F-Secure Corporation

F-Secure® Cryptographic Library™
FIPS 140-2 Validation Security Policy

Author: Alexey Kirichenko

Module version: 2.2.5, 2.2.7, 2.2.8, and 2.2.12 (Windows), 1.1.8, 1.1.9, 1.1.10, 1.1.12, and 1.1.15 (Solaris/Linux/AIX/HP-UX)

Document version:
F-Secure,FSCLM,FSCLM2_Security_Policy_level_1.rtf,00000009

Created: May 2003
Last modified: December 2006

Abstract: This document describes the F-Secure® Cryptographic Library™ Security Policy submitted for validation, in accordance with the FIPS publication 140-2, level 1.

Introduction

The F-Secure® Cryptographic Library™ for Windows is a software module, implemented as a 32-bit Windows™ ‘NT/2000/2003/XP/98/ME compatible DLL (FSCLM.DLL). The F-Secure® Cryptographic Library™ for Solaris, Linux, HP-UX, and AIX are software modules, implemented as shared libraries (LIBFSCLM.SO). These instances of the F-Secure® Cryptographic Library™ provide an identical set of cryptographic services to client applications, and in this document we refer to them all as “the Module”. The services are accessible for the client through an Application Programming Interface (API).

The Module was tested for FIPS 140-2 Level 1 requirements on Windows 2000, Windows 98, Windows XP Professional, Windows ME, Trusted Solaris 8 7/03, HP-UX B.11.11, AIX 5 and Red Hat Enterprise Linux (RHEL) 3 operating systems. Additionally, the Module may also be used with Trusted Solaris 8 4/01, Solaris 8 and 9, HP-UX 10, and Red Hat Enterprise Linux (RHEL) 4 running on a GPC without affecting the FIPS 140-2 validation as per FIPS 140-2 Implementation Guidance G.5. [1]

[1] The same physical binary runs on Solaris 8, Solaris 9, Trusted Solaris 8 4/01 and Trusted Solaris 8 7/03 operating systems. The same physical binary runs on both HP-UX 11.11 and HP-UX 10 operating systems and on both RHEL 3 and 4 operating systems.

Overall Design and Functionality

The Module is designed and implemented to meet the Level 1 requirements of FIPS publication 140-2 when running on a GPC under Windows NT/2000/2003/XP/98/ME, Trusted Solaris 8, Solaris 8 and 9, Linux RHEL 3 and 4, HP-UX 10 and 11, AIX 5 operating systems.

The Module is written in the “C” programming language, with some small performance-critical sections of the Windows and Linux versions written in the assembly language. The assembly language portions include code of the core transformation functions of certain symmetric ciphers and hash functions and a number of big integer arithmetic routines.

At the source code level, we use nearly an identical set of source files to build cryptographic libraries for a number of platforms, operating systems and linkage options. Almost all platform-dependent code is clearly separated into a small number of platform-specific files. The F-Secure Cryptographic Library for Windows is a dynamically linked module (DLL) for the user mode level of Windows 2000, Windows 2003, Windows XP, Windows 98, and Windows ME operating systems, the Solaris version is a shared library (Shared Object) for Sun Trusted Solaris 8, Solaris 8 and 9 operating systems, the Linux version is a shared library for Linux RHEL 3 and 4 operating system, the HP-UX version is a shared library for HP-UX 10 and 11 operating systems, and the AIX version is a shared library for AIX 5 operating system. Other examples of our cryptographic library “instances” are: kernel mode export driver and statically linked library for Windows NT/2000/2003/XP; kernel mode driver for Linux RHEL 4; DLL for Pocket PC 2002 and 2003 and Windows Mobile 2005; DLL for Symbian OS. (Note that only some of these instances were tested and validated for compliance with the FIPS 140 Level 1 requirements.)

The Module supports the FIPS approved AES, DES[2] and Triple DES (TDES), SHA-1, HMAC-SHA-1, SHA-256, HMAC-SHA-256, DSA, and RSA digital signing (PKCS#1) [3] algorithms. It also provides non-FIPS approved DES (CTR), Blowfish, RC2, CAST-128, MD5, RIPEMD-160, HMAC-MD5, Diffie-Hellman key agreement, RSA encryption (PKCS#1), and passphrase-based key derivation (PBKDF2 as specified in PKCS#5) algorithms. The Module implements a high-quality cryptographically strong Pseudorandom Number Generator (PRNG), which is compliant with the algorithm specified in Appendix 3.1 of the **FIPS PUB 186-2** document.

To defeat certain types of attacks based on timing analysis, the F-Secure Cryptographic Library employs blinding methods. Since the library is a software module that runs on a general-purpose computing systems, no other special effort was taken to mitigate side-channel attacks, in particular those based on power analysis and fault induction.

[2] DES is to be used only with legacy systems. For all other systems, F-Secure recommends using AES or Triple DES for encryption and decryption.

[3] Note that only the two RSA signing schemes defined in PKCS#1(RSA v.1.5 and PSS) can be used in the FIPS mode of operation. The signing scheme specified in RFC2409 is not a FIPS-approved service and cannot be used in the FIPS mode.

Use of an appropriate synchronization technique in the Module helps ensure that it functions correctly when simultaneously accessed by multiple threads[4]. We also want to note that performance considerations were an important criterion for the synchronization objects choice.

[4] By default, synchronization is disabled in the HP-UX version. That is done to ensure that the same physical binary would function correctly under various versions of the HP-UX operating system. We note that absolute majority of widely used applications running on HP-UX do not use multiple threads.

The Cryptographic Module and Cryptographic Boundary

In FIPS140-2 terms, the Module is a “multi-chip standalone module.” The F-Secure Cryptographic Library for Windows runs as a dynamically linked export library in any commercially available IBM Compatible PC under Windows 2000/2003/XP/98/ME Operating Systems (OS). The F-Secure Cryptographic Libraries for Solaris, Trusted Solaris, Linux, HP-UX, and AIX run as a shared library (Shared Object) in any commercially available computing system under the appropriate operating system. A “cryptographic boundary” for the Module is defined as those applicable software and hardware components internal to a host computing system that is running one of the supported operating systems.

The Windows OS separates user processes into memory spaces called “process spaces.” When a client process attaches the Module DLL, the DLL code is mapped to the address space of the process and a copy of the process-specific DDL data is allocated in the client process space. We informally call this mapping and process-specific data *an instance of the Module*. Multiple instances of the Module may reside inside a cryptographic boundary, however such instances are completely independent and each of them belongs to a single process. Any data passed between the Module and its client never leave the client’s process space and, therefore, never leave the cryptographic boundary. The OS is responsible for multitasking operations so that only one instance of the Module is active at any particular moment in time.
Furthermore, under Windows 2000 and Windows XP OS, any process space belongs to a single user and cannot be shared with any other user. The OS and the underlying central processing unit (CPU) hardware control access to each process space in such a way that other users cannot write to or read from the process’ memory.

