# 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:

We have some goals in testing SCAPI:

* Detect software failures so that defects may be discovered and corrected.
* Find bugs and undesired behavior.
* Check that SCAPI meets the requirements.
* Make sure that new bugs are not introduced in new versions.

To reach all these goals, testing SCAPI composed of some stages described below.

#### Unit test

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

#### 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.

#### Vector test

The programmer or a QA member check that the component behaves as expected when passed known and standard vector tests.

#### Regression test

Once we are satisfied that the current version works well we save a “snapshot” of the current state. The “snapshot” includes a label of the version as well as the results of the tests. When a new version is developed (i.e. new features, bug fixes, etc.) after going through the first three stages a regression test should be performed to check that previous working features were not damaged.

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.

### Package:

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

* edu.biu.scapi.tests.comm – testing of the communication layer
* edu.biu.scapi.tests.primitives – testing of the primitive's layer

### General design:

Our test package contains the last three tests. The unit tests are done by the programmer after he writes each component and will not be include in the test package. The regression test of each concrete test will run its own wrong behavior test and the vector test.

### Output:

The output of the tests will be written to a file upon request.

The file format can be found in the [output file format](#_Output_file_format) in the appendix

### Timing:

The tests will be performed every version for regression tests. The testing package is a standalone application. To keep track, every version will include the regression test result files produced by the test package.

### Checking results:

* If all the tests passed, the outcome of the regression test is success.
* The results will be given in files. We will automatically check if there is any difference between the results of the current version and a past tested version. The automatic check will output the difference in a new file.

## Communication layer.

### Wrong behavior:

The first test will be for incorrect behavior and how the tool handles them. The test will print the thrown exceptions. 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.

### Vector test:

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.

### 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.

### Performance:

We will check the performance of the communication layer. We will measure the amount of time it takes an average size message to get from one point to another. We will also check different environments. Both the proper and wrong behavior tests with varying message sizes will be timed. The overhead of encrypted and authenticated channels will also be compared to simple plain channel. Different number of participants will also be compared.

### General design

The tests of the communication layer will be divided into two, regression tests and tests that we do not intend to run for every version (such as causing slow communication, communication traffic, different lists of ip's and other tests that do not include code). The regression test will be part of the design of the testing package. For more information see section [Testing general design](#_General_design:).

## Primitive layer.

### Wrong behavior:

The first test will be for incorrect arguments and how the tool handles them. We will use the following configurations. Every family will test the configurations that related to it. For detailed information of each family, see section [Testing detailed design.](#_Testing_Detailed_Design)

1. Wrong key size
2. Wrong type of key. For example, an RSA key will be passed to the Rabin algorithm.
3. Wrong encoding. Pass a key encoded in one way but indicates an encoding of a different way.
4. Do not call init. Call functions of algorithms that should be initialized without initializing them.
5. Wrong block size. Call functions of an algorithm that expects a certain size of input with a different size.
6. 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.
7. Wrong offset. Pass an array and an offset. Set the offset to be higher than the size of the input array.
8. Wrong length. Pass an array and a length. Set the length to other than the array length.
9. Wrong argument. For example, an AlgorithmParametersSpec for RSA will be sent to Hmac.
10. Wrong algorithm. For example, call the prfFactory with "SHA1" as the algorithm name.
11. Wrong algorithm. For example, call a factory with a name of algorithm that does not exist.
12. Perform bad casting. Cast type that is determined in run time to a wrong type.
13. Wrong type of argument. For example, pass a ZpElement to ECDlogGroup in DlogGroup.

### Vector test:

In addition, we will check that the families in the package are working as expected. In this test every family will have a test vector containing input and expected output. The test will operate the families’ actions on the input and return success if the output will be as expected.

### Sources:

SCAPI primitive layer will be tested with the 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 test vectors.

