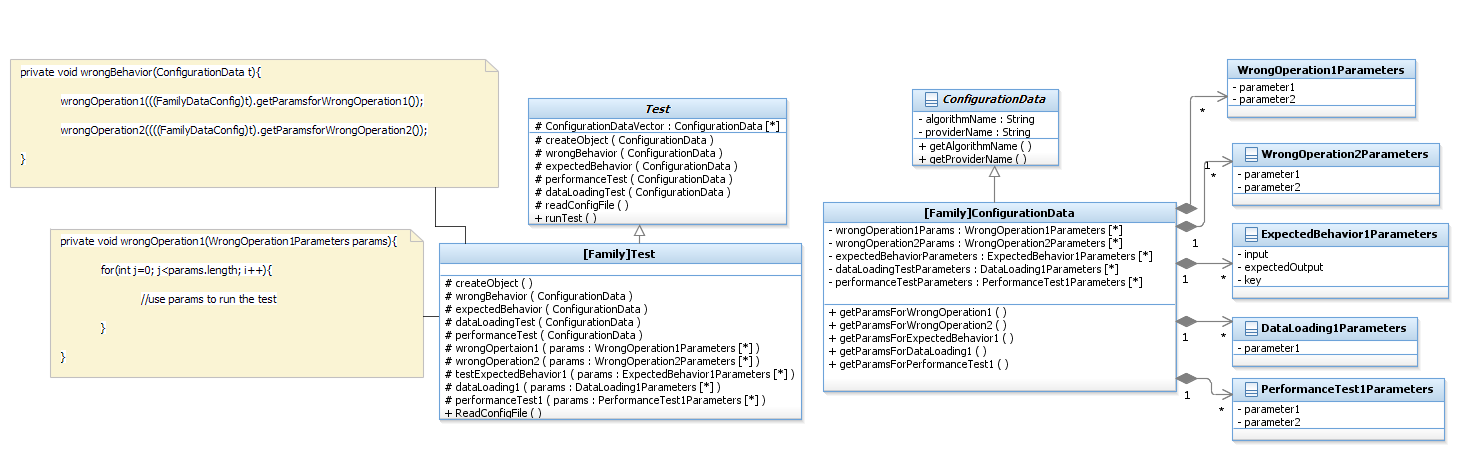
Every [Family]ConfigurationData class has the following attributes:

* Name of algorithm to test. (given from the abstract class)
* Provider name. (given from the abstract class)
* Nested class for each test function in the corresponding test class that holds the parameters for this function.
* An array of instances of each nested class.

The abstract Test class holds a vector of ConfigurationData. This vector contains the object that the test classes need to check. The Test class also has the following functions:

* createObject
* expectedBehavior
* wrongBehavior
* performanceTest
* dataLoadingTest
* runTest

The first five functions are protected while the last function is public. The function runTest call the other functions for each instance of the configurationData vector (as shown in the previous diagram).



The diagram above shows a general family test. The class [Family]Test has two wrongOperation functions, one expectedBehavior function, one dataLoadingTest function and one performanceTest function. Respectively, there is a [Family]ConfigurationData that has five nested classes, one for each test function in the [Family]Test class. The [Family]ConfigurationData class holds five vectors of parameters, one for each test function. Every vector contains instances of different nested class of the [Family]ConfigurationData.

An instance of [Family]ConfigurationData represents an object that need to be tested. The object can be built by the factory of the family using the algorithm name and provider name. The tests that need to be executed are represented in the arrays of parameters in the [Family]ConfigurationData. If there is an instance in one of the arrays, the relevant check will get these parameters and run the check on the object.

The structure of the family test classes is as follows.

1. Holds a vector of ConfigurationData, which is the abstract utility class that holds a data to the checks. (This vector is given from the abstract Test class.)
2. There are functions for testing wrong behavior, expected behavior, data loading test and performance test. These functions perform their tests using the parameters given in the configuration class instance.

As a convention, every expected behavior function starts with the prefix “test” and every wrong behavior function starts with the prefix “wrong”.