In the case of the Solaris, Linux, HP-UX, and AIX versions, POSIX.1 standard mandates that processes must be kept separate and thus each process will have its own virtual memory address space that cannot be accessed from other processes running on the same system. When a client process loads the Module shared library, the operating system copies the relevant memory area to the client process address space. This is enforced by the operating system through the use of the hardware memory management unit (MMU), which causes an exception if a process tries to access memory outside of its allocated address space.

The module provides no physical security beyond that of the physical enclosure of a “hosting” computer system.

The assumption, which we make about the operating environment of the Module, is that it is installed, initialized and used by following the rules described below in section “Roles and Services.”

The Module was internally tested by the vendor (F-Secure Corporation) on the following computing platforms:

Hardware:	Dell OptiPlex GX 240 Personal Computer system
Processor:	Intel P4 1.6 GHz
Operating System:	Windows 2000 with service pack 3 and RH Enterprise Linux 3

Hardware: SunFire V210
Processor: Sun UltraSparc III
Operating System: Solaris 8

Hardware: HP Visualize B1000
Processor: HP 9000/785
Operating System: HP-UX B.11.00

Hardware: IBM pSeries C610
Processor: Power 3
Operating System: AIX 5L

Additionally, the Module was tested by a CMVP laboratory on the following computing platforms:

Hardware: Dell Optiplex GX 400 Personal Computer system
Processor: Intel P4 1.7 GHz
Operating System: Windows 2000 *with service pack 3 and Q326886 Hotfix*

Hardware: HP Visualize C200
Processor: HP-PA 2.0, 200Mhz
Operating System: HP-UX B.11.11

Hardware: IBM RS/6000
Processor: PowerPC POWER3 processor, 233MHz
Operating System: AIX 5

Hardware: Custom PC
Processor: AMD Athlon 800 MHz
Operating System: Windows 98

Hardware: Custom PC
Processor: AMD Athlon 800 MHz
Operating System: Windows ME

Hardware: Dell Latitude C400
Processor: Intel x86 1.2 GHz
Operating System: Windows XP Professional

Hardware: Custom PC
Processor: AMD Athalon 800 MHz
Operating System: Linux RHEL 3

Hardware: Sun Blade 100
Processor: 550Mhz UltraSPARC IIe CPU
Operating System: Trusted Solaris 8 7/03

Roles and Services

The F-Secure Cryptographic Library implements the following two roles: Crypto Officer role and User role. Since the Module is validated at security level 1, it does not provide an authentication mechanism. Hence roles are assumed implicitly based on the services that are performed.

The two roles are defined per the FIPS140-2 standard as follows:
A **User** is any entity that can access services implemented in the Module.
A **Crypto Officer** is any entity that can access services implemented in the Module, install the Module in a device, and configure the device to ensure proper operating of the Module in the FIPS 140-2 mode of operation.
There is no **Maintenance** role.

An operator performing a service within any role can read and write security-relevant data only through the invocation of a service by means of the Module API.

The following operational rules must be followed by any user of the Module:

1. Virtual memory of the computing system must be configured to reside on a local, not a network, drive.
2. All public keys entered into the Module must be verified as being legitimate and belonging to the correct entity by the client applications.
3. On Solaris, Linux, HP-UX, and AIX a special operating system device providing high quality randomness must be present on the computer. The Module attempts to read data from both the blocking /dev/random device, and the non-blocking /dev/urandom device to seed its PRNG.

It is a responsibility of the Crypto-Officer to configure the operating system to operate securely and, whenever it is necessary, to prevent remote login. Note that the Crypto Officer must have administrative privileges in the computer system being configured.

On Windows platforms, it is also recommended that the Crypto Officer sets value of "ClearPageFileAtShutdown" to 1 under "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management" key and sets "Interactive:Read" ACL for "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib" key (as opposed to "Everyone:Read" ACL) in the Registry.

The services provided by the Module to the User are effectively delivered through the use of appropriate API calls. In this respect, the same set of services is available to both the User and the Crypto Officer.

When a client process attempts to load an instance of the Module into memory, the Module runs an integrity test and a number of cryptographic functionality self-tests. If all the tests pass successfully, the Module makes a transition to "User Service" state, where the API calls can be used by the client to carry out desired cryptographic operations. Otherwise, the Module returns to "Uninitialized" state and the OS reports failure of the attempt to load it into memory.

The Module provides the following FIPS-approved services:
1. Cryptographic data hashing using FIPS PUB 180-2 SHA-1 and SHA-256.
2. Symmetric data encryption and decryption using FIPS PUB 197 AES, FIPS PUB 46-2 DES and TDES.
3. Random number generation using a software-based algorithm as specified in FIPS 186-2, *Digital Signature Standard (DSS*), Appendix 3.1.
4. MAC computation and verification using FIPS PUB 198 HMAC-SHA-1 and HMAC-SHA-256 algorithms (when key size is at least half of the algorithm output size).
5. Digital signature computation and verification using FIPS PUB 186-2 DSS (when key size, in bits, is a multiple of 64 and does not exceed 1024) and PKCS#1 RSA algorithms (RSA v.1.5 and PSS schemes).

Other non-approved services provided by the Module include:
6. Cryptographic data hashing using MD5 and RIPEMD-160 algorithms.
7. MAC computation and verification using HMAC-MD5 algorithm.
8. Symmetric data encryption and decryption using Blowfish, RC2, and CAST-128 block ciphers.
9. Passphrase-based key derivation (PBKDF2 as specified in PKCS#5) algorithm.
10. Key wrapping and unwrapping using PKCS#1 RSA encryption and decryption (RSA v.1.5 and OAEP schemes).
11. Diffie-Hellman key agreement.
12. Symmetric data encryption and decryption using DES (CTR).

Non-FIPS-approved services cannot be selected if the Module is operating in accordance with FIPS 140-2, that is, in the FIPS mode of operation. The exception to this are the Passphrase-based key derivation service based on the FIPS-approved SHA-1 hash function and HMAC-SHA-1 algorithm, key wrapping and unwrapping algorithm based on PKCS#1 RSA method, and Diffie-Hellman key agreement method. These services provide functionality that is not properly covered by any of the FIPS-approved algorithms at present time.

We note that the client must ensure that keys derived with PBKDF2 are only used for authentication purposes while in the FIPS mode. Such keys cannot be used for symmetric encryption/decryption when the Module is in the FIPS mode of operation.

Key Management

The Module implements a number of functions that are either used internally or exposed in the API to meet the FIPS140-2 Level 1 requirements for Key Management.

Key Generation

Keys for symmetric ciphers and HMAC algorithms can be generated by simply requesting the PRNG implemented in the Module to produce a desired number of bytes. The PRNG employs a FIPS-approved algorithm as specified in FIPS 186-2, *Digital Signature Standard (DSS*), Appendix 3.1. No other RNGs are used by the Module.

