Secuware Security Framework - Crypt4000 Module Security Policy

18-11-2008

Version 4.0

SECUWARE S.L.

Plaza Pablo Ruiz Picasso, S/N
Picasso Tower, 14th floor
28020, Madrid

Telephone:	+34 915649149
FAX	+34 915629697

Table of contents

1 Introduction

1 This document is the FIPS 140-2 security policy for the **Secuware Security Framework – Crypt4000 Module** (SCM) software object module to meet FIPS 140-2 level 1 requirements.

2 This Security Policy details the secure operation of the **Secuware Security Framework – Crypt4000 Module v 3.0** developed by SECUWARE as required in Federal Information Processing Standards Publication 140-2 as published by the National Institute of Standards and Technology (NIST) of the United States Department of Commerce.

1.1 Audience

3 This document is required as a part of the FIPS 140-2 validation process. It describes the **Secuware Security Framework – Crypt4000 Module** in relation to FIPS 140-2 requirements. The companion document **Secuware Security Framework – Crypt4000 Module User Guide** is a technical reference for developers using and installing the SCM.

1.2 Document organization

4 This Security Policy document is one part of the FIPS 140-2 Submission Package. The Submission Package contains:

- Security Policy: this document
- Algorithm certificates: see 2.5 Approved cryptographic algorithms
- Functional specification and design documentation: see sections 2.3 Functional Specification and 2.4 Cryptographic module design of this document.
- User guide: SCM Crypto Officer and User Guidance reference [GUI] (summarised in this document)
- Finite state machine model: see section 4 Finite state machine model of this document.
- Configuration Item list: see section 8.1.3 Configuration Item List of this document.
- Source code listing.

5 This document outlines the functionality provided by the module and gives high level details on the means by which the module satisfies FIPS 140-2 requirements.

1.3 References

[GUI]	Secuware Security Framework – Crypt4000 Module User Guide, v 3.0 June2008
[SP800-38A]	Recommendations for Block Cipher Modes of Operation
[F197]	FIPS 197 Advanced Encryption Standard (AES), Nov 26, 2001

2 Module Specification

6 The **Secuware Security Framework – Crypt4000 Module**, referred elsewhere in this document as SECUWARE Cryptographic Module (SCM), is defined as a specific discrete unit of binary object code (the “FIPS Object Module”) generated from a specific set of C language source files embedded within an object distribution.

7 This object code in an isolated and separated form which consists of a single file, is used as a library to provide a cryptographic API (Application Programming Interface) to any external applications which statically links with it, embedding it into them.

8 The Module implements the AES algorithm.

2.1 The FIPS object module

9 The Implementation Under Test (IUT) is a function library implementing crypto services which is delivered to the final user as a software cryptographic object Module, running on Windows operating system in a General Purpose Computer.

10 The generation of the SCM and the documented process for creating it, was developed to satisfy FIPS 140-2 requirements.

11 Although the SCM is software, the physical embodiment will be a general purpose computer which consists of multiple components, considered to be a multichip standalone module by FIPS140-2.

12 The logical cryptographic boundary for the SCM is the discrete block of object code containing the machine instructions and data generated from the SCM FIPS source, which will be allocated continuously in a main memory address space, as used by the calling application.

13 The physical cryptographic boundary contains the general purpose computing hardware of the system executing the application. This system hardware includes the central processing unit(s), cache and main memory (RAM), system bus, and peripherals including disk drives and other permanent mass storage devices, network interface cards, keyboard and console and any terminal devices.

2.2 Ports and interfaces

14 The module provides a logical interface via an Application Programming Interface (API). The API provides functions that may be called directly by the referencing application.

15 The API interface provided by the Module is mapped onto the FIPS 140-2 logical interfaces: data input, data output, control input, and status output. Each of the FIPS 140-2 logical interfaces relates to the module's callable interface, as follows:

– Data input: input parameters to all functions that accept input from IT entities acting either as Crypto Officer or User entities.

– Data output: output parameters from all functions that return data as arguments or return values to Crypto Officer or User IT entities.

– Control input: all API function input into the module by the Crypto Officer and User IT entities

– Status output: information about status that may be queried by Crypto Officer or User IT entities, using the appropriate function.

16 The API functional specification explaining the logical interfaces is included below.

2.3 Functional Specification

17 The following functions represent the logical interfaces available for an external application linked to the SCM:

– **int SCM_Init()**

  This function calls the **SCM_Self_test()** function and initialized the module in FIPS mode of operation. It returns the status of the module by calling the **SCM_Show_status()** function.

– **void SCM_KeySetup(LPBYTE Key)**

With a given pointer to the key, this function initializes the AES sub-keys that are kept internal to the Module. The subkeys length must be 240 bytes. The key must enter into the Cryptographic module encrypted by the user performing a simple AES ECB mode encrypt with a predefined key which will be given to the user and hardcoded into the module.

LPBYTE *Key*: pointer to where the encrypted AES Key is allocated, length must be of 32 bytes.

– **void SCM_Cipher (DWORD counter, LPBYTE buffer)**

This function receives the buffer that the programmer wants to encrypt with the subkeys previously generated, saving the encrypted block in the same buffer. Counter modifies the result according to AES-CTR.

DWORD *counter*: sequential number in a larger data structure than buffer which the programmer wants to encrypt/decrypt according to AES-CTR.

LPBYTE *buffer*: input/output buffer for the encrypted/decrypted result. Its length must be 16Bytes.

– **int SCM_Show_status ()**

This function returns an integer showing the status of the cipher in the form 2 * system status + key fixed.

- **int system_status**

Represents a control variable which value is 1 if the system is ready to operate and a different value if the system is not ready to operate:

0. INITIAL_STATE
1. OK (ready to operate)
2. INT_ERROR (integrity error)
3. KAT_ERROR
4. SELF_TESTING

- **int key_fixed**

Represents a control variable which value is 1 if the keys are set up, and 0 in the other case.

– **void SCM_Terminate (LPBYTE Key, DWORD *counter)**

This function must be called at the end of the encryption / decryption or in case of recovery, doing a zeroization in the memory address that allocates the Key, the counter and the generated subkeys that are kept internal to the Module.

LPBYTE *Key*: pointer to where the AES Key is allocated.

DWORD * counter: pointer to the AES-CTR counter.

– **void SCM_Self_test()**

Runs the self tests on demand, updating the system status variable.

2.4 Cryptographic module design

18 The basic design of the **SCM**, collects three subsystems with their corresponding *interfaces* properly defined, as well as the *interaction* that identifies the reason of the communication between the subsystems.

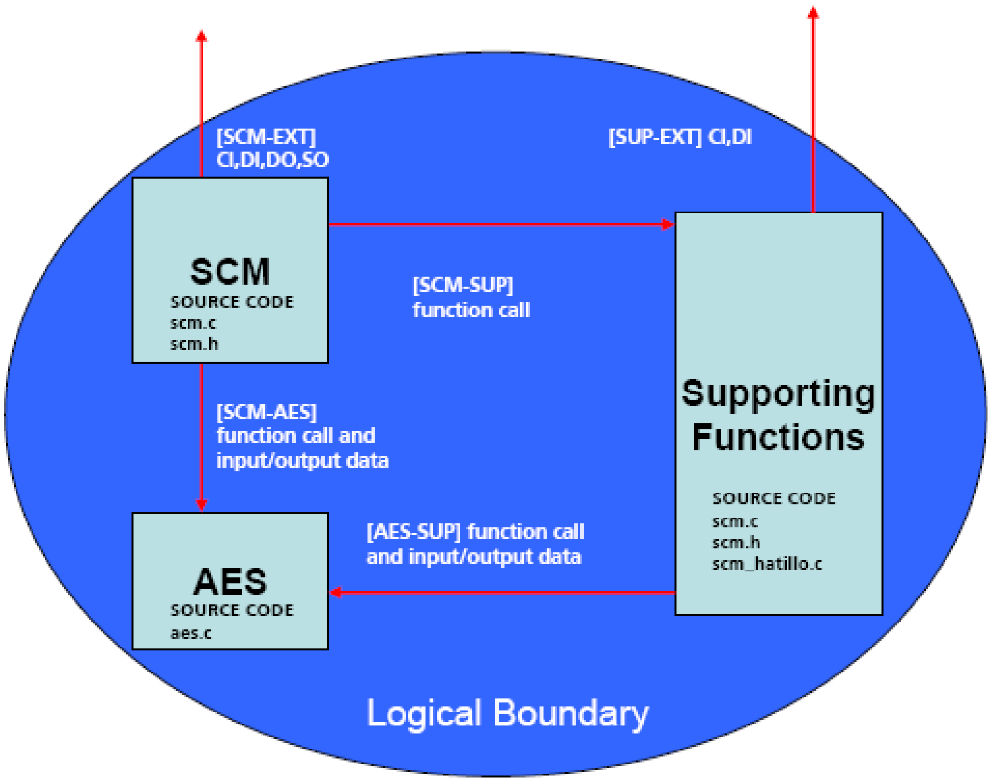


Figure 1: SCM logical Boundary

Note (as defined in 2.2 Ports and interfaces) for external interfaces:

- **DI: Data input**
- **DO: Data output**
- **CI: Control input**
- **SO: Status output**

Al interfaces are detailed in 2.4.2 External Interfaces Specification

19 The module consists of a single object code component (**scm_v40.o**) which is built from the following files: scm.h, scm.c, aes.c, scm_hatillo.c, associated with each of the subsystems within the logical boundary as illustrated in Figure 1. Applications that use the module must link the module object code **scm_v40.o** in to use the services it provides.