1. A readConfigFile function that fills the vector of ConfigurationData with objects of the corresponding utility class.

In the constructor of the family test class there will be a call to the function readConfigFile. This function will read the configuration file of this family and fill the vector of ConfigurationData. The configuration file contains the name of the algorithm that needs to be checked, the provider’s name (optional) and a block of parameters for each check. If there is a check that is not related to the algorithm, the block of parameters of this check shouldn’t be in this algorithm’s part of the file. It is up to the tester to build a correct configuration file, and it should be done with caution. The format of a configuration file for each family can be found in the [appendix](#_Configuration_files_format).

The function create object used in order to seve the creation of the tested object in each test function. The function creates the tested object once, in the beginning of the runTest function and saves the created object. The other functions run the tests on the saved object. This way the creation of the tested object is done once for each instance of ConfigurationData.

Wrong behavior and expected behavior functions call some inner functions, each one checks different case.

Each inner check function gets an array of the required parameters class. In most cases this array will hold just one instance of parameter class, but there will be cases that the tester wants to test multiple input, too. The function works in the following way:

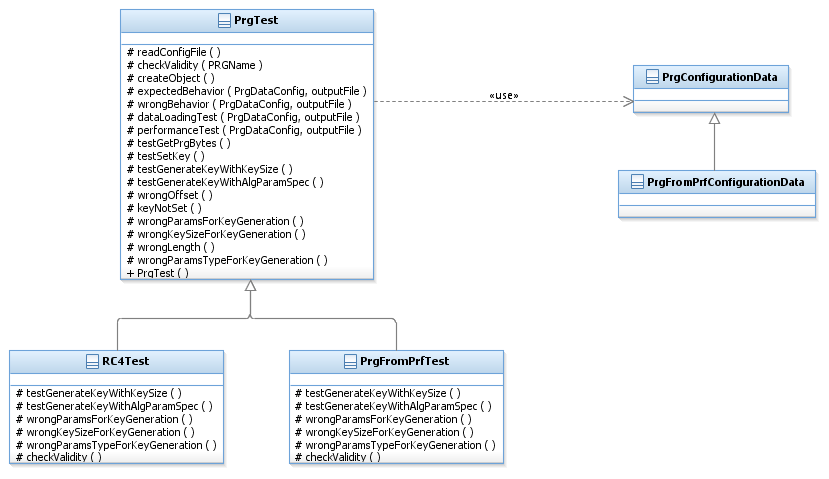
* For each instance of the given array
* Run the test of this function using the parameters in this instance. (see note of wrongOperation1 in the diagram above)

Note that there are cases of component in the family that need additional tests. For example, a prp object needs to check the invert function, while other prf objects do not. In these cases we will add derived classes that add these additional checks. The derived class will have a different configuration class, while the configuration file may be different and may not, depends on the specific case.

List of the test classes and configuration files can be found in the [appendix](#_List_of_family).

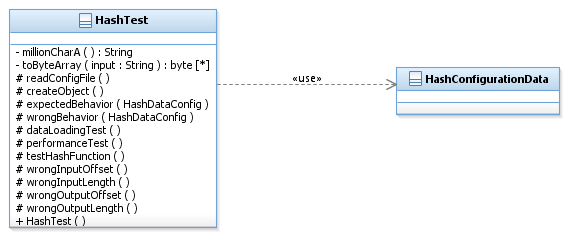
The following class diagrams display the testing classes of each primitive family.