Services for generating DSA, RSA and Diffie-Hellman key pairs are also available. The FIPS-approved key generation method specified in FIPS 186-2 is used for DSA key pairs generation.

Intermediate key generation values are never output from the Module.

Key Distribution and Storage

The Module supports import and export of electronic keys in both encrypted and plaintext forms. It should be noted, however, that all keys are processed, stored, and used in the Module only on behalf of and for immediate use by a process, typically, an application program, that attaches an instance of the Module.

The Module can be used for electronic key distribution in the frames of a NIST-approved key distribution protocol and for implementing key exchange protocols. This usually involves symmetric ciphers, RSA encryption/decryption, Diffie-Hellman key agreement, and digital signing algorithms, all of which are provided by the Module. While in approved mode, RSA encryption/decryption can only be used for key transport.

At run-time, an application that uses the Module API calls to implement key distribution or key exchange mechanisms and protocols attaches an instance of the Module. Thus, all keys generated and/or otherwise processed by the Module reside in the application’s “process space”. This effectively means that the application program process has a full control over all such keys, and it is the responsibility of the application program developers to ensure FIPS140-2 compliance of protocols and algorithms they implement.

The Module does not provide long-term cryptographic key storage.

Zeroization of Keys

Keys and critical security parameters in the Module can be divided into two groups: those used by the Module internally and the ones that actually belong to its clients.

The Module takes care of zeroizing all its internal keys and critical security parameters (such as the PRNG internal state or various pre-computed values): (1) when those are not needed any more, (2) when the client process detaches the instance of the Module, and (3) when the Module enters the error state.

For the other group, when a client requests the Module to destroy a data object containing keys or critical security parameters, the Module always zeroizes all such data objects prior to freeing their memory. Also, the Module performs so-called “objects clean-up at exit.” When the client process is attempting to detach the instance of the Module, we check if there are any objects (e.g., cipher or HMAC contexts or private keys) allocated and not freed by the client, and we zeroize and free all such objects. This is especially important if a fatal error occurs in the Module, or the client does not have a chance to take proper care of cleaning up objects possibly containing sensitive information.

Protection of Keys

Keys created within or passed into the Module for one user are not accessible to any other user via the Module. It is a responsibility of its clients to protect keys exported from the Module and validate keys passed into the Module. To export private key data in plaintext form, the client has to pass appropriate values to <u>two</u> arguments on the private key export API function argument list. This serves as a double check and means that two independent actions of the client are required to let private key data be exported in plaintext form.

The Module takes care of never exposing its own internal keys and critical security parameters outside, and of zeroizing those prior to exiting or freeing corresponding portions of memory. In particular, we mention the PRNG state and intermediate generation values, whose disclosure or modification may compromise the security of the Module.

List of Keys stored in the module

Following keys are stored in the Module:

1. Keys for symmetric encryption/decryption algorithms:
    a. DES key
    b. Triple DES key
    c. AES key
    d. Blowfish key
    e. CAST-128 key
    f. RC2 key

2. Keys for asymmetric cryptographic algorithms:
    a. RSA public and private keys
    b. DSA public and private keys
    c. Diffie-Hellman public and private keys

3. Keys for HMAC methods:
    a. HMAC-SHA-1 key
    b. HMAC-SHA-256 key
    c. HMAC-MD5 key

4. Key for self-integrity test:
    a. HMAC-SHA-1 key

Out of the above keys, only the HMAC-SHA-1 key used for the self-integrity test is stored across power cycles. The rest of the keys are ephemeral keys, which are zeroized before the Module exits.

Module Interfaces

Being a software module, the F-Secure Cryptographic Library defines its interfaces in terms of the API that it provides. We define Data Input Interface as all those API calls that accept, as their arguments, data to be used or processed by the Module. The API calls that return, by means of return value or arguments of appropriate types, data generated or otherwise processed by the Module to the caller constitute Data Output Interface. Control Input Interface is comprised of the call used to initiate the Module and the API calls used to control the operation of the Module. Finally, Status Output Interface is defined as the API calls, which provide information about the status of the Module.

Self-Testing

The F-Secure Cryptographic Library implements a number of self-tests to check proper functioning of the Module. This includes power-up self-tests (which are also callable on-demand) and conditional self-tests.

Power-up Self-Testing

When an instance of the Module starts loading into memory, power-up self-testing is initiated automatically. It is comprised of the software integrity test, known answer tests of cryptographic algorithms, and pairwise-consistency test of DSA. If any of the tests fail, the Module returns to “Uninitialized” state and the OS reports failure of the attempt to load it into memory.
The following known answer tests are implemented in the Module:

- AES KAT
- DES KAT
- TDES KAT
- Blowfish KAT
- CAST-128 KAT
- SHA-1 KAT
- SHA-256 KAT
- HMAC-SHA-1 KAT
- MD5 KAT
- RSA signing/verification tests
- PRNG KAT
- PRNG Statistical Tests

Note: No DSA KAT is implemented. Instead the pairwise-consistency test is performed for every DSA key pair generated. In particular, this test runs at the power-up time for a fixed DSA key.

The software integrity test computes DAC value by applying the HMAC-SHA-1 method, FIPS 198, to data of all the relevant sections of disk image of the Module. The test fails if the DAC value computed on the disk image of the Module does not match the original value computed on the Module by a special utility at the vendor’s site (F-Secure Corporation) and stored in a special place inside the Module.

On-Demand Self-Testing

The Module exports an API routine, “fsclm_Selftest”, which can be called to initiate the power-up self-tests. If any of the tests fail, the Module enters the error state. This error state is unrecoverable; upon entering it, the Module stops providing cryptographic services to the client.

Conditional Self-Testing

This includes continuous PRNG testing. The very first output block generated by the PRNG is never used for any purpose other than initiating the continuous PRNG test, which compares every newly generated block with the previously generated block. The test fails if the newly generated PRNG output block matches the previously generated block. In such a case, the Module enters the unrecoverable error state.

Pairwise Consistency Self-Testing

The test is run whenever private key is generated or imported by the Module. The private key structure of the Module always contains either the data of the corresponding public key or the information sufficient for computing the corresponding public key. Thus, generating or importing private key is equivalent to generating or importing key pair.

Depending on key type, the test generates and verifies digital signatures for a fixed message under private and public keys of the key pair being tested and/or applies encryption and decryption operations to the message. If the test fails for a generated key pair, the Module enters the unrecoverable error state. If an imported key pair does not pass the test, the corresponding function returns an appropriate error code but the Module does not enter the error state. This reflects the fact that a corrupted or inconsistent imported key pair does not mean malfunction of the Module.

Mitigation of Attacks Based on Timing Analysis

In Timing Analysis based attacks, the attacker attempts to collect and analyze information about the time required by a cryptographic module to carry out certain mathematical operations involved into cryptographic processing. This may help the attacker reveal partial or even full information about keys and other critical security parameters. The Module employs so-called blinding techniques to level timing variation of operations that may become a target of attacks based on Timing Analysis. In particular, the client can choose to use blinding in decryption and signing operations with RSA private keys and in shared secret computation of Diffie-Hellman key agreement protocol. In fact, in the RSA case, blinding is used by default.