20 This section describes the modules collection of the SCM product and their interfaces.

2.4.1 Modules description

SCM Subsystem

21 The SCM subsystem implements three different functionalities:

– The SCM initialization.

– The proper encrypt/decrypt operation calls.

– The system status check.

22 This subsystem provides an interface to the AES subsystem. All encrypt/decrypt operation is performed via SCM subsystem.

23 The initialization functionality is in charge of calling the mandatory power-up self-tests, including the integrity and KAT algorithm tests, and calling AES sub-keys derivation. For this purpose, the initialisation receives a 256 bits encrypted key as an input parameter and prior to the key expansion it will ECB-decrypt the entered key using a hardcoded key, and returns the sub-keys derived from the entered key.

24 As part of the initialization process, the self-test functionality performs power-up self-tests to ensure that the system is not on error state:

- Integrity of the object cryptographic module (SCM) in the runtime executable application at runtime.

- KAT over the AES module in order to check whether the AES-ECB decrypt algorithm, and the AES-CTR algorithm behaves as expected.

25 Once the initialization functionality has finished as expected - power-up self test performed with OK result and AES sub-keys derived-, the AES-CTR encrypt/decrypt functionality is available.

26 Each encryption/decryption operation requires carrying out a system status checking prior to its execution. On error state, the crypto operations will not be available and the outputs will be inhibited.

27 The encryption/decryption operation calls are made using **SCM_Cipher()** function.

AES Subsystem

28 The AES subsystem implements four different functionalities:

- The sub-keys derivation for encryption.
- The sub-keys derivation for decryption.
- The CTR encrypt/decrypt operation.
- The ECB decrypt operation.

29 The subsystem implements AES sub-keys expansion ([F197] section 5.2). For this purpose, it receives a 256 bits key as an input parameter, which the sub-keys used by the encryption algorithm are derived from. A self test must have been performed before starting with AES sub-keys derivation for encryption.

30 The process of generating subkeys works according to the AES key expansion method ([F197] section 5.2), where 256 bits key is expanded to a 240 Bytes expanded key, applying several operations a number of times.

31 The AES module implementation, programmed in C, works in AES-CTR mode where the size of the buffer is 16 Bytes.

32 The module will always use 256 bits keys, so the number of AES rounds is fixed. This implementation has a great performance and very low memory consumption.

33 In each AES encryption round, the subBytes, shiftRows, addRoundKey and mixColumns operations are done in an optimized way.

34 Once the AES subsystem has finished its execution, the zeroization functionality will erase the AES Key, the counter and the generated AES sub-keys.

35 This subsystem also implements the sub-keys derivation algorithm for decryption and the AES-ECB decryption algorithm. Those functions are called from the SCM subsystem for decrypting the key used by the user for the AES-CTR functionality.

36 AES Operation mode

- The crypto module operates in an AES-CTR (Advanced Encryption Standard Counter) confidentiality mode when the buffer size is fixed to 16 Bytes. The underlying block cipher algorithm of the mode, AES, is a **FIPS approved algorithm**, a symmetric key algorithm which operates on fixed-length groups of bits. The algorithm is implemented using C code and it may be used to encrypt and decrypt a buffer.
- The Counter (CTR) mode is a confidentiality mode that features the application of the forward cipher to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext, and vice versa. The sequence of counters must have the property that each block in the sequence is different from every other block ([SP800-38A]).

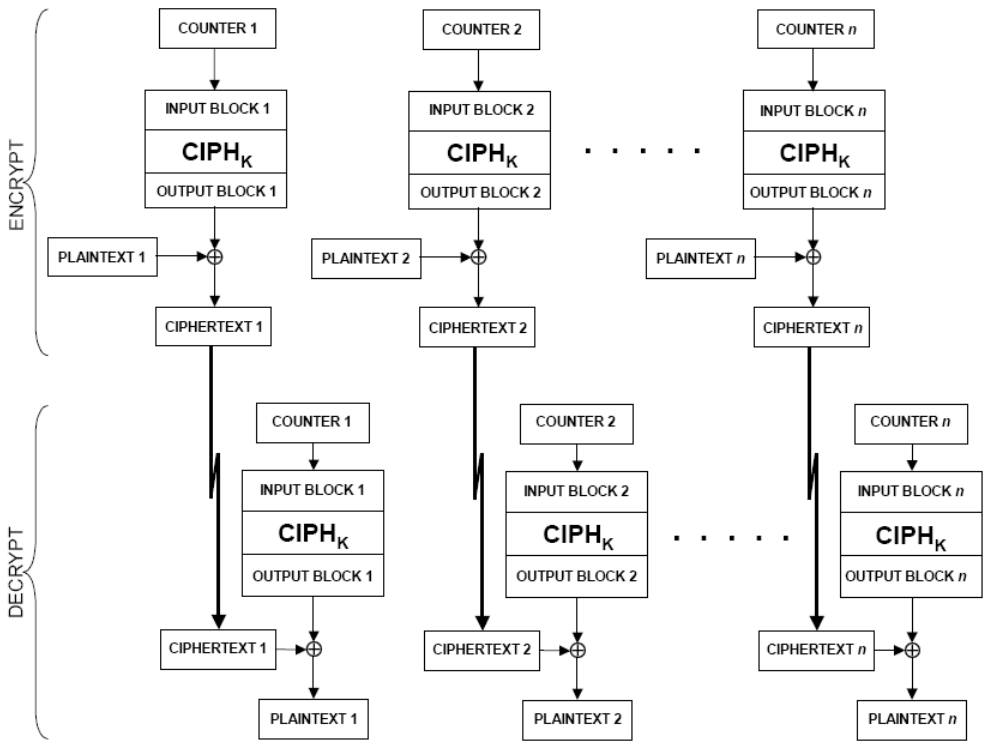


Figure 2: CTR schema

Supporting Functions Subsystem

37 The following supported functions are provided:

SelfTest

The self-test function (see also section 7 Self-Tests) performs self-tests (on module power-up or on user demand) to ensure that the AES module behaves properly. *Power-up self-tests* are performed when the cryptographic module starts its execution.

- If the cryptographic module fails the power-up self-tests, it does not progress in its normal operation, staying in a non operative status.
- If the cryptographic module fails a self-test, the module enters a recovery state and updates the system status indicator which can be read via the status output interface. The cryptographic module does not perform any cryptographic operations while in the recovery state.

The following values are taking into account (see section 4 Finite state machine model) when checking the availability of the cryptographic operations due to a possible error:

System Status	Key Fixed	Functionality
INITIAL_STATE	TRUE	Unreachable state
	FALSE	Crypto operations not available
OK	TRUE	Crypto operations available
	FALSE	Crypto operations not available
INT_ERROR	TRUE	Recovery required Crypto operations not available
	FALSE	Crypto operations not available
KAT_ERROR	TRUE	Recovery required Crypto operations not available
	FALSE	Crypto operations not available
SELF_TESTING	TRUE	Crypto operations not available
	FALSE	Crypto operations not available

The power-up tests are initiated automatically without requiring operator intervention. When the power-up tests are completed, the results are returned via the "status output" interface. If the tests succeed, the AES Module enters in a healthy state and begins its normal operation; otherwise the module does not progress in its normal operations staying in a non operating state.

In addition the cryptographic module allows operators to initiate the tests on demand for periodic testing of the module.

The following self tests (for both power-up and on demand) are provided:

- Software integrity test over the object cryptographic module in the runtime executable application at runtime. The integrity tests are performed using a HMAC-SHA1 digest calculated over the final executable.

- Cryptographic algorithm test: Known Answer Tests (KATs) are tests where a cryptographic value is calculated and compared with a stored previously determined answer. KAT for the AES-CTR Algorithm encrypts with a 256 bit key a known vector and then the obtained value is compared with the known result. KAT for the AES-ECB Algorithm, decrypts with a 256 bit key a known vector and then the obtained value is compared with the known result.

The variables system_status and key_fixed, implemented in the source code, are in charge of determining the System Status. The following table shows the correspondence with the design and the finite state machine system status control:

System_status	Key_fixed	**FSM States**
INITIAL_STATE	TRUE	Unreachable state
	FALSE	NO OPERATIVE
OK	TRUE	OPERATIVE
	FALSE	HEALTHY
INT_ERROR	TRUE	RECOVERY
	FALSE	ERROR
KAT_ERROR	TRUE	RECOVERY
	FALSE	ERROR
SELF_TESTING	TRUE	SELF_TEST
	FALSE	SELF_TEST

- When **system_status** is on INT_ERROR, the integrity Self-Test has failed.

- When **system_status** is on KAT_ERROR, the KAT Self-Test has failed.

- When **system_status** is on SELF_TESTING, the Self-Test is being performed.

- On a recovery required status, the SCM will finish any crypto operation and the module will have to be again initialized.

- While in no operative status or during the self-test execution, the SCM encryption/decryption functionality is not available and the output interface is inhibited.

- The status of the system may be queried by the user using an API function (**SCM_Show_status ()**).