### PRG

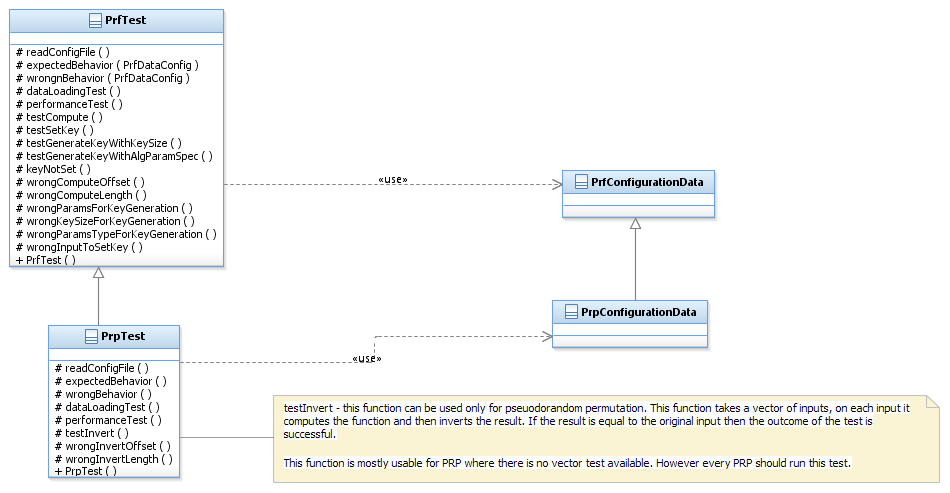


Since RC4 and PrgFromPrf algorithms have different implementations of generateKey functions, we separate the tests of these functions into two derived classes. Each derived class will implement the test function according to the algorithm implementation.

### CryptographicHash



### PRF



Note that prp has a different test class than the prf since it needs to add checks for the invert function.

Since all our implementation of PRF algorithm so far implement the generateKey functions in the same way, (generateKey that accept an AlgorithmParameterSpec throws an UnSupportedOperationException, and generateKey that accept keySize is the function that generates the key), we choose to implement the test of these function in the PRFTest level instead of declare it an abstract functions.

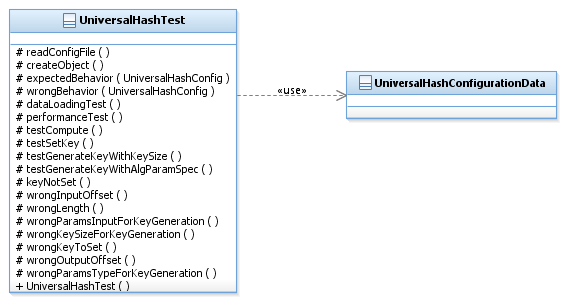
This allows us to have one generic class for PRF instead of many small classes that have the same implementation of these test functions.

The test functions that related to the key generation are:

* testGenerateKeyWithKeySize
* testGenerateKeyWithAlgParamSpec
* wrongParamsInputForKeyGeneration
* wrongParamsTypeForKeyGeneration
* wrongKeySizeForKeyGeneration

As a result, if there will be in the future an algorithm that implement the generateKey functions differently, it will have to add a test class that extend the PRFTest class and override the implementation of the generateKey tests.

### Universal hash function



As the same as PRF, since all our implementation of UniversalHash algorithm so far implement the generateKey functions in the same way, we choose to implement the test of these function in the UniversalHashTest level instead of declare it an abstract functions.

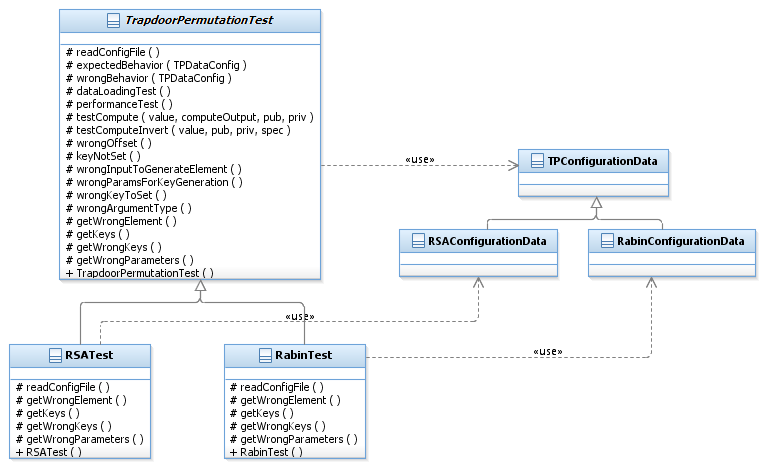
This allows us to have one generic class for UniversalHash instead of many small classes that have the same implementation of these test functions.