### 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

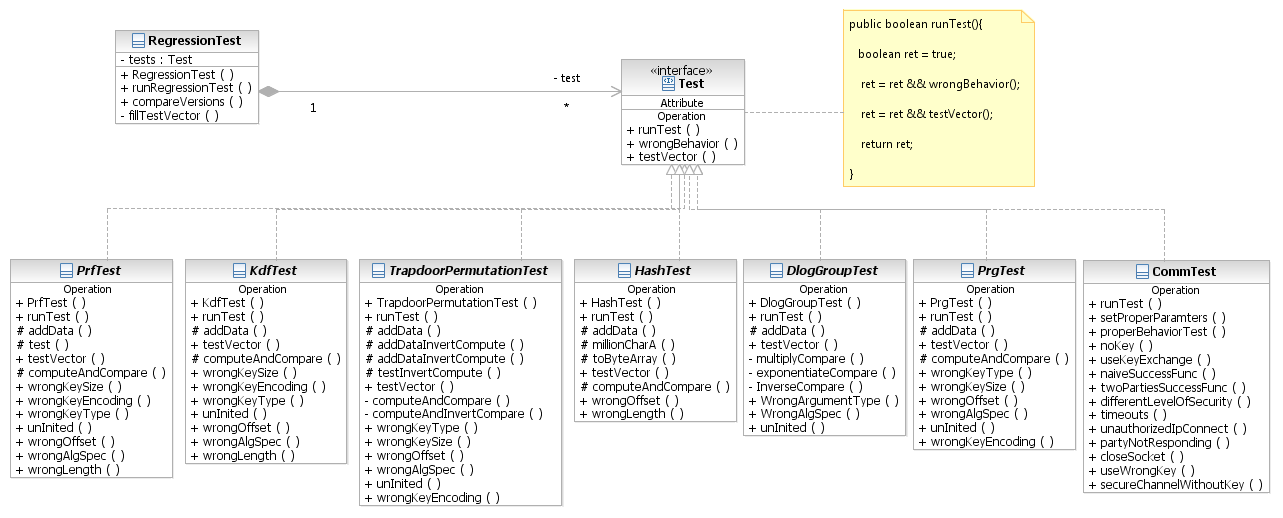
There will also be performance tests. We will compare similar algorithms from different sources. The decision which default implementation to supply will take into account the results of the performance tests.

## Testing general design

In general, we will have an interface Test (with a function runTest) that all tests must implement. This interface has three functions: wrongBehavior(), testVector() and runTest(). The function runTest used for the regression test and calls the two other functions.

We will also have a RegressionTest class that will hold a vector of Tests and will perform runTest on each instance in the vector. Also, each primitive interface will have an abstract class. For example, prfTest, prgTest and so on. The concrete tests will derive from these abstract classes that implement the Test interface.

Below is a class diagram containing the Test interface and the abstract classes. It also contains the commTest which is not abstract since we only have one communication feature.



## Testing Detailed Design

The testing module has a general structure. A test interface, family abstract classes (PrfTest, PrgTest and so on) and concrete derived classes that inherit from the relevant family abstract class.

The structure of the abstract family classes is as follows.

1. There is an instance of the related family (PRGTest holds a PseudorandomGenerator and PRFTest holds Pseudorandomfunction).
2. Has an inner class called TestData. This class is family depended. It always has input and expected output and other parameters such as key.
3. Holds a vector of TestData.
4. There are functions for testing wrong behavior, test vector and regression test.

Most of the functionality is done in the family abstract class on the underlying instance that is passed in the constructor of the derived class.

The function runTest of the abstract class calls the wrong behavior function and the test vector function. This function is called by the regression test.

Wrong behavior function is implemented in the abstract class and is common to all the derived classes. There will be some inner functions, each one checks different wrong case.

The test vector check is composed of two stages. First, the derived class adds to the vector of TestData of the abstract class the vector tests it wishes to test. Second, the abstract class checks for every instance in the vector that the actual computed result is equal to the expected output in the TestData. Since it holds the main family instance all derived classes must implement the related function that does the computation (if the abstract class does not implement it) and thus can be called generally in the abstract class.

A family that also implements an invert function (such as PRP and TrapdoorPermutation) also holds a vector of data for computing and inverting test. The functionality in the abstract family class goes over the vector and gets each input and other necessary auxiliary parameters and performs compute and invert. It then checks that the inverted result equals to the original input. Note that is some cases where we do not have a known vector test we can use this functionality to test the correctness of the algorithms.