Zeroization

Zeroization must be called by the user IT entity of the API, to erase the keys and the counter in the following cases:

- The AES module has finished

- A recovery is required

The function uses a technique called zeroization, which fills keys structure and the counter with 0's. User keys established externally by the API user IT entity which are the input parameter for the key setup functionality in order to derive the AES sub-keys, are also zeroized: when the module is initialized, the main AES key, the counter and the derived sub-keys are established and reside in RAM, until they are zeroized by the user application at the end of the session by calling **SCM_Terminate.** The user application must call **SCM_Terminate** at the end of each session, regardless if it completed successfully or if processing errors where encountered.

2.4.2 External Interfaces Specification

SCM Subsystem external observable Interface [SCM-EXT]

38 SCM provides to the user application the following functions with their associated parameters as specified in section 2.3 Functional Specification:

- **int SCM_Init()**

- **void SCM_KeySetup(LPBYTE Key)**

- **void SCM_Cipher (DWORD counter, LPBYTE buffer)**
- **int SCM_Show_status ()**

Supporting Functions Subsystem external observable Interface [SUP-EXT]

39 The Supporting Functions Module provides to the user application the following functions with their associated parameters as defined in section 2.3 Functional Specification:

- **void SCM_Self_test()**
- **void SCM_Terminate (LPBYTE Key, DWORD *counter)**

2.4.3 Inter-subsystems Interfaces Specification

SCM Subsystem – Supporting Functions Subsystem Interface [SCM-SUP]

40 The Supporting Functions Module provides to the SCM module the following function with their associated parameters as specified in section 2.3 Functional Specification for power up self test purposes:

- **void SCM_Self_test()**

SCM Subsystem - AES Subsystem Interface [SCM-AES]

41 The AES module provides to the SCM Module the following functions with their associated parameters:

- **int AesFastKeySetupEnc256(u32 sk[240], const u8 Key[32])**

  With a given pointer to the key, this function initializes the AES sub-keys at the specified memory position. If the operation succeeds, the function return 1 else it returns 0.

  u32 *sk[240]*: pointer to where the programmer wants to save the generated subkeys, length must be 240 bytes

  const u8 *Key[32]*: pointer to where the AES Key is allocated, length must be 32 bytes.

- **void AesFastCipherBlock_sk(const u32 sk[240], u32 counter, u8 buffer[16])**

  This function receives the pointer to the AES subkeys, and the buffer that the programmer wants to encrypt or decrypt, saving the encrypted block in the same buffer.

  const u32 sk[240]: pointer to where the programmer has saved the generated subkeys.

  u32 counter: sequential number in a larger data structure than buffer which the programmer wants to encrypt/decrypt.

  u8 buffer[16]: buffer where the programmer wants to save the encrypted/decrypted result.

- **int AesFastKeySetupDec256(u32 sk[240], const u8 Key[32])**

  With a given pointer to the key, this function initializes the AES sub-keys for decryption at the specified memory position. If the operation succeeds, the function return 1 else it returns 0.

  u32 *sk[240]*: pointer to where the programmer wants to save the generated subkeys, length must be 240 bytes

  const u8 *Key[32]*: pointer to where the AES Key is allocated, length must be 32 bytes.

- **void StandAloneECB(const u32 sk[240], u8 buffer[16])**

  Execute standalone AES-ECB encryption.

  const u32 sk[240]: pointer to the generated subkeys.

  u8 buffer[16]: buffer to encrypt.

- **void StandAloneECBDeciph(const u32 sk[240], u8 buffer[16])**

  Execute standalone AES-ECB decryption.

  const u32 sk[240]: pointer to the generated subkeys.

  u8 buffer[16]: buffer to decrypt.

AES Subsystem – Supporting Functions Subsystem Interface [AES-SUP]

42 The AES module provides to the Supporting Functions Module the following functions with their associated parameters as specified in **SCM Module - AES Module Interface** section:

- **int AesFastKeySetupEnc256(u32 sk[240], const u8 Key[32])**
- **void AesFastCipherBlock_sk(const u32 sk[240], u32 counter, u8 buffer[16])**

2.5 Approved cryptographic algorithms

43 The Module supports the following FIPS approved cryptographic algorithms: Advanced Encryption Standard (AES – FIPS 197, 256 key length), symmetric key block cipher algorithm validated under the CAVP with certificate number #792 as published in the on-line AES algorithm validation list in http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html#792.

44 For the integrity tests purposes, SCM applies a Key-hash Message Authentication Code (HMAC – FIPS 198 Key-hash Message Authentication Code with supporting SHA-1 implementation) validated under the CAVP with certificate number #513 as published in the on-line HMAC algorithm validation list in http://csrc.nist.gov/groups/STM/cavp/documents/mac/hmacval.html#513.

45 The SHA-1 implementation is validated under the CAVP with certificate number #905 as published in the on-line SHA-1 algorithm validation list in http://csrc.nist.gov/groups/STM/cavp/documents/shs/shaval.htm#905.

2.6 Non-Approved cryptographic algorithms

46 There are no Non-Approved cryptographic algorithms in SCM.

2.7 Approved mode of operation

47 A single initialization call, **SCM_Init ()**, is required to initialize the SCM for operation in the only implemented FIPS 140-2

approved mode: AES-CTR. All services are performed in this approved mode.

48 SCM can only be operated in AES-CTR FIPS Approved mode; after the module initialization, it will be operating in this approved mode (see paragraph 36 AES Operation mode).

3 Roles, Services and Authentication

3.1 Roles

49 The User and Crypto Officer roles are implicitly assumed by any entity that can access services implemented in the Module. In addition, the Crypto Officer role can install and initialize the Module (not to be confused with **SCM_Init** API functionality, see section 8.2 Initialization, Start-up and Operation); this role is implicitly entered when installing the module or performing system administration functions on the host operating system as described in the following table:

Role	Authorized Services
User	All the services except secure installation, initialization and start-up
Crypto Officer	All the services including secure installation, initialization and start-up

3.2 Services

50 The operational services provided by the SCM are listed in the following table. All operational services may be performed in both User and Crypto Officer roles. Non operational services (secure installation, initialization and start-up services as described in section 8.2 Initialization, Start-up and Operation) may only be performed by in the Crypto Officer role.

Roles	Service	CSP	Algorithm	API function	Access
User Crypto Officer	Initialization	None	NA	SCM_Init	X
User Crypto Officer	Terminate	symmetric key, subkeys and counter	Zeroization	SCM_Terminate	RWX
User Crypto Officer	Symmetric Encryption/ Decryption	subkeys and counter	AES-256	SCM_Cipher	RWX
User Crypto Officer	Key setup	symmetric key and subkeys	NA	SCM_Keysetup	RWX
User Crypto Officer	Show status	None	NA	SCM_Show_status	X

User Crypto Officer	Self Test (includes integrity, known answer tests)	HMAC-SHA-1 key	HMAC-SHA-1 (for integrity)	SCM_Self_test	X

51 The services inputs and the services outputs are defined in the functional specification.

3.3 Operator Authentication

52 The **Cryptographic Module** does not provide identification or authentication mechanisms that would distinguish between the two supported roles. These roles are implicitly assumed by the services that are accessed. As a library and as allowed by FIPS 140-2 level 1, the SCM does not support user identification or authentication for those roles.

4 Finite state machine model

53 This section describes the Finite State Machine (FSM) model for an application utilizing the SCM FIPS Object Module. Figure 3 represents a finite state diagram showing the states and transitions between states. Anytime the SCM is in one and only one state.

4.1 Diagram

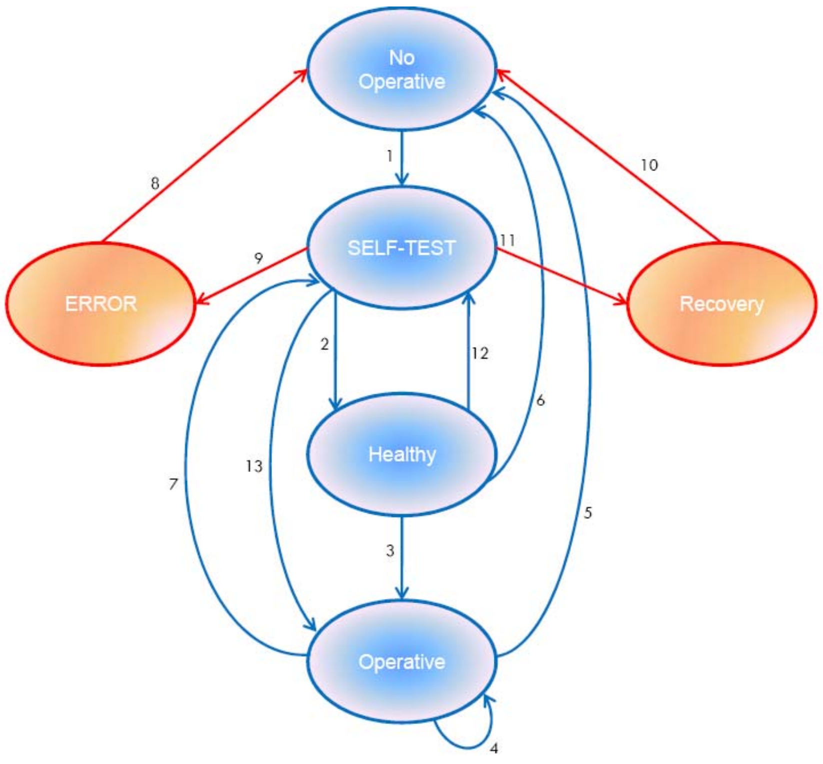


Figure 3: Finite State Machine Diagram

Transition Number	Initial State	Final State	Input Events	Output Events
1	NO OPERATIVE	SELF-TEST	SCM_Init	system_status = SELF_TESTING and key_fixed = NO
2	SELF-TEST	HEALTHY	system_status = OK and key_fixed = NO	system_status = OK and key_fixed = NO
3	HEALTHY	OPERATIVE	SCM_KeySetup	system_status = OK and key_fixed = YES
4	OPERATIVE	OPERATIVE	SCM_Cipher or	system_status = OK and