The test functions that related to the key generation are:

* testGenerateKeyWithKeySize
* testGenerateKeyWithAlgParamSpec
* wrongParamsInputForKeyGeneration
* wrongParamsTypeForKeyGeneration
* wrongKeySizeForKeyGeneration

As a result, if there will be in the future an algorithm that implement the generateKey functions differently, it will have to add a test class that extend the UniversalHashTest class and override the implementation of the generateKey tests.

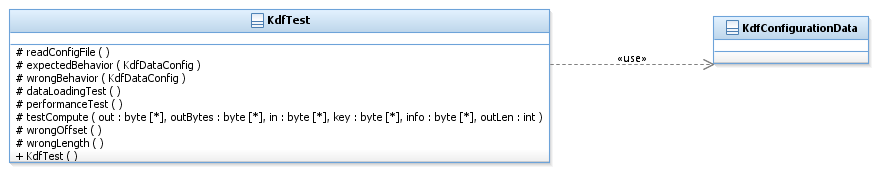
### Trapdoor permutation



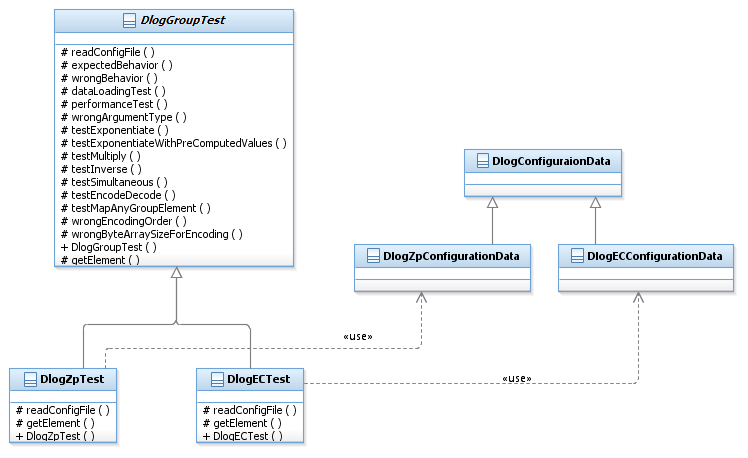
Note that there is a separation between RSA and Rabin tests. These algorithms are completely different although they both are trapdoor permutation, and have different keys, params and elements.

Despite this, the tests themselves can treat them in the same way, by getting the specific parameters like keys by a get function. Thus, the tests remain in the abstract level, while each concrete algorithm implement the gets functions.

### Key derivation function



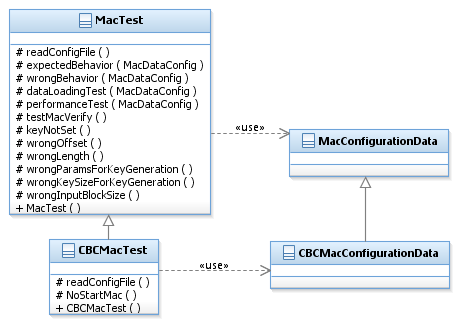
### Discrete logarithm



Note that there is a separation between Dlog over Zp and Dlog over elliptic curve. This separation is because the elements are different, in Zp an element is a number and in elliptic curve an element is a point with two coordinates. Thus, the configuration file and config class cannot be the same and need to be separate. But, because the only difference is in the elements, the tests functions remain in the abstract dlog level and when there is a use in an element, there will be a call to getElement function.

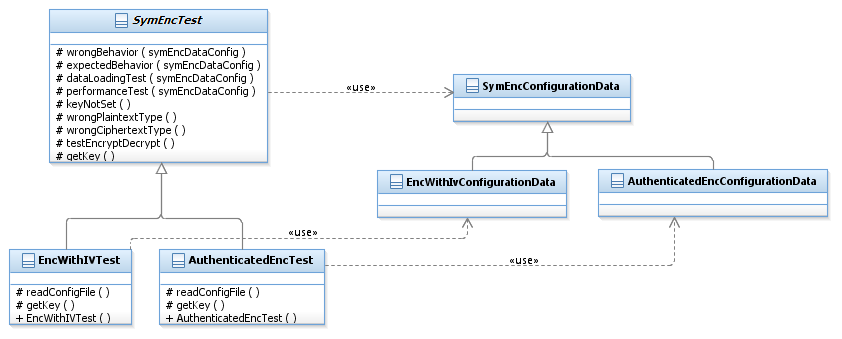
The following class diagrams display the testing classes of each mid layer component.