The essence of the employed blinding method is in introducing random, unpredictable for the attacker, values into mathematical computations. While this makes operations marginally slower, measuring timings becomes practically useless for the attacker. We notice that pre-computation can in certain cases significantly ease the negative performance consequences of the use of blinding.

List of the API Functions, Operating Modes, Important Technical Considerations

In this section, we briefly describe the services that the Module provides and related security and usage considerations. In order to guarantee secure and robust functioning of the Module, it is important that the clients follow our recommendations as fully and precisely as possible.

The following list presents the Module API functions split into a number of groups in accordance with their functionality.

Mode of operation and Information functions

fsclm_GetModuleVersion
This routine provides the callers with the Module version information.

fsclm_GetModuleMode
This routine returns the current mode of operation of the Module.
The F-Secure Cryptographic Library supports two modes of operation: FIPS 140 mode and non-FIPS mode. Only FIPS-approved algorithms are available to the caller in FIPS 140 mode. Any attempt to use non-FIPS-approved algorithms in FIPS 140 mode results in an appropriate error code returned by the Module. It is a responsibility of client application developers to design their products in such a way that they function properly in the both modes of operation. We recommend avoiding schemes and protocols, which are based on non-selectable non-FIPS-approved algorithms in any part.

fsclm_SetModuleMode
This routine sets the mode of operation of the Module. The two options are:
FSCLM_MODE_NONFIPS - all methods included in the Module are available to the caller;
FSCLM_MODE_FIPS140 - only FIPS-approved methods are available to the caller.
Use of "fsclm_SetModuleMode" makes it easy to ensure that non-FIPS-approved algorithms are unavailable, no matter what cryptographic services the client application requests from the Module.

fsclm_GetModuleStatus
This routine returns the current status of the Module. There are five states defined in the Module Finite State Machine (FSM):
FSCLM_STATUS_UNINITIALIZED
FSCLM_STATUS_SELF_TESTING
FSCLM_STATUS_USER_SERVICE
FSCLM_STATUS_UNLOADING
FSCLM_STATUS_ERROR

fsclm_GetErrorCode
This function returns "fatal" error code if the Module is in the error state, or FSCLM_ERROR_FATAL_NONE otherwise.

Symmetric encryption functions

The Module implements a number of symmetric ciphers, including FIPS-approved AES, DES, and TDES modes. In the code, we use a layered approach based on the internal “cipher API”, which makes it very easy to exclude existing or add new ciphers if desired. The cipher modes of operation are implemented as a generic layer, so each newly included cipher can immediately be used in any of the supported modes. (The Module supports the standard ECB, CBC, CFB, and OFB modes as well as Counter and IWEC modes.)

All the encryption and decryption functions support “in-place” operations, which means that the same buffer may be used as both source and destination parameters.

fsclm_CipherInfo
Provides information about the specified cipher. This makes it possible to learn if the cipher is supported by the Module, if it is FIPS-approved, and what key and block sizes are supported for it.

fsclm_CipherAlloc
Allocates and initializes the cipher context object for the specified cipher in the specified mode of operation and with the specified key. Any allocated cipher object must eventually be freed by calling "fsclm_CipherFree". The Module takes care of never exposing contents of cipher objects outside and of proper zeroizing their memory when appropriate.

fsclm_CipherFree
Zeroizes and frees the memory of the specified cipher object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_CipherReset
This resets the given cipher object so that it would look like a newly allocated and initialized one. The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_CipherEncrypt
This encrypts the given input buffer and writes the resulting ciphertext to the given output buffer. Encryption mode and other parameters are taken from the given cipher context object.

fsclm_CipherDecrypt
This decrypts the given input buffer and writes the resulting plaintext to the given output buffer. Mode of operation and other parameters are taken from the given cipher context object.

fsclm_CipherEncryptIV
This encrypts the given input buffer and writes the resulting ciphertext to the given output buffer. The only difference between this routine and "fsclm_CipherEncrypt" is that the latter takes IV/counter information from the cipher object and updates it appropriately, while the former uses "iv" value passed to it as a parameter and updates that value (leaving IV/counter information in the cipher object intact).

fsclm_CipherDecryptIV
This decrypts the given input buffer and writes the resulting plaintext to the given output buffer. The only difference between this routine and "fsclm_CipherDecrypt" is that the latter takes IV/counter

information from the cipher object and updates it appropriately, while the former uses "iv" value passed to it as a parameter and updates that value (leaving IV/counter information in the cipher object intact).

fsclm_CipherSetIV
This sets encryption or decryption IV/counter value in the specified cipher object. This value will then be used for the subsequent encryption ("fsclm_CipherEncrypt") or decryption ("fsclm_CipherDecrypt") operation respectively.
Note that the same cipher object can be used for both encryption and decryption operations, thus we maintain separate encryption and decryption IV/counter information in the cipher object.

fsclm_CipherGetIV
This copies the current encryption or decryption IV/counter value in the specified cipher object to the caller-supplied buffer.

fsclm_CipherComputeIV
Certain modes of operation of block ciphers make use of counter value. In such modes, processing of a particular block of input depends on the initial value of counter and index (or offset) of the block. (Two examples supported by the Module are Counter and IWEC modes.) If you want to perform encryption or decryption operation starting with the *n*-th block, you would need to know the corresponding counter value, and this is what this routine helps you do: given the initial counter value and the block index, it computes and writes to the caller-supplied buffer the counter value for the block.
Note that counter-based modes provide you with a random read-write access to large streams of encrypted data, the property that CBC, CFB, and OFB modes do not enjoy.

fsclm_CipherEncryptBuffer
This routine performs one-pass encryption of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:
fsclm_CipherAlloc
fsclm_CipherEncryptIV
fsclm_CipherFree

fsclm_CipherDecryptBuffer
This routine performs one-pass decryption of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:
fsclm_CipherAlloc
fsclm_CipherDecryptIV
fsclm_CipherFree

Hash functions

The Module currently implements three hash functions: FIPS-approved SHA-1 and SHA-256, and non-FIPS-approved MD5. In the code, we use a layered approach based on the internal “hash API”, which makes it very easy to exclude existing or add new hash functions if desired.

fsclm_HashInfo
Provides information about the specified hash function. This makes it possible to learn if the hash function is supported by the Module, if it is FIPS-approved, and what its output (digest) and block sizes are.

fsclm_HashAlloc
Allocates and initializes the hash context object for the specified hash function. Any allocated hash object must eventually be freed by calling "fsclm_HashFree".
Hash objects may contain confidential information. The Module takes care of never exposing contents of hash objects outside and of proper zeroizing their memory when appropriate.