			SCM_KeySetup	key_fixed = YES
5	OPERATIVE	NO OPERATIVE	SCM_Terminate	system_status = INITIAL_STATE and key_fixed = NO
6	HEALTHY	NO OPERATIVE	SCM_Terminate	system_status = INITIAL_STATE and key_fixed = NO
7	OPERATIVE	SELF-TEST	SCM_Self_test	system_status = SELF_TESTING and key_fixed = YES
8	ERROR	NO OPERATIVE	SCM_Terminate	system_status = INITIAL_STATE and key_fixed = NO
9	SELF-TEST	ERROR	(system_status = INT_ERROR or system_status = KAT_ERROR) and key_fixed = NO	(system_status = INT_ERROR or system_status = KAT_ERROR) and key_fixed = NO
10	RECOVERY	NO OPERATIVE	SCM_Terminate	system_status = NO OPERATIVE and key_fixed = NO
11	SELF-TEST	RECOVERY	(system_status = INT_ERROR or system_status = KAT_ERROR) and key_fixed = YES	(system_status = INT_ERROR or system_status = KAT_ERROR) and key_fixed = YES
12	HEALTHY	SELF-TEST	SCM_Self_test	system_status = SELF_TESTING and key_fixed = NO
13	SELF-TEST	OPERATIVE	system_status = OK and key_fixed = YES	system_status = OK and key_fixed = YES

Table 1: State Transitions Table

4.2 States and transitions

States

54 **No Operative**:

The user application is running but the cryptographic module does not provide any cryptographic service. Only when performing a call to the function SCM_Init (which contains the self testing) the transition is provoked and the next state (Self-Test state) is reached.

55 **Self-Test**:

The user application is performing a self-test operation. There is no operation which can be performed during self-test execution. If the operation succeeds the system_status will be updated to OK else the system_status will change its value to INT_ERROR or KAT_ERROR.

56 **Healthy**:

The user application has performed a Self-Test with success. The keys have not been fixed yet.

57 **Operative**:

The user application has performed a Self-Test and the Keys are setup. The cryptographic functionality is available and the user application is ready to perform encrypt/decrypt operations.

58 **Recovery**:

An "on demand Self-Test" required has returned a fail value. As the keys are setup, a recovery process zeroizing the AES sub-keys should be mandatory for the API IT entity user. The zeroization is not performed automatically but the transition to this state is the only one allowed when self test fail.

59 **Error**:

An "on demand Self-Test" required has returned a fail value. The keys still are not fixed. The function SCM_Terminate must be called by the API IT entity user.

Transitions

60 **Transition 1 (NO OPERATIVE – SELF TEST)**:

The transition from NO OPERATIVE to SELF TEST state happens when the SCM performs a SCM_Init(). The status output is set to system_status = SELF_TESTING and key_fixed = NO. There are not input and output data. The control data is SCM_init().

61 **Transition 2 (SELF TEST - HEALTHY)**:

The transition from SELF-TEST to HEALTHY state happens when the power up self-test succeeds and the keys are not fixed.

The status output is set to system_status = OK and key_fixed = NO. There are not input and output data. The transition depends on the values of the system_status and key_fixed variables.

62 **Transition 3 (HEALTHY – OPERATIVE)**:

The transition from HEALTHY to OPERATIVE state happens because the SCM performs a key fixation. The status output is set to system_status = OK and key_fixed = YES. The input data is the AES key. There is not output data, the subkeys generated are stored inside the SCM. The control data is SCM_KeySetup.

63 **Transition 4 (OPERATIVE – OPERATIVE)**:

The reflexive transition in the OPERATIVE state happens because the SCM may perform a crypto operation or a new key fixation. All these actions do not provoke a state transition. The status output remains as system_status = OK and key_fixed = YES. There are two control data that may be used: SCM_Cipher or SCM_KeySetup. The input and output data depend on the control data used. For SCM_Cipher: The input data are the counter and the buffer to be encrypted. The output data is the buffer encrypted. For SCM_KeySetup: The input data is the AES key and the output data is the subkeys generated again.

64 **Transition 5 (OPERATIVE – NO OPERATIVE)**:

The transition from OPERATIVE to NO OPERATIVE state happens when the SCM perform the recovery process zeroizing the sub-key values. Encryption/decryption functionality is not available and the output interface is inhibited. The status output is set to system_status = INITIAL_STATE and key_fixed = NO. The input data are the counter and the key. There is not output data. The control data is SCM_Terminate.

65 **Transition 6 (HEALTHY – NO OPERATIVE)**:

The transition from HEALTHY to NO OPERATIVE state happens when the SCM perform the recovery process zeroizing the sub-key values. Encryption/decryption functionality is not available and the output interface is inhibited. The status output is set to system_status = INITIAL_STATE and key_fixed = NO. The input data are the counter and the key. There is not output data. The control data is SCM_Terminate.

66 **Transition 7 (OPERATIVE – SELF TEST)**:

The transition from OPERATIVE to SELF TEST state happens when the SCM performs an on demand self-test. The status output is set to system_status = SELF_TESTING and key_fixed = YES. There are not input and output data. The control data is SCM_Self_test.

67 **Transition 8 (ERROR – NO OPERATIVE)**:

The transition from ERROR to NO OPERATIVE state happens when the SCM perform the recovery process zeroizing the sub-key values. Encryption/decryption functionality is not available and the output interface is inhibited. The status output is set to system_status = INITIAL_STATE and key_fixed = NO. The input data are the counter and the key. There is not output data. The control data is SCM_Terminate.

68 **Transition 9 (SELF TEST – ERROR)**:

The transition from SELF-TEST to ERROR state happens when the SCM performs an on demand Self-Test returning a FAIL result. The keys will not be fixed. The status output is set system_status = INT_ERROR or KAT_ERROR and key_fixed = NO. There are not input and output data. The transition depends on the values of the system_status and key_fixed variables.

69 **Transition 10 (RECOVERY – NO OPERATIVE)**:

The transition from RECOVERY to NO OPERATIVE state happens when the SCM perform the recovery process zeroizing the sub-key values. Encryption/decryption functionality is not available and the output interface is inhibited. The status output is set to system_status = INITIAL_STATE and key_fixed = NO. The input data are the counter and the key. There is not output data. The control data is SCM_Terminate.

70 **Transition 11 (SELF TEST – RECOVERY)**:

The transition from SELF-TEST to RECOVERY state happens when the SCM performs an on demand Self-Test returning a FAIL result. This event will require that the module recovers itself zeroizing the sub-key values. The status output is set system_status = INT_ERROR or KAT_ERROR and key_fixed = YES. There are not input and output data. The transition depends on the values of the system_status and key_fixed variables.

71 **<u>Transition 12 (HEALTHY – SELF TEST)</u>**:

The transition from HEALTHY to <u>SELF TEST</u> state happens when the SCM performs an on demand self-test. The status output is set to system_status = SELF_TESTING and key_fixed = NO. There are not input and output data. The control data is SCM_Self_test.

72 **<u>Transition 13 (SELF TEST - OPERATIVE)</u>**:

The transition from SELF-TEST to OPERATIVE state happens when the power up self-test succeeds and the keys are fixed. The status output is set to system_status = OK and key_fixed = YES. There are not input and output data. The transition depends on the values of the system_status and key_fixed variables.

73 For clarity reasons, the remainder events that did not provoke a state transition or those whose occurrence is considered as meaningless have not been included in the finite state diagram.

74 The API input parameters do not affect the ultimate state of the transitions.

5 Operational Environment

5.1 Operational Environment Policy

75 The following assumptions are to be satisfied by the operational environment:

a) The linker shell script (linker), the current HMAC-SHA1 calculator (scm_precodigo.c) and the scm.h are delivered in conjunction with the object module. The integrity may be compromised by modifying the head file scm.h to export directly internal low level cryptographic functions or making visible the status system variable and modifying its value bypassing this way the status control. Therefore, the replacement or modification of any component of the delivered module by unauthorized users is prohibited.

b) The Operating System enforces authentication method(s) to prevent unauthorized access to Module services.

c) The generation and input of the main key is under user application responsibility. It is supposed that is verified as correct and is securely generated and stored. The main key exchanged between the user application and the SCM Module must be entered encrypted by the user application, performing an AES-ECB on the key with a key which is given to the user as part of the delivered package.

d) All host system components that can contain sensitive cryptographic data (main memory, system bus, disk storage) must be located in a secure environment.

e) The user application accessing the module services in a separate virtual address space with a separate copy of the executable code.

f) The application designer must be sure that the client application is designed correctly and does not corrupt the address space of the SCM.

g) The unauthorized reading, writing, or modification of the address space of the SCM is prohibited.

h) The writable memory areas of the SCM (data and stack segments) are accessible only by a single application so that the module is in "single user" mode, i.e. only the one application has access to that instance of the module.

i) The operating system is responsible for multitasking operations so that other processes cannot access the address space of the process containing the SCM.

j) The user shall not link multi threaded applications to the SCM API.

k) The Crypto Officer shall be well-trained and non-hostile.

l) The Crypto Officer should install the generated files in a location protected by the host operating system security features. These protections should allow write access only to Crypto Officers and read access only to authorized users.