### Mac



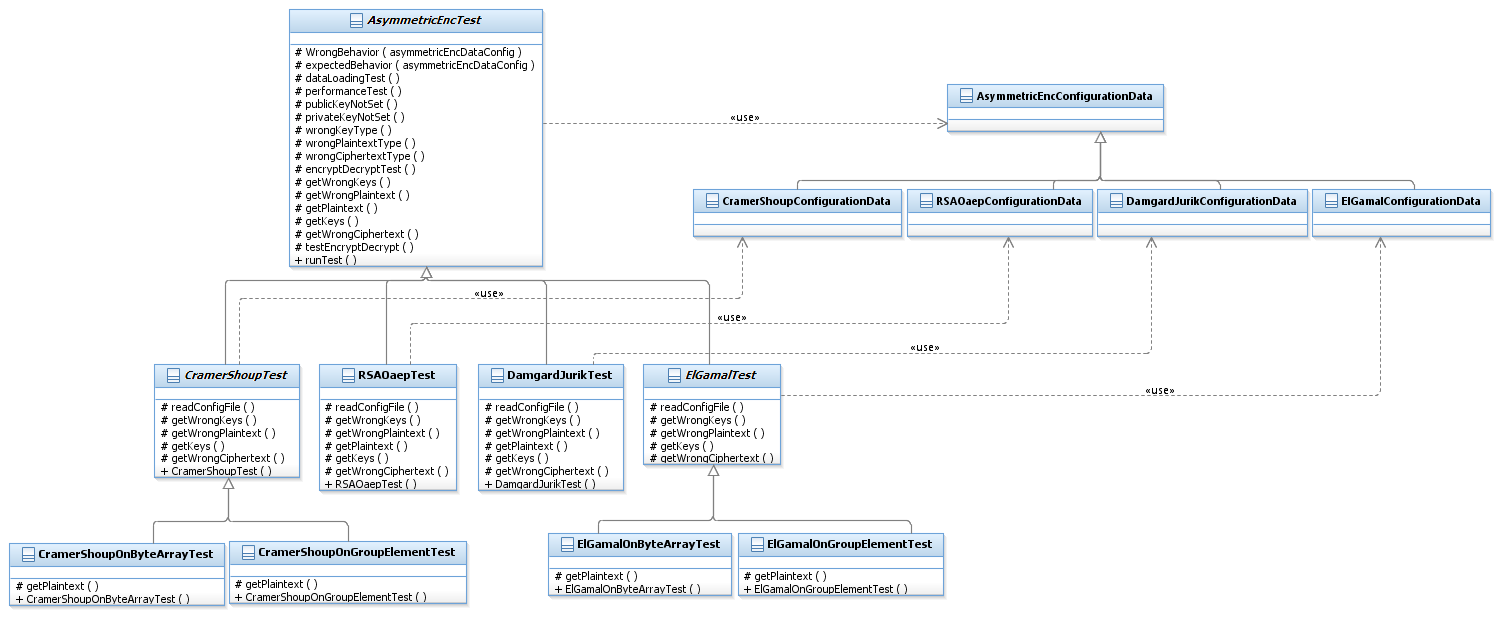
Note that CBCMac has a different test class than the mac since it needs to add checks for the startMac function.

### Symmetric encryption

****

Note that there is a separation between encryption with IV and authenticated encryption tests. This separation is because the keys are different and cannot be passed in the same way.