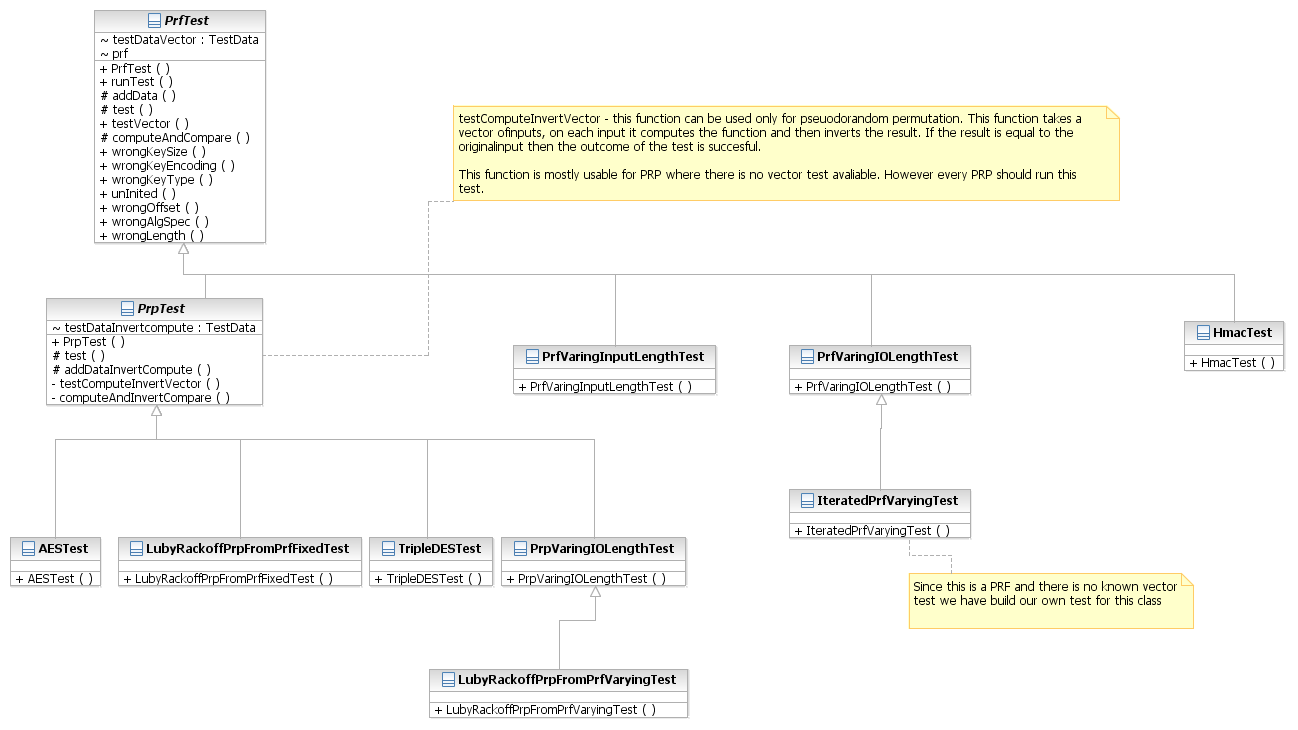
In the following family class diagrams one can see that the derived concrete classes only implement the constructor where they pass the concrete family object and fill the base vectors of data to be tested. Most of the algorithms have a general class (SHA1Test, HmacTest and so on) not depending on the wrapped library excluding the derived classes of TrapdoorPermutation and DlogGroup where there is a different class for each library. This is because there are concrete elements for each implementation so in order to create an element one should know which implementation he uses.

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

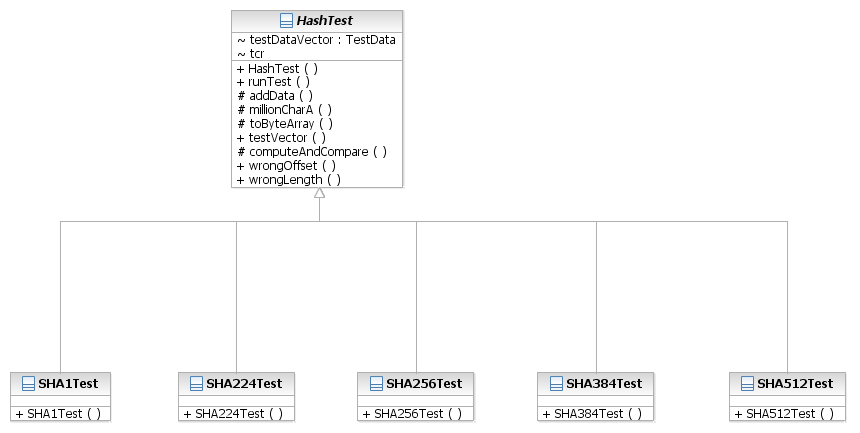
### PRG



### PRF



### Target Collision resistant Hash



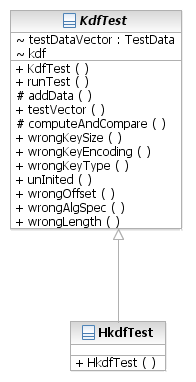
### Trapdoor permutation



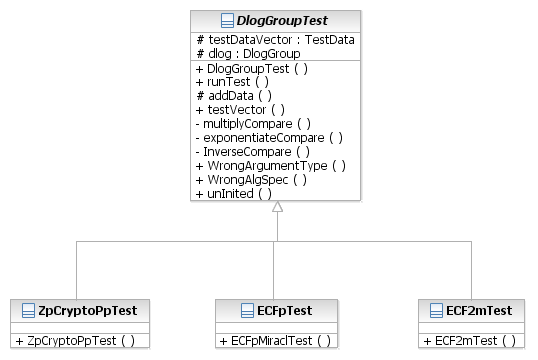
### Perfect universal hash



### Key derivation function



### Discrete logarithm



## 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 |  |  |  |
| Perfect 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 |  |  |  |

### Output file format

The file contains a line for each test. Each line lists the performed function, the algorithm that was checked and the test result (success/failure).

For example:

Compute and invert function on algorithm AES result in success

Compute function on algorithm AES result in success

Compute and invert function on algorithm DESede result in success

Compute function on algorithm SHA-224/HMAC result in success