5.2 Compatible platforms

76 The Module is designed to run on any Windows platform (up from W95). The hardware (I386 platform) must support any Windows operating system. Any such computing platform that meets the conditions listed above can be used to host a FIPS 140-2 validated Module generated in accordance with this Security Policy. However, the module has been tested in a Windows XP SP2.

5.3 Software security

77 Integrity checks are performed in the process of generating and running an application using the SW Cryptographic Object Module:

a) Integrity of the Cryptographic module object file generated from the source code: SECUWARE delivers the product including a Cryptographic module object digital signature which will have to be validated by the final user for integrity assurance.

b) Integrity of the Cryptographic module object in the runtime executable application is checked at runtime when the application starts or an on demand self-test is requested. At runtime the **SCM_Self_test**() function uses the embedded HMAC-SHA1 digest to check the integrity of the memory mapped contents of the Cryptographic module object.

78 This chain of integrity checks assures that applications that are required to make use of cryptography will use FIPS 140-2 validated cryptographic module. This chain starts from SECUWARE when signing the SCM object module and the final user is in charge of verifying it before build the user application with the SCM.

79 The only way this module can be used in a FIPS 140-2 Approved mode of operation is if it is built according to the method described in section 8.2 - Initialization, Start-up and Operation; any deviation from the specified building method will violate the validation.

5.4 Critical security parameters

80 A Critical Security Parameter (CSP) is information, such as passwords, symmetric keys, asymmetric private keys, etc., that must be protected from unauthorized access. Since the Module is accessed via an API from a user referencing application, the Module is not in charge of preventing unauthorised access to any CSP.

81 The following CSPs are managed by the module:

- A 256-bit key (input parameter) to derive the associated subkeys for the AES algorithm, as part of the initialization process. As it has been previously mentioned, the generation, establishment and input of the main key is under user application responsibility.
- The counter is a sequential number in a larger data structure than buffer which the programmer wants to encrypt/decrypt according to AES-CTR.
- The AES sub-keys derived, are the input parameter of the AES algorithm.
- The hard coded key which decrypts the main AES-256 key entered by the user in encrypted form.
- The hard coded key used for the HMAC integrity tests

82 The application designer and the end user share a responsibility to ensure that CSPs are always protected from unauthorized access. This protection will generally make use of the security features of the host hardware and software

5.5 Physical Security

83 The Module does not claim to enforce any physical security as it is implemented entirely in software.

5.6 Electromagnetic Interference/Electromagnetic Compatibility

84 The software runs in a platform which conforms to the EMI/EMC requirements specified by 47 Code of Federal Regulations, Part 15, Subpart B, Unintentional Radiators, Digital Devices, Class B.

5.7 Mitigation of Other Attacks

85 The SCM does not implement any more countermeasures to mitigate against any specific attacks.

6 Cryptographic key management

6.1 RNG

86 Not applicable; the SCM does not utilize Random number Generators.

6.2 Key Generation

87 Not applicable; the SCM does not generate any key internally.

6.3 Key Establishment

88 Not applicable; no key establishment among crypto modules.

6.4 Key Entry and Output

89 The AES main key is generated by the user application and is entered to the SCM using automated methods and in encrypted form.

90 The entered key is always associated to the correct user application entity as long as the Operational Environment Policy assumptions (bullets e), f), g), h), i) and j)) are met.

91 The main AES-256 key must be entered into the SCM encrypted by the user performing a simple AES ECB mode encryption with a predefined key which will be given to the user. Properties of this predefined key are following:

- type and identifier: 256 bits AES ECB key used for decrypting the main AES-256 key.
- Storage location: hard coded into the module (scm.c).
- Form in which the key is stored: plaintext.

92 Encryption of the main AES-256 key is under user application responsibility. As the SCM is a library which is customized for each customer, each user will have its own predefined key.

93 No key is outputted from the SCM.

6.5 Key storage and Key Zeroization

94 The **SCM** does not store any CSP (except the hard coded key which decrypts the main AES256 key entered by the user in encrypted form and the hard coded key for HMAC) in persistent media; while the module is initialized the main AES key, the counter and the derived sub-keys reside temporarily in RAM shall be zeroized by the user application at the end of the session and in case of recovery (**SCM_Terminate**). In order to zeroize the above mentioned hard coded keys, a reformatting of the hard drive which contains the module shall be performed.

95 The size of the sub-keys is 240 bytes, the size of the counter is double word and the size of the main key is 256 bits. The function uses a technique called Zeroization, which fills byte to byte the associated structures with 0s.

96 The hard coded key and its associated subkeys are also zeroized in RAM after been used to decrypt the user key,

7 Self-Tests

97 The SCM provides a self-test functionality including "power–up" and "on demand" self-test to ensure proper operation of the module.

98 Power-up tests are run automatically when the SCM is initialized. Additionally, self-tests may be executed at any time by calling the **SCM_Self_test()** function which update the **system_status**. No FIPS mode cryptographic functionality will be available until successful execution of all power-up tests.

99 No authentication is required to perform self-tests either automatically or upon demand.

100 On self-test failure, all cryptographic operations are disabled and outputs inhibited until the module is recovered. The most likely cause of a self-test failure is memory or hardware errors. In practice, a self-test failure means that the user's application should exit and be restarted.

7.1 Power-up and "on demand" Tests

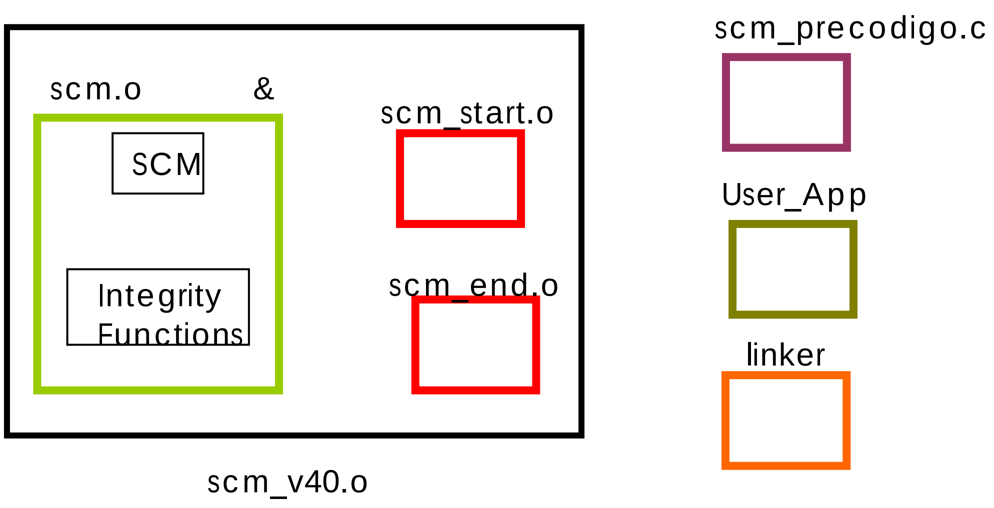
101 Power-up and "on-demand" tests include integrity tests and cryptographic algorithm known answer tests.

7.1.1 Integrity Tests

102 The integrity tests are those that check the object cryptographic module in the runtime executable application at runtime as the first power-up self-tests or "on demand". The integrity tests are performed using a HMAC-SHA1 digest calculated over the running executable.

Integrity Algorithm

103 The object code generated by the compilation of the Cryptographic module files is carefully isolated from application object code. This isolation is accomplished by collecting all of Cryptographic module object code into a single discrete unit of object code (**scm_v40.o**).

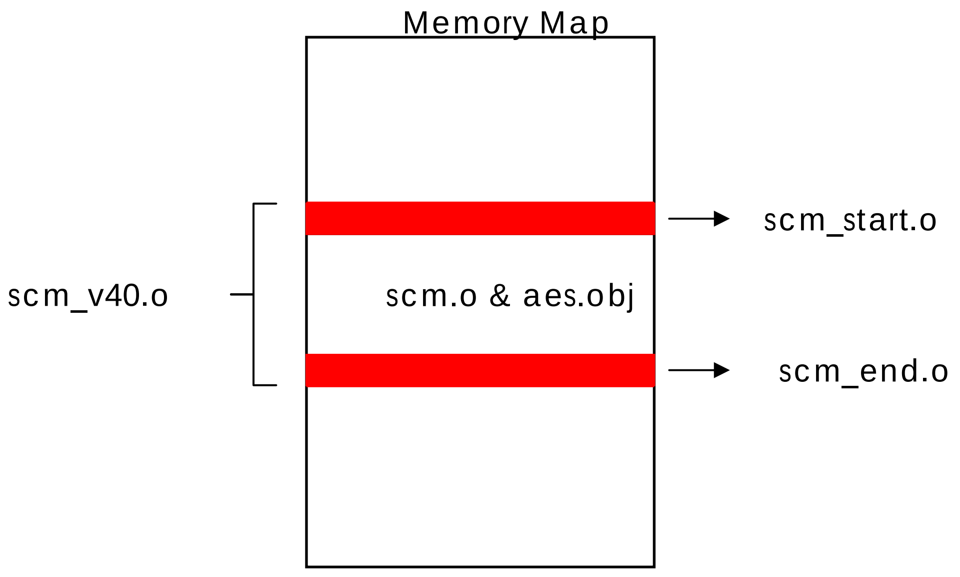


104 We refer to this discrete unit as the **Secuware Security Framework – Crypt4000 Module** (SCM). The SCM contains only the module object code and its integrity is protected by a HMAC-SHA1 which is "auto-calculated" in a first stage when the whole code (SCM + User_App) is compiled with **scm_precodigo.c**.