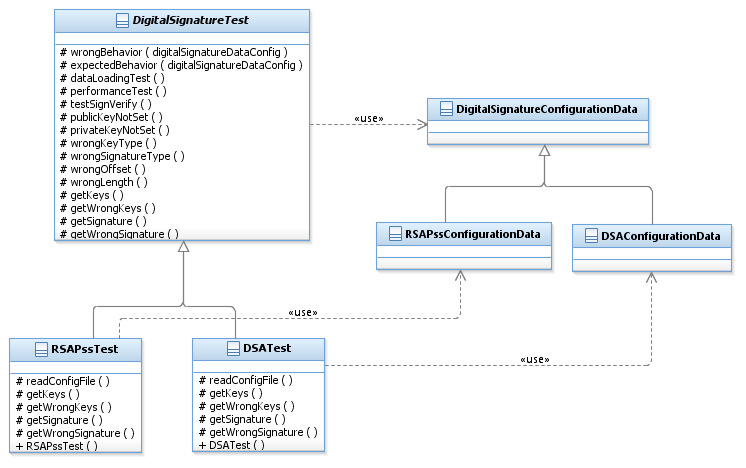
### Asymmetric encryption

****

Note that there is a separation between the algorithms of the asymmetric encryption family. This separation is because the algorithm has different keys, plaintexts and ciphertexts. Therefore, the configuration file cannot be the same and each algorithm needs to create the parameters that fits it requirements.

Despite this, the tests themselves can treat them in the same way, by getting the specific parameters like keys by a get function. Thus, the tests remain in the abstract level, while each concrete algorithm implement the gets functions.

### Digital signatures



Note that there is a separation between RSAPss and DSA test. These are different algorithm that has different keys and signatures. Therefore, the configuration file cannot be the same and each algorithm needs to create the parameters belong to him. Despite this, the tests themselves can treat them in the same way, by getting the specific parameters like keys by a get function. Thus, the tests remain in the abstract level, while each concrete algorithm implement the gets functions.

## Appendix

### Table of algorithms/ sources:

In the following table we list all the algorithms we need to test and the source of the vector tests.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Interface | Primitive | NIST | IETF | Other |
| HashTest | SHA1 | <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> page 25 |  | Handbook of Applied ryptography, page 345 |
| SHA256 | <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> page 33 |  |  |
| SHA224 |  | http://tools.ietf.org/rfc/rfc3874.txt |  |
| SHA512 | <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> page 51 |  |  |
| SHA384 | <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf> page 56 |  |  |
| Universal one-way/target collision-resistant hashing |  |  |  |
| universal hash functions |  |  |  |
| PRPTest | AES | <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>, page 35 |  |  |
| TripleDes | <http://csrc.nist.gov/publications/nistpubs/800-20/800-20.pdf> for different modes of operation. |  |  |
| PRP varying input/output length |  |  |  |
| PRFTest | Hmac |  | <http://datatracker.ietf.org/doc/rfc4231> |  |
| PRF varying input and output length |  |  |  |
| PRGTest | RC4 |  | <http://datatracker.ietf.org/doc/draft-josefsson-rc4-test-vectors> | http://www.freemedialibrary.com/index.php/RC4\_test\_vectors |
| SHA-based PRG |  |  |  |
| TrapdoorPermutationTest | RSA function |  |  | <http://www.rsa.com/rsalabs/node.asp?id=2125> check the zip file : [RSA-OAEP and RSA-PSS test vectors (.zip file)](ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1-vec.zip) |
| Rabin function |  |  |  |
| DiscreteLogTest | Zp\* |  |  |  |
| EC over the field GF(2m) |  |  |  |
| EC over the field Zp\* |  |  |  |
| KDFTest | HKDF |  |  |  |
| KDFISO18033 |  |  |  |

### Wrong behavior and expected results

|  |  |
| --- | --- |
| **Wrong behavior** | **Expected result** |
| Wrong key size | InvalidKeyException |
| Wrong key type | InvalidKeyException/InvalidParameterSpecException |
| keyNotSet | IllegalStateException |
| Wrong offset | ArrayOutOfBoundsException, NegativeArraySizeException |
| Wrong length | ArrayOutOfBoundsException,  NegativeArraySizeException |
| Wrong argument type | IllegalArgumentException |
| Wrong algorithmParameterSpec | InvalidParameterSpecException |

### List of family test classes and the configuration file they uses

|  |  |
| --- | --- |
| RC4 | PRGTestConfig |
| PrgFromPrf | PRGTestConfig |
| PRP | PRPTestConfig |
| PRF | PRFTestConfig |
| UniversalHash | UniversalHashTestConfig |
| CryptographicHash | HashTestConfig |