fsclm_HashFree
Zeroizes and frees the memory of the specified hash object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_HashReset
This resets the given hash context object so that it would look like a newly allocated and initialized one. It is useful when you want to use the same hash function for computing hash values (also called *digests*) of multiple data blocks.
The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_HashUpdate
This updates the given hash context with the given input.
When you need to compute digest of a data stream which comes in a number of portions (or when you want to split a very long stream in a number of pieces), you can simply feed such portions to "fsclm_HashUpdate" one by one. The resulting digest value will be identical to what you would get if passing the entire stream as a single buffer.
Note that in order to obtain digest value of your data, any sequence of calls to "fsclm_HashUpdate" must eventually be followed by a call to "fsclm_HashFinal".

fsclm_HashFinal
This function completes computation of hash value of a data stream, which has been processed by calls to "fsclm_HashUpdate" function. The resulting digest is written to a caller-supplied buffer.
Note that after "fsclm_HashFinal" has been called for a hash object, the object should not be used for any further operations until you call "fsclm_HashReset" for it. After resetting, you may start computation of hash value for a new data stream.

fsclm_HashOfBuffer
This routine computes digest of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:

fsclm_HashAlloc
fsclm_HashUpdate
fsclm_HashFinal
fsclm_HashFree

HMAC functions

The Module clients can use HMAC methods based on any hash function that is implemented in the Module. By specifying the ID of a hash function of your choice, you fully specify the HMAC algorithm that you want to use. To obtain information about parameters of a particular HMAC algorithm, simply call "fsclm_HashInfo" for the corresponding hash function.

fsclm_HMACAlloc

Allocates and initializes the context object for the HMAC algorithm based on the specified hash function, and with the specified key. Any allocated HMAC object must eventually be freed by calling "fsclm_HMACFree".
The Module takes care of never exposing contents of HMAC objects outside and of proper zeroizing their memory when appropriate.

fsclm_HMACFree

Zeroizes and frees the memory of the specified HMAC object. This routine is always available to the caller, even if the Module is in the error state.

fsclm_HMACReset

This resets the given HMAC context object so that it would look like a newly allocated and initialized one. It is useful when you want to use the same HMAC function, possibly with a different key, for computing message authentication code (MAC) values of multiple data blocks.
The "reset" operation also zeroizes all remnants of the previous processing.

fsclm_HMACUpdate

This updates the given HMAC context with the given input.
When you need to compute MAC of a data stream which comes in a number of portions (or when you want to split a very long stream in a number of pieces), you can simply feed such portions to "fsclm_HMACUpdate" one by one. The resulting MAC value will be identical to what you would get if passing the entire stream as a single buffer.
Note that in order to obtain MAC value of your data, any sequence of calls to "fsclm_HMACUpdate" must eventually be followed by a call to "fsclm_HMACFinal".

fsclm_HMACFinal

This function completes computation of MAC value of a data stream, which has been processed by calls to "fsclm_HMACUpdate" function. The resulting MAC is written to a caller-supplied buffer.
Note that after "fsclm_HMACFinal" has been called for an HMAC object, the object should not be used for any further operations until you call "fsclm_HMACReset" for it. After resetting, you may start computation of MAC value for a new data stream (possibly using a different key).

fsclm_HMACOfBuffer

This routine computes MAC value of a given buffer, which can be a useful shortcut in certain cases. It encapsulates a number of other API calls to save the application developer effort. This call is equivalent to the following sequence:

```
 fsclm_HMACAlloc
 fsclm_HMACUpdate
 fsclm_HMACFinal
```

fsclm_HMACFree

PRNG functions

The PRNG implemented in the Module is based on hybrid architecture. It uses a one-way output function on top of the well-known “entropy pool” scheme. The design is FIPS-compliant as the output algorithm is the one specified in Section 3.1, Appendix 3 of **FIPS PUB 186-2** document, with the function G constructed from the SHA-1 as specified in Section 3.3, Appendix 3 of the same document.

The PRNG is initialized when the Module gets loaded into memory. During the initialization phase, various system and hardware parameters and statistics are collected and mixed in the PRNG pool with the SHA-1 transform function to achieve a good diffusion of “entropy” bits. Seeding/reseeding code for each supported platform resides in the respective platform-specific source file.

fsclm_PrngDeepPoll

Invokes platform-specific “deep” polling for entropy (i.e., hard-to-predict bits) to achieve good-quality seeding of the PRNG. This deep polling gets called automatically occasionally during the entire lifetime of the Module. Also, the function is called at the PRNG initialization time.
The main purpose of this function is to help maintain the PRNG pool in a state, which is infeasible to guess for the adversary.

fsclm_PrngAddNoise

This exclusive-ORs bytes from the given buffer with the PRNG pool content and serves the purpose of adding unpredictability to the PRNG state. (We leave it up to the client whether to use this function or not as the automatic PRNG seeding in the Module should be good enough to prevent the adversary from guessing the PRNG state or any of the output values.)
The exclusive-OR operation cannot force the PRNG in a weaker state because it obviously cannot reduce the pool data entropy.

fsclm_PrngMixPool

Mixes (i.e., cryptographically processes) the PRNG pool. The mixing operation is based on the SHA-1 transform function. It provides good “entropy” diffusion and is irreversible.
This function gets called automatically at the initialization time and then regularly during the entire lifetime of the Module.

fsclm_PrngGetBytes

This routine writes to the caller-supplied buffer the requested number of PRNG-produced bytes. Although what the generated bytes will be used for is entirely up to the caller, we recommend calling this function if you need to generate:

- any keying material (in both symmetric and asymmetric settings)
- IV or initial counter values used in many popular methods (e.g., modes of operation of block ciphers)
- padding bytes for various cryptographic schemes
- random nonces and challenges required in many cryptographic protocols (e.g., authentication protocols)
- salts to be combined with passphrases in passphrase-based key derivation algorithms
- random values for probabilistic cryptographic algorithms (e.g., signing with DSA)

We stress that it is a responsibility of the client to protect bytes provided by the Module PRNG (in particular, from being exposed to the adversary).

fsclm_PrngGetParameters
Fills in the fields of a caller-supplied structure with the current values of the PRNG object parameters.

fsclm_PrngSetParameters

This function lets the caller change the PRNG parameters used in the algorithms for generating output and updating the PRNG pool.