105 In a second stage, the whole code is compiled again including in the object digest previously calculated. This process is carried out by the provided script **linker**, which behaves as a C compiler.

106 This digest is checked whenever an application linked against the SW Cryptographic Object Module file, performs a call to the **SCM_init()** or **SCM_Self_test()** functions.

107 The design of the SW Cryptographic Object Module includes the definition of reference points within the object code that are used to define the object code to be protected by the runtime integrity test (**scm_start.o** and **scm_end.o**).



108 As it was explained above, at application link time, a HMAC-SHA-1 digest of the memory mapped object code is created and stored in the SW Cryptographic Object Module. This digest is calculated entirely within the confines of the SW Cryptographic Object Module and so will never include extraneous object code.

109 The integrity test is required over the object code within the logical boundary only. This integrity test will be performed each time the SCM is initialized and as part of the self-test user requests.

110 The **linker** script is in charge of performing the following steps:

1. Compile and link the files: scm_precodigo.c, scm_v40.o and the user application code generating an executable file.
2. Execute the executable generated in the previous step. The object digest will be returned.
3. Recompile and link the files: scm_precodigo.c, scm_v40.o, and the application code giving as a parameter the previously calculated digest. The final executable file or object will be generated.

111 The final result is an executable program that executes the application code. The integrity check will be performed whenever the SCM starts (the application calls the **SCM_init()** function).

Integrity Process File Description

112 The description of the files involved in the module compilation and the executable application generation is as follows:

- **scm.o**: this file contains the SCM API including the integrity functions.
- **aes.obj**: C AES functions
- **scm_start.o**: start reference point within the object code that is used to define the object code to be protected by the runtime integrity test.
- **scm_end.o**: end reference point within the object code that is used to define the object code to be protected by the runtime integrity test.
- **scm_hatillo.c:** in charge of generating the start and end references above mentioned.
- **scm_v40.o**: a single discrete unit of object code that contains scm_start.o, scm.o, aes.obj and scm_end.o.
- **scm_precodigo.c**: during the first compilation, it is in charge of calling the functions that will calculate the HMAC-SHA1 digest. After that, it is involved in final executable generation.
- **User application**: The user IT entity - application code that contains the main application functionality. It calls SCM API functions.
- **linker**: It is in charge of generating the final application that contains the SCM and the CM integrity check.

7.1.2 Known Answer Tests

113 Known Answer Tests (KATs) are tests where a cryptographic value is calculated and compared with a stored previously determined answer. KAT tests are performed to check whether the algorithm behaves as expected.

114 KAT are performed over:

- AES-CTR algorithm
- AES ECB decrypt function

KAT for AES-CTR algorithm

115 The method uses the following parameters:

- A 256 bit prefixed key, which the sub-keys used by the AES-CTR encryption algorithm are derived from.
- An initial 16 bytes known vector for the AES-CTR KAT.

- A final 16 bytes known vector for the AES-CTR KAT. This final vector is the result of encrypting the initial vector before performing the KAT.
- A fixed counter for the AES-CTR KAT.

116 The KAT procedure performs the following steps:

- AES (CTR) encryption

  1. The sub-keys used by the CTR algorithm are derived from the 256 bit prefixed key.
  2. The initial vector is encrypted with the sub-keys derived in the previous step and the fixed counter.
  3. The vector obtained is compared byte to byte to the final known vector.
  4. If the comparison succeeds, this known answer test has finished successfully.

- AES (CTR) decryption

  1. The sub-keys used by the CTR algorithm are derived from the 256 bit prefixed key.
  2. The final vector is decrypted with the sub-keys derived in the previous step and the fixed counter.
  3. The vector obtained is compared byte to byte to the initial known vector.
  4. If the comparison succeeds, this known answer test has finished successfully.

- AES (ECB) encryption

  1. The sub-keys used by the ECB decryption algorithm are derived from the 256 bit prefixed key for encryption.
  2. The initial vector is encrypted with the sub-keys derived in the previous step.
  3. The vector obtained is compared byte to byte to the final known vector.

4. If the comparison succeeds, this known answer test has finished successfully.

- AES (ECB) decryption

  1. The sub-keys used by the ECB decryption algorithm are derived from the 256 bit prefixed key for decryption.
  2. The initial vector is decrypted with the sub-keys derived in the previous step.
  3. The vector obtained is compared byte to byte to the initial known vector.
  4. If the comparison succeeds, this known answer test has finished successfully.

117 If all the tests have finished successful, then the module returns ok.

KAT for AES-ECB decrypt function

118 The method uses the following parameters:

- A 256 bit prefixed key, which the sub-keys used by the AES-ECB decryption algorithm are derived from.
- An initial 16 bytes known vector for the AES-ECB decrypt function KAT.
- A final 16 bytes known vector for the AES-ECB KAT. This final vector is the result of decrypt the initial vector before performing the KAT.

119 The KAT procedure performs the following steps:

1. The sub-keys used by the ECB decryption algorithm are derived from the 256 bit prefixed key.
2. The initial vector is encrypted with the sub-keys derived in the previous step.
3. The vector obtained is compared byte to byte to the final known vector.

4. If the comparison succeeds, the known answer test has finished successfully.

7.2 Conditional Tests

120 **Pair-wise Consistency Test:** Not applicable; the SCM does not implement any public key cryptographic function.

121 **Software/Firmware Load Test:** Not applicable; the SCM does not utilize externally loaded cryptographic modules.

122 **Manual Key Entry Test:** Not applicable; keys are not manually entered into the Module.

123 **Bypass Test:** Not applicable; the SCM does not implement any bypass capability.

7.3 Critical Function Tests

124 The SCM does not implement any critical function tests.

8 Design Assurance

8.1 Configuration management

125 Configuration management: CIs version control, change control, flaw remediation tracking, is managed by SECUWARE using VSS 2005 (Visual Source Safe) for managing the life-cycle of its products. The source code revisions are maintained in a VSS repository with write access restricted to the authorised developers.

8.1.1 Configuration Items Identification Method

126 In order to ensure that it would be possible for consumers to identify the SCM version (e.g. at the point of purchase or use), the module contains a unique reference (version number) which is stated as part of the delivered Software Cryptographic Object Module file name: **scm_vXY.o** where X is de version number and Y the revision number.

127 For source code files, VSS assign automatically the revision number used for internal purposes. In addition, to allow traceability to the item configuration list (section 8.1.3 Configuration Item List), a tag with the version number X.Y (where X is de version number and Y the revision number – it shall not to be confused with the internal VSS revision) is manually added to each source file.

128 When any source file is created, version v10 is always assigned. Every minor revision implies a Minor revision number increase. Every major revision implies a Major revision number increase. The minor revision number is set to 0 when major revision is increased.

129 The method used to uniquely identify the documentation configuration items describe how the status of each configuration item can be tracked throughout the life-cycle of the SCM.

130 Each document to be managed by VSS is considered a manageable element as a whole. The methodology user for managing and controlling the versioning of the documentation throughout its life-cycle is the following:

- Naming.

Name vX.Y, where *Name* is the name of the related document. The name differentiates each document among the others, so each name must be unique among the managed documentation elements. *vX.Y* is the version of the document.

– Version update

The Version of a document consists of two parts:

- Major number (X)

  This number indicates the actual version of the document.

- Minor number (Y)

  This number indicates the revision number of the document.

131 When a document is created, version v1.0 is always assigned. Every minor revision in the content of the document, such as those related to normal evolution of the document (e.g. references update, small changes in the content, reviews, etc.), implies a Minor revision number increase. Every major revision in the content of the document (e.g. new parts added, important structural changes, etc.) implies a Major revision number increase. The minor revision number is set to 0 when major revision is increased.

132 The CI identification method allows:

a) tracking versions of the same configuration item;
b) identifying superseded versions of a configuration item.