### Configuration files format

#### PRF configuration file format:

NumOfObjects =

PRFName =

ProviderName=

NumOfTests =

TestCompute

NumOfIterations =

input =

expectedOutput =

key =

TestSetKey

NumOfIterations =

key =

TestGenerateKeyKeySize

NumOfIterations =

keySize =

WrongInputToSetKey

NumOfIterations =

key =

WrongComputeOffset

NumOfIterations =

key =

input =

inputOffset =

outputOffset =

outputLength =

WrongComputeLength

NumOfIterations =

key =

input =

inputLength =

outputLength =

WrongKeySizeForKeyGeneration

NumOfIterations =

keySize =

#### PRP configuration file format:

NumOfObjects =

PRPName =

ProviderName=

NumOfTests =

TestCompute

NumOfIterations =

input =

expectedOutput =

key =

TestInvert

NumOfIterations =

input =

key =

TestSetKey

NumOfIterations =

key =

TestGenerateKeyKeySize

NumOfIterations =

keySize =

WrongInputToSetKey

NumOfIterations =

key =

WrongComputeOffset

NumOfIterations =

key =

input =

inputOffset =

outputOffset =

outputLength =

WrongComputeLength

NumOfIterations =

key =

input =

inputLength =

outputLength =

WrongInvertOffset

NumOfIterations =

key =

input =

inputOffset =

outputOffset =

outputLength =

WrongInvertLength

NumOfIterations =

key =

input =

inputLength =

outputLength =

WrongKeySizeForKeyGeneration

NumOfIterations =

keySize =

#### CryptographicHash configuration file format:

NumOfObjects =

HashName =

ProviderName =

NumOfTests =

TestHashFunction

NumOfIterations =

input =

expectedOutput =

WrongInputOffset

NumOfIterations =

input =

wrongOffset =

WrongInputLength

NumOfIterations =

input =

wrongLength =

WrongOutputOffset

NumOfIterations =

input =

wrongOffset =

WrongOutputLength

NumOfIterations =

input =

wrongLength =

#### UniversalHash configuration file format:

NumOfObjects =

UniversalHashName =

NumOfTests =

TestCompute

NumOfIterations =

input =

expectedOutput =

key =

TestSetKey

NumOfIterations =

key =

TestGenerateKeyKeySize

NumOfIterations =

keySize =

WrongInputOffset

NumOfIterations =

key =

input =

wrongOffset =

WrongLength

NumOfIterations =

key =

input =

wrongLength =

WrongOutputOffset

NumOfIterations =

key =

input =

wrongOffset =

WrongKeySizeForKeyGeneration

NumOfIterations =

keySize =

#### Prg configuration file format:

NumOfObjects =

PRGName =

NumOfTests =

TestGetPRGBytes

NumOfIterations =

outputLength =

expectedOutput =

key =

TestSetKey

NumOfIterations =

key =

TestGenerateKeyKeySize

NumOfIterations =

keySize =

WrongOffset

NumOfIterations =

key =

outputLength =

wrongOffset =

WrongLength

NumOfIterations =

key =

expectedOutputLength =

wrongOutputLength =

WrongKeySizeForKeyGeneration

NumOfIterations =

keySize =

WrongParamsForKeyGeneration

entropySource =

keySize =

### Regression test output file format

### Version test output file format

The output will be written to a csv file format every time we run the testing software. As mentioned above the file's name has the current date & time information, therefore a new file will be created for each run. This will allows us to keep track of the test performed as well as make comparisons of results. The format of the file is as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Provider** | **Computation** | **Type of test** | **Input** | **Output** | **Result** |
| AES | BC | Compute | Wrong behavior |  |  | Success: Expected exception "BlaBla" thrown. |
| AES | BC | Compute | Vector test | IV1 | OV1 | Failure |
| AES | BC | Compute | Vector test | IV2 | OV2 | Success: [Explanation] |
| AES | BC | Invert | Wrong behavior |  |  | [Success/Failure] |
| … | … | …. | … | … | … | … |
| AES | Crypto | Compute | Wrong behavior |  |  | [S/F] |
| … | … | … | … | … | … | … |