Mathematical functions

Mathematical functions are extensively used in the asymmetric key cryptographic methods implemented in the Module. In many cases, however, the user may want to have direct access to the mathematical functionality. The mathematical API described below covers a number of basic operations with large integers as well as certain more advanced methods and format conversion routines. In the Windows and Linux versions, those low-level operations, which are the most important performance-wise, are written in the assembly language and tightly optimized.

fsclm_BigIntAllocate
This allocates a new “Big Integer” object and returns its handle to the caller. Note that the value of a freshly allocated object is not defined.

fsclm_BigIntFree
This function zeroizes and frees memory of the given “Big Integer” object. The object handle becomes invalid after this operation.

fsclm_BigIntReset
This is used to reset the given object to make it appear a newly allocated one. Memory that the object value was stored in will be zeroized and freed.

fsclm_BigIntAssign
This function assigns value of one given “Big Integer” object to the other one.

fsclm_BigIntSetInt
This assigns value of the given argument of “int” type (in the C language terms) to the given “Big Integer” object.

fsclm_BigIntSetUInt
This assigns value of the given argument of “unsigned int” type (in the C language terms) to the given “Big Integer” object.

fsclm_BigIntPowerOfTwo
This function sets value of the given “Big Integer” object to the specified power of 2.

fsclm_BigIntHighestBitIndex
This function retrieves position of the highest “1” bit in the binary representation of the given “Big Integer” object.

fsclm_BigIntSetBit
This sets bit of the given "Big Integer" in the specified position to the given value.

fsclm_BigIntGetBit
This function retrieves value of the bit in the specified position of the binary representation of the given "Big Integer" object.

fsclm_BigIntFirstSetBitIndex

This function retrieves position of the lowest “1” bit in the binary representation of the given “Big Integer” object.

fsclm_BigIntCmp
This is used to compare values of the two given "BigInt" objects.

fsclm_BigIntCmpInt
This is used to compare values of the given "BigInt" object and (C language) integer.

fsclm_BigIntCmpUInt
This is used to compare values of the given "BigInt" object and (C language) unsigned integer.

fsclm_BigIntCmpAbs
This is used to compare absolute values of the two given "BigInt" objects.

fsclm_BigIntNeg
This function changes sign of the given "BigInt" object.

fsclm_BigIntAbs
This function replaces value of the given "BigInt" object with its absolute value.

fsclm_BigIntGetUInt
This function “extracts” value of the least significant “word” (which is of the unsigned integer type in the C language terms) of the given "BigInt" object. The operation is defined only for non-negative numbers.

fsclm_BigIntAdd
This routine computes sum of values of the two given "BigInt" objects. We support in-place operations, that is, the “destination” object (the one that accepts the sum value) can coincide with any of the summands.

fsclm_BigIntAddInt
This routine computes sum of values of the given "BigInt" object and (C language) integer. We support in-place operations, that is, the “destination” object (the one that accepts the sum value) can coincide with the “BigInt” summand.

fsclm_BigIntAddUInt
This routine computes sum of values of the given "BigInt" object and (C language) unsigned integer. We support in-place operations, that is, the “destination” object (the one that accepts the sum value) can coincide with the “BigInt” summand.

fsclm_BigIntSub
This routine computes difference of values of the two given "BigInt" objects. We support in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntSubInt

This routine computes difference of values of the given "BigInt" object and (C language) integer. We support in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with the “BigInt” operand.

fsclm_BigIntSubUInt
This routine computes difference of values of the given "BigInt" object and (C language) unsigned integer. We support in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with the “BigInt” operand.

fsclm_BigIntMod
Given a pair of "BigInt" objects, this routine computes modular residue of the first one modulo the second one. It supports in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntModUInt
Given a "BigInt" object, this computes its modular residue modulo the given (C language) unsigned integer.

fsclm_BigIntModPowerOfTwo
This routine computes modular residue of the given "BigInt" object modulo the specified power of 2. This is equivalent to extracting the specified number of the least significant bits in the binary representation of the "BigInt" value. The function supports in-place operations.

fsclm_BigIntMul
This routine computes product of values of the two given "BigInt" objects. We support in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntMulInt
This computes product of values of the given "BigInt" object and (C language) integer. It supports in-place operations.

fsclm_BigIntMulUInt
This computes product of values of the given "BigInt" object and (C language) unsigned integer. It supports in-place operations.

fsclm_BigIntSquare
This function computes square of value of the given "BigInt" object. It supports in-place operations.

fsclm_BigIntModMul
This is a odular multiplication function. It computes product of values of the two given "BigInt" objects modulo another given "BigInt" object. We support in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntDiv
This function implements "division with remainder" operation. It supports in-place operations.

fsclm_BigIntDivInt
This function implements "division with remainder" operation in the case when the divisor is a (C language) integer. It supports in-place operations.

fsclm_BigIntDivUInt
This function implements "division with remainder" operation in the case when the divisor is a (C language) unsigned integer. It supports in-place operations.

fsclm_BigIntSHL
This routine implements "shift to the left" by the specified number of bits operation for value of the given "BigInt" object. This is equivalent to multiplying the given “BigInt” by an appropriate power of 2. We support in-place operations.

fsclm_BigIntSHR
This routine implements "shift to the right" by the specified number of bits operation for value of the given "BigInt" object. This is equivalent to dividing the given “BigInt” by an appropriate power of 2. We support in-place operations.

fsclm_BigIntInvMod
Given a pair of "BigInt" objects, this function computes multiplicative inverse of the first one modulo the second one. If the inverse doesn't exist, because GCD of the operands is not 1, the GCD value will be returned to the caller. The function supports in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntGCD
Given a pair of "BigInt" objects, this function computes Greatest Common Divisor (GCD) of their values. It supports in-place operations.

fsclm_BigIntGCDExt
Given a pair of "BigInt" objects, this function computes Greatest Common Divisor (GCD) of their values and its representation as a linear combination of the given values. It supports in-place operations.

fsclm_BigIntModExp
Given "base", "exponent" and "modulus" "BigInt" objects, this function performs modular exponentiation operation. It supports in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntModExpUInt
Given (C language) unsigned integer "base", and "exponent" and "modulus" "BigInt" objects, this function performs modular exponentiation operation. It supports in-place operations, that is, the “destination” object (the one that accepts the resulting value) can coincide with any of the operands.

fsclm_BigIntIsPrime
This routine implements a primality test, that is, it can be used for determining whether a given number is (probably) prime or composite. Word "probably" appears in the previous sentence because the final

part of the implemented primality testing is probabilistic. The routine can also test if the given number is a safe prime. (Number P is called a safe prime if it is prime and (P - 1)/2 is also prime).

fsclm_BigIntGetRandom
This function generates a random non-negative integer in the interval [0, $2^{numBits}$ - 1]. The module PRNG is used in the generation, so produced numbers should be unpredictable and suitable for use in any cryptographic setting.

fsclm_BigIntGetRandomPrime
This function generates a random (probable) prime or safe prime in the interval [0, $2^{numBits}$ - 1]. The module PRNG is used in the generation, so produced numbers should be unpredictable and suitable for use in any cryptographic setting.

fsclm_BigIntGetBufferSize
This routine returns size of the buffer (in bytes) the caller has to allocate to export the given number with the specified options. It can be used prior to calling "fsclm_BigIntToBuffer" or "fsclm_BigIntExport" to determine the required buffer size.

fsclm_BigIntFromBuffer
This function imports “BigInt” in raw binary. It assumes the given buffer contains absolute value of the number, and the sign information is supplied as a separate argument. A number of formatting options are supported.

fsclm_BigIntToBuffer
This function exports “BigInt” in raw binary. The routine writes absolute value of the number to the given buffer, and the sign information is saved in a separate argument. A number of formatting options are supported.

fsclm_BigIntImport
This function imports “BigInt” in raw binary. It is similar with "fsclm_BigIntFromBuffer" but assumes that the highest bit of the number being imported is the sign bit, that is, value of 1 indicates that the number is negative. A number of formatting options are supported.

fsclm_BigIntExport
This function exports “BigInt” in raw binary. It is similar with "fsclm_BigIntToBuffer" but saves the sign information in the highest bit of the number representation in the buffer, that is, value of 1 indicates that the number is negative. A number of formatting options are supported.

fsclm_BigIntStringIn
This routine accepts a character array and converts its content into a "BigInt" object. Only four options are supported for "representation base": 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

fsclm_BigIntStringOut
This routine converts value of the given "BigInt" object into a character string in the specified base. As above, only four options are supported: 2 (binary), 8 (octal), 10 (decimal), 16 (hexadecimal).