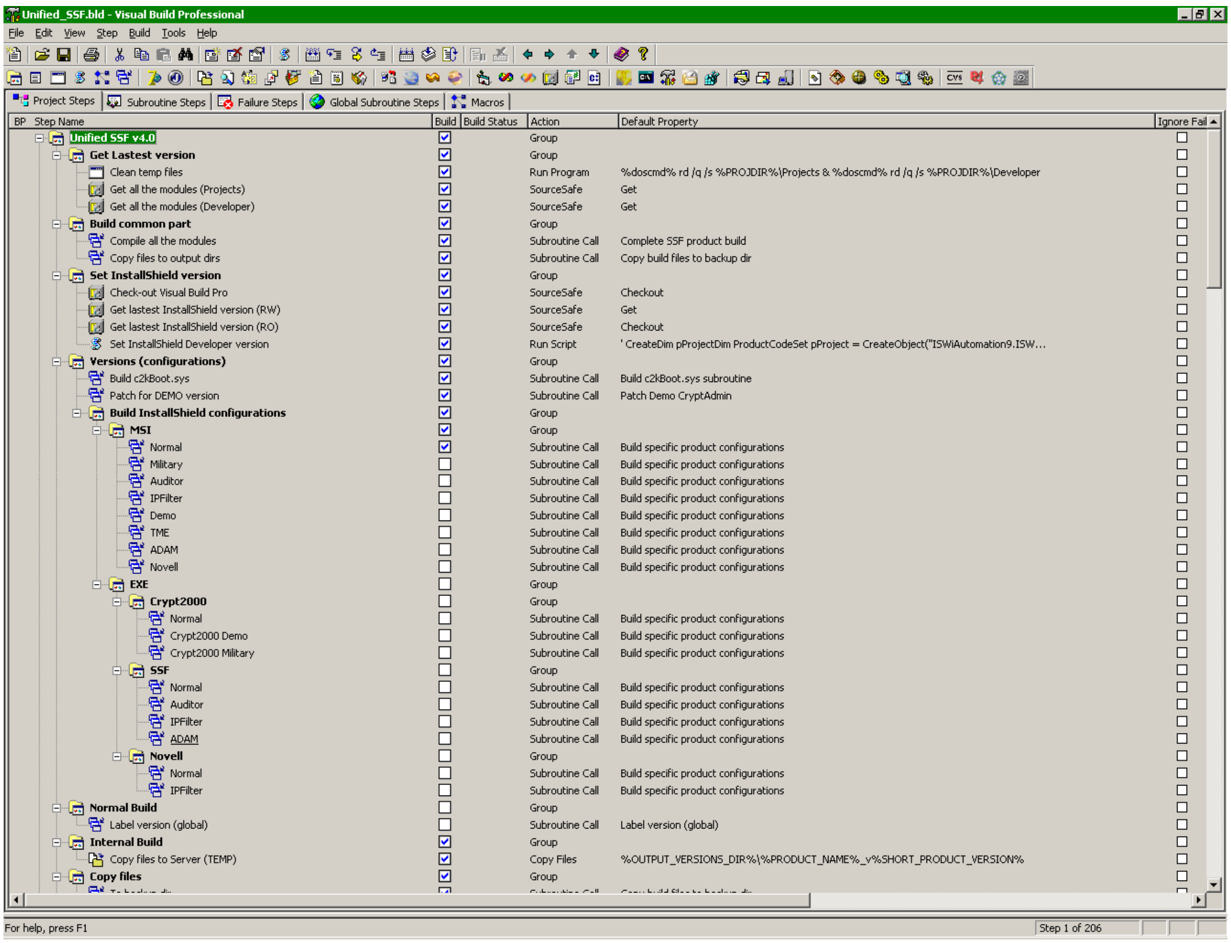
8.1.2 Configuration Management System

133 As a preparation for the configuration management, the following directories are created as the folders to be managed by VSS:

– Development Directory: all versions and code modifications are held in this folder.
– Documentation Directory: all (versioned) documents are held in this folder.
– Tools: all (versioned) tools used for a product development are stored in this folder.

134 Updates and modifications are carried out following the options VSS offers:

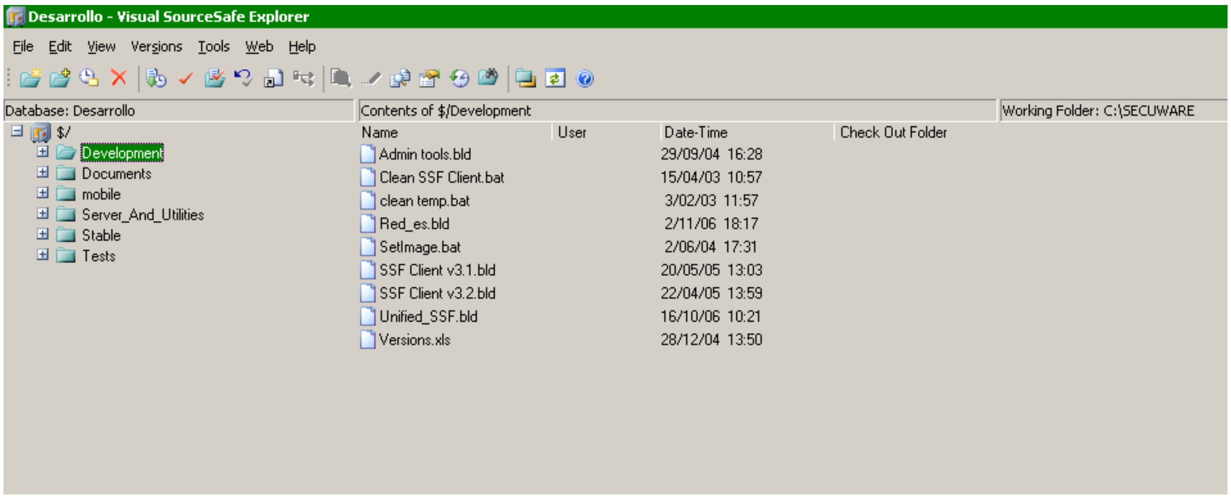
- Define users with a predefined profile.
- Each user is able to modify solely the assigned resources, both documentation and source code. At the same time, each user only has privilege to access certain projects.
- Every change is registered in VSS, and it is possible to create a log file with those changes.
- VSS allows realizing comparisons between different versions.
- VSS also allows performing automatic build tasks and compiling certain code version from a certain date.


Unified_SSF.bld - Visual Build Professional
File Edit View Step Build Tools Help
Project Steps
Subroutine Steps
Failure Steps
Global Subroutine Steps
Macros

BP	Step Name	Build	Build Status	Action	Default Property	Ignore Fail
	Unified SSF v4.0	✓		Group		☐
	Get Lastest version	✓		Group		☐
	Clean temp files	✓		Run Program	%doscmd% rd /q /s %PROJDIR%\Projects & %doscmd% rd /q /s %PROJDIR%\Developer	☐
	Get all the modules (Projects)	✓		SourceSafe	Get	☐
	Get all the modules (Developer)	✓		SourceSafe	Get	☐
	Build common part	✓		Group		☐
	Compile all the modules	✓		Subroutine Call	Complete SSF product build	☐
	Copy files to output dirs	✓		Subroutine Call	Copy build files to backup dir	☐
	Set InstallShield version	✓		Group		☐
	Check-out Visual Build Pro	✓		SourceSafe	Checkout	☐
	Get lastest InstallShield version (RW)	✓		SourceSafe	Get	☐
	Get lastest InstallShield version (RO)	✓		SourceSafe	Checkout	☐
	Set InstallShield Developer version	✓		Run Script	' CreateDim pProjectDim ProductCodeSet pProject = CreateObject("ISWiAutomation9.ISW...	☐
	Versions (configurations)	✓		Group		☐
	Build c2kBoot.sys	✓		Subroutine Call	Build c2kBoot.sys subroutine	☐
	Patch for DEMO version	✓		Subroutine Call	Patch Demo CryptAdmin	☐
	Build InstallShield configurations	✓		Group		☐
	MSI	✓		Group		☐
	Normal	✓		Subroutine Call	Build specific product configurations	☐
	Military	☐		Subroutine Call	Build specific product configurations	☐
	Auditor	☐		Subroutine Call	Build specific product configurations	☐
	IPFilter	☐		Subroutine Call	Build specific product configurations	☐
	Demo	☐		Subroutine Call	Build specific product configurations	☐
	TME	☐		Subroutine Call	Build specific product configurations	☐
	ADAM	☐		Subroutine Call	Build specific product configurations	☐
	Novell	☐		Subroutine Call	Build specific product configurations	☐
	EXE	☐		Group		☐
	Crypt2000	☐		Group		☐
	Normal	☐		Subroutine Call	Build specific product configurations	☐
	Crypt2000 Demo	☐		Subroutine Call	Build specific product configurations	☐
	Crypt2000 Military	☐		Subroutine Call	Build specific product configurations	☐
	SSF	☐		Group		☐
	Normal	☐		Subroutine Call	Build specific product configurations	☐
	Auditor	☐		Subroutine Call	Build specific product configurations	☐
	IPFilter	☐		Subroutine Call	Build specific product configurations	☐
	ADAM	☐		Subroutine Call	Build specific product configurations	☐
	Novell	☐		Group		☐
	Normal	☐		Subroutine Call	Build specific product configurations	☐
	IPFilter	☐		Subroutine Call	Build specific product configurations	☐
	Normal Build	☐		Group		☐
	Label version (global)	☐		Subroutine Call	Label version (global)	☐
	Internal Build	✓		Group		☐
	Copy files to Server (TEMP)	✓		Copy Files	%OUTPUT_VERSIONS_DIR%\%PRODUCT_NAME%_v%SHORT_PRODUCT_VERSION%	☐
	Copy files	✓		Group		☐

For help, press F1 Step 1 of 206

Figure 4: Automatic compilation


Desarrollo - Visual SourceSafe Explorer
File Edit View Versions Tools Web Help

Database: Desarrollo

- $/
  - Development
  - Documents
  - mobile
  - Server_And_Utilities
  - Stable
  - Tests

Contents of $/Development

Working Folder: C:\SECUWARE

Name	User	Date-Time	Check Out Folder
Admin tools.bld		29/09/04 16:28	
Clean SSF Client.bat		15/04/03 10:57	
clean temp.bat		3/02/03 11:57	
Red_es.bld		2/11/06 18:17	
SetImage.bat		2/06/04 17:31	
SSF Client v3.1.bld		20/05/05 13:03	
SSF Client v3.2.bld		22/04/05 13:59	
Unified_SSF.bld		16/10/06 10:21	
Versions.xls		28/12/04 13:50	

Figure 5: VSS Directory configuration

135 There is an assigned role for each maintenance and update task:

– Development controller: role in charge of controlling and maintaining the source code, as well as defining which new elements related to the development must be managed by VSS.