Asymmetric Key functions

Routines that belong to this group provide digital signing, asymmetric encryption, and key exchange functionality that is extensively used in many popular cryptographic protocols. In particular, the Module supports DSA, the RSA-based schemes defined in the PKCS#1 document, and Diffie-Hellman key exchange methods. Also, the Module implements functionality for importing and exporting public and private key data in a number of popular formats, in particular, X.509 for public and PKCS#8 for private keys.

fsclm_PKTypeInfo
This routine provides information about the specified asymmetric key type. This makes it possible to learn if the key type is supported by the module, which encryption and signature schemes are supported for keys of this type (by default), and the key type-specific size restrictions in FIPS and non-FIPS modes.

fsclm_PrivateKeyGenerate
This function allocates and initializes the "private key" object of the specified type. If the caller provides no data to be used in the key generation process, the module generates a fresh key. Otherwise, if the caller-supplied data are found consistent and suitable, only missing key values are generated. It is possible to fully specify all the required key information.
Any allocated private key object must eventually be freed by calling "fsclm_PrivateKeyFree".
Private keys typically contain confidential information. The Module takes care of never exposing contents of private key objects outside and of proper zeroizing their memory when appropriate.

fsclm_PrivateKeyFree
This function zeroizes and frees the memory of the specified private key object. It is always available to the caller, even if the Module is in the error state.

fsclm_PrivateKeyGetData
This lets the caller retrieve various information of the specified private key object.

fsclm_PrivateKeySign
This routine digitally signs the specified message with the given private key.

fsclm_PrivateKeyDerivePublic
This function allocates and initializes a “public key" object which is a counterpart of the given private key.

fsclm_PrivateKeySetSchemes
This lets the caller control which signature and encryption schemes are supported for the specified private key object.

fsclm_PublicKeyDefine
This function allocates and initializes a "public key" object of the specified type. The caller must provide data sufficient to fully define the new public key object. Any public key object must eventually be freed by calling "fsclm_PublicKeyFree".

fsclm_PublicKeyFree
This function zeroizes and frees the memory of the specified public key object. It is always available to the caller, even if the Module is in the error state.

fsclm_PublicKeyGetData
This lets the caller retrieve various information of the specified public key object.

fsclm_PublicKeyVerifySignature
This verifies validity of the given digital signature for the given message with the given public key. The signature scheme and scheme-specific parameters identical to those used when generating the signature should be passed to this routine.

fsclm_PublicKeyClone
This function allocates and initializes a "public key" object which is a copy of the given public key.

fsclm_PublicKeySetSchemes
This lets the caller control which signature and encryption schemes are supported for the specified public key object.

fsclm_PKVerifyKeyPair
This function tests if the given private and public key objects are parts of the same key pair.

fsclm_PKProtectionInfoFree
This zeroizes and frees the memory of the specified protection info structure allocated by the Module (“fsclm_PrivateKeyImport”). This routine is always available to the caller, even if the module is in the error state.

fsclm_PublicKeyExport
This exports data of the given public key in the specified format to the given byte array.

fsclm_PublicKeyImport
This function imports public key data provided in the given buffer. If successful, it creates a new public key object and returns the format information to the caller.

fsclm_PrivateKeyExport
This exports data of the given private key in the specified format to the given byte array. To protect the exported data, a properly initialized protection info structure must be supplied.

fsclm_PrivateKeyImport
This function imports private key data provided in the given buffer. If successful, it creates a new private key object and returns the format and protection information to the caller.

fsclm_PrivateKeyChangePassphrase
Given a buffer with exported private key data, this function changes the passphrase used to protect the data (keeping intact the data and preserving the export format).

fsclm_PKEncryptMaxInputSize

This informs the caller of the maximum possible size of input to encryption operation for the given public key and encryption scheme. It should be called prior to calling "fsclm_PublicKeyEncrypt" to determine the upper bound on acceptable input size.

fsclm_PublicKeyEncrypt
This encrypts the given message with the given public key.

fsclm_PrivateKeyDecrypt
This decrypts the given encrypted message with the given private key. The encryption scheme and scheme-specific parameters identical to those used when encrypting the message should be passed to this routine.

fsclm_DHGroupGenerate
This function allocates and initializes a "DH Group" object. If the caller provides no data to be used in the generation process, the module generates a fresh group. Otherwise, if the caller-supplied data are found consistent and suitable, only missing values are generated. It is possible to fully specify all the required group information.
Any "DH Group" object must eventually be freed by calling "fsclm_DHGroupFree".

fsclm_DHGroupFree
This zeroizes and frees the memory of the specified DH Group object. The caller should keep in mind that if there exist any DH Pair objects associated with the given group, no action will be taken and this call will return an appropriate error code.

fsclm_DHGroupGetData
This lets the caller retrieve various information of the specified DH Group object.

fsclm_DHGroupPrecomputePairs
This function can be used to precompute a number of DH pairs (private/public values) for the given DH Group object. Note that this does not result in any new DH Pair objects. However, availability of precomputed DH pairs makes any subsequent calls to "fsclm_DHPairGenerate" run very fast, which could be quite useful in many applications. A typical use case would be running this precomputation routine in a background (as a separate thread).

fsclm_DHPairGenerate
This function allocates and initializes a "DH Pair" object associated with the specified DH Group object. Any "DH Pair" object must eventually be freed by calling "fsclm_DHPairFree".

fsclm_DHPairFree
This zeroizes and frees the memory of the specified DH Pair object. This routine is always available to the caller, even if the module is in the error state.

fsclm_DHPairGetPublicValue
This lets the caller retrieve public value of the specified DH Pair object. It is primarily intended for use in the "static" case, when the same DH pair is used in multiple key exchange operations.

fsclm_DHPairComputeSharedValue

Given a DH Pair object and a public DH value (of the other party), this function computes the resulting shared value. The caller can also request to run validity check of the given public value. This makes the operation slower but may be important to detect a potential attempt of the other party to recover the caller's DH pair private value (mainly a concern in the "static" case).
If requested, the Module will use blinding in this operation.

fsclm_DLParamsInit
This function initializes public "FSCLM_DLParameters" structure (allocated by the caller). It must be called for any "FSCLM_DLParameters" structure prior to passing it to FSCLM API functions.
Any initialized "FSCLM_DLParameters" structure must eventually be zeroized by calling "fsclm_DLParamsZeroize".

fsclm_DLParamsZeroize
This zeroizes and frees fields of the given "FSCLM_DLParameters" structure.

fsclm_DLParamsGenerate
This function generates parameters that are used in Discrete Log based methods, in particular, DSA and Diffie-Hellman. If the caller provides no data to be used in the generation process, the module generates a fresh parameter set. Otherwise, if the caller supplied data are found consistent and suitable, only missing values are generated. It is possible to fully specify all the required values.

fsclm_DLParamsVerify
This function verifies the given Discrete Log parameters set for consistency and correctness. What exactly is checked depends on the values passed by the caller. If the FIPS 186 parameters generation method or its generalization specified in RFC 2631 was used for the given parameters set, and the caller supplies appropriate intermediate generation values, this function also verifies the generation method compliance.

fsclm_DSSKeyStructInit
This function initializes the specified "FSCLM_DSSPrivateKey" or "FSCLM_DSSPublicKey" structure (allocated by the caller). It must be called for any "DSS key" structure prior to passing it to FSCLM API functions. Any initialized "DSS key" structure must eventually be zeroized by calling "fsclm_DSSKeyStructZeroize".

fsclm_DSSKeyStructZeroize
This zeroizes and frees fields of the given "FSCLM_DSSPublicKey" or "FSCLM_DSSPrivateKey" structure.

fsclm_DSSPrecomputeSignValues
This function can be used to precompute a number of signing values for the given DSS private key. Availability of precomputed signing values noticeably improves signing operation performance and could be quite useful in certain applications. A typical use case would be running this precomputation routine in a background (as a separate thread).

fsclm_RSAKeyStructInit
This function initializes the specified "FSCLM_RSAPrivateKey" or "FSCLM_RSAPublicKey" structure (allocated by the caller). It must be called for any "RSA key" structure prior to passing it to

FSCLM API functions. Any initialized "RSA key" structure must eventually be zeroized by calling "fsclm_RSAKeyStructZeroize".

fsclm_RSAKeyStructZeroize
This zeroizes and frees fields of the given "FSCLM_RSAPublicKey" or "FSCLM_RSAPrivateKey" structure.

fsclm_SetRSABlindingFlags
The Module is capable of employing blinding technique in operations with RSA private keys (decryption, signing) to prevent certain side-channel attacks, esp. timing and power analysis ones. By default, we apply blinding for all operations with all RSA private keys. However, this hurts performance, and if the caller is confident that side-channel attacks is not a concern in a given context and improving performance is important, blinding can be turned off by passing an appropriate argument to this routine.

fsclm_GetRSABlindingFlags
This function retrieves the current blinding settings for RSA decryption and signing.

fsclm_RSAPrecomputeBlindingValues
This function can be used to precompute a number of blinding values for the given RSA private key. Availability of precomputed blinding values improves signing and decryption operation performance (if blinding is enabled) and could be quite useful in certain applications. A typical use case would be running this precomputation routine in a background (as a separate thread).

Other functions

fsclm_Selftest
Calling this routine makes the Module run a number of self-tests. This on-demand self-testing includes self-integrity test, Known Answer Tests of cryptographic algorithms, and, optionally, the set of PRNG statistical tests (as specified in the FIPS 140-2 document). If any of the tests fail, the Module enters the error state, which means that its cryptographic services become unavailable to the clients. To use the services again, the user will usually need to restart the client application or reload the Module in some other way.

fsclm_DeriveSymmetricKey
This routine implements the passphrase-based key derivation function specified in PKCS#5 (PBKDF2). The implementation uses HMAC-SHA1 as a PRF.
The two main goals of this key derivation algorithm are:
- preventing the adversary from compiling a universal dictionary of passphrases and precomputing the corresponding keys (achieved by using so-called “salt”, whose presence in the algorithm results in a very large number of keys that correspond to each passphrase)
- making exhaustive search attacks much more computationally expensive, which is especially important in the case of “weak” passphrases (achieved by iterating the key derivation function many times and recursively)

We stress that it is a responsibility of the client to protect keys derived by this routine (in particular, from being exposed to the adversary). This is a non-Approved service.

fsclm_OverwriteMemory
This function can be used for overwriting a given block of memory with a bit stream that enjoys good statistical properties (i.e., appears as a Binary Symmetric Source output).
We use it internally to overwrite portions of memory that may contain confidential data.
Also, this function can (and should !) be used instead of the PRNG to produce random-looking bits when we do not care about “cryptographic quality”. A typical example is generating “witnesses” for probabilistic primality testing.

fsclm_GetBase64Length
Clients should call this routine prior to calling “fsclm_EncodeBase64” to determine size of the buffer that Base64 encoded data will be written to. Values of the encoding option arguments passed to “fsclm_GetBase64Length” must be identical to the ones subsequently passed to “fsclm_EncodeBase64”.

fsclm_EncodeBase64
Given an input buffer, this routine encodes the data in Base64 format. The client can specify desired line length and ending for the encoded byte stream.

fsclm_DecodeBase64
This routine transforms a given Base64 encoded byte stream to the original (raw) form.

Detailed description of the Module API can be found in the Module public header files (FSCLM.H, FSCLM_MP.H, FSCLM_PK.H).

We conclude this section by listing a number of recommendations aimed at helping the Module clients avoid security-related and technical problems when implementing data security products.

- Prior to freeing any memory blocks that may contain critical security parameters or other confidential data, take care of zeroizing them properly. When you free an object allocated by the Module (for example, symmetric cipher context, Diffie-Hellman pair, or private key) by calling an appropriate FSCLM API function, the Module zeroizes the object memory. The client applications are responsible for zeroizing any other memory blocks, in particular, those intermediate variables containing keying or otherwise confidential data.

- It is a responsibility of the clients to ensure they work with cryptographic objects allocated by the Module in a multi-threading safe way. Please keep in mind that the Module provides no synchronisation for accessing such objects concurrently by multiple threads of the client applications.

- While the Module is designed to prevent the use of non-FIPS-approved methods in the FIPS mode of operation, there are three exceptions to that rule. Specifically, the client can use longer than 1024-bit keys with DSA, can apply RSA encryption to arbitrary data blocks (not only those containing keying information), and can sign with RSA using non-FIPS-approved “RSA Plain” signing scheme (specified in RFC2409 - IKE) even if the Module is in the FIPS mode of operation. To maintain full FIPS 140-2 compliance, the client applications have to ensure that their use of DSA and RSA methods meets the appropriate NIST requirements.