– Documentation controller: role in charge of controlling and maintaining the documentation, as well as defining which new documents must be managed by VSS.

– Tools controller: role in charge of controlling and maintaining the tools (e.g. when a new version of a tool is released, this role must evaluate and decide if migrate the current one or not).

8.1.3 Configuration Item List

136 The following table contains the Configuration Items which are controlled in accordance with the configuration management procedures above described:

CI	**Version**	**Date**
SW Crypto Module Security Policy	4.0	Nov 2008
Software Cryptographic Module User Guidance	4.0	Nov2008
Source code files:		Nov2008
▪ Makefile	4.0	
▪ aes.c	4.0	
▪ scm.c	4.0	
▪ scm.h	4.0	

▪ scm_hatillo.c	4.0	
IUT: ▪ scm_v40.o	4.0	Nov2008

137 The following table shows items under configuration control that are not part of the crypto module but are used to support the generation of the final object:

Item	Version	Date
Supporting files for building the final object: ▪ scm_precodigo.c ▪ linker	 4.0 4.0	Nov 2008

8.2 Initialization, Start-up and Operation

8.2.1 Secure installation

138 This section describes the applied procedures for achieving a security installation for the SW Crypto Module.

Integrity Assurance

139 PGP (Pretty Good Privacy) is used for signing the product with a SECUWARE certificate, assuring that any modification in the content is properly detected.

Confidentiality Assurance

140 When using Internet as the media for delivering the product, an FTP server with access-protected paths is used, and therefore only the corresponding customer/partner can access the content after proper identification/authentication.

141 The SECUWARE Service Department creates a virtual directory specific for the customer/partner in a FTP server (ftp://ftp.clients.secuware.com/client-x), with a control access based on user/password exclusively for that customer/partner

142 PGP is also used for, besides signing the content, encrypting it with the public key of the customer/partner, ensuring that only the corresponding receiver can access the content of the product.

143 It is important to highlight, as stated in the previous paragraph, the fact that the library binaries, with the persisted key inside (for decrypting the AES256 main key entered by the user in encrypted form), are delivered to the final customer in an encrypted form and therefore, the customer must keep the library binaries within the boundaries of a secure perimeter once they have been delivered to them. The hard-coded main AES-256 key can not be changed by the user.

Proof of origin

144 PGP digital signature assures proof of origin.

8.2.2 Secrets distributions

145 PGP is used for protecting the product when using the FTP directory. For that purpose, PGP public keys must be exchanged between Secuware and the customer/partner. This information is sent only once, and can it can be done at the beginning of the Agreement of Service Contract between Secuware and the customer/partner or the first time the customer/partner buys a product.

146 The same way, the harcoded key to be used by the user application to encrypt the main AES256 key, must be delivered to the final customer.

147 The customer shall generate a PGP key pair and must send its public key to Secuware. Secuware will encrypt with this public key all the secrets:

- information to access the ftp including the URL, user and password,
- key to be used for AES256 main key ECB encrypt

and will send it to the customer, by e-mail (secuware´s public key for signature validation purposes has been previously made available for the customer).

8.2.3 Initialization and start-up

148 The Crypto Module building procedure performs a collection of steps to build a user application, which interacts with the SW Crypto module:

- Previously it is necessary to install the MingW32 environment and MSYS. An easy way to do this, is to run the two Windows® installer programs from http://sourceforge.net/projects/mingw and www.mingw.org respectively. The current versions are called MinGW-5.1.3.exe and MSYS-1.0.10.exe. Run the MINGW installer *first*.

- Copy the provided code into the MingW32 environment. The user application source and the Crypto module API, must be copied as a necessary step for Cryptographic module initialization and Start-up. Those sources will be used to generate an executable.

- Execute the provided script, ***linker***, as a gcc compiler or a ld linker, with the following parameters under the mingw32 environment:

  ./linker *userApplication.c* –o *executableName*

  where *userApplication.c* is the application source code that contains CM API calls and *executableName* is the final executable name. The files **scm_v40.o** and **scm_precodigo.c** must be on the same path than the **linker** script.

- Once the final executable has been generated, then it is ready to be executed like whichever windows executable file.

149 The **linker** script functionality is described in section 7.1.1 Integrity Tests.

150 The final result is an executable program that executes the Application code. The integrity check will be performed whenever the SCM starts (the application calls the **SCM_Init** function).

151 The version of the compiler and linker needed to build correctly the module are gcc.exe (GCC) 3.4.5 (mingw special) and GNU ld version 2.17.50 20060824.

8.2.4 Operation rules

152 The operation rules of the Cryptographic Module are the followings:

1. The Crypto Officer must provide the user with the user guidance [GUI].
2. Before performing the cryptographic module key setup, the system status must be set to OPERATIVE which means that the self-test (power-up or "on demand") must have been successfully completed.
3. The main AES-256 key must be entered into the SCM encrypted by the user performing a simple AES ECB mode cipher with a predefined key which will be given to the user (see 8.2.2 Secrets distributions) and hard coded into the module. Encrypting the main key is under user application responsibility. As the SCM is a library which is customized for each costumer, each user will have its own key.
4. The library binaries, with the persisted key inside (for decrypting the AES256 main key entered by the user in encrypted form), are delivered to the final customer in an encrypted form and therefore, the customer must keep the library binaries within the boundaries of a secure perimeter once they have been delivered to them.

5. Before performing any cryptographic operation, the system status must be set to OPERATIVE which means that the self-test (power-up or "on demand") must have been successfully completed and the key setup performed.
6. To update the system status, a self-test should be performed before each crypto operation.
7. Before performing a Self-Test operation, the module must have been initialized with the SCM_Init function.
8. On Self-test successful, the system status is updated to OPERATIVE value; else the system status is updated to the NO OPERATIVE value.
9. On recovery required status, no cryptographic services will be available, the sub-keys, the key and the counter will be zeroized and the module will have to be again initialized.
10. While no operative status or during the self-test execution, the SCM encryption/decryption functionality is not available and the output interface is inhibited.
11. When the SCM has finished any crypto operation, the sub-keys, the key and the counter must be zeroized.
12. The specification of the CTR mode requires a unique block counter (counter according to the API) for each plaintext block that is ever encrypted under a given key, across all messages. If, contrary to this requirement, a block counter is used repeatedly, then the confidentiality of all of the plaintext blocks corresponding to that counter block may

be compromised. Two main aspects are to be taking into account to satisfy the uniqueness of the counter block:

- o An incrementing function for generating the counter blocks from any initial counter block to ensure that counter blocks do not repeat within a given message.
- o The initial counter blocks must be chosen to ensure that counters are unique across all messages that are encrypted under a given key. The maximum number of distinct blocks that SCM can crypt without repeating the counter is $2^{\text{double word size.}}$ See [SP800-38A].

8.3 Development

153 The operational rules are a trivial translation of the finite state machine and its implementation as it has been justified in the SCM design.

154 The trace of source code to the modules design is shown for completeness and accuracy checking, in the following table:

Function	**Module**
High level functions	
`void WINAPI SCM_Init(void);`	SCM
`void WINAPI SCM_KeySetup(LPBYTE Key);`	SCM
`void WINAPI SCM_Cipher(DWORD counter, LPBYTE buffer);`	SCM
`int WINAPI SCM_Show_status(void);`	SCM
`void WINAPI SCM_Terminate(LPBYTE Key, DWORD *counter);`	Supporting Functions
`void WINAPI SCM_Self_test(void);`	Supporting Functions
Low level functions	
`int WINAPI SCM_Check_INT(void);`	Supporting Functions
`int WINAPI SCM_Check_KAT(void);`	Supporting Functions
`int CompruebaFirmaInterna(void);`	Supporting Functions
`void FirmaInterna(unsigned char *hash);`	Supporting Functions
`int AesFastKeySetupEnc256(u32 sk[240], const u8 Key[32]);`	AES
`void AesFastCipherBlock_sk(const u32 sk[240], u32 counter, u8 buffer[16]);`	AES

`int AesFastKeySetupDec256(u32 sk[240], const u8 Key[32]);`	AES
`void StandAloneECB(const u32 sk[240], u8 buffer[16]);`	AES
`void StandAloneECBDeciph(const u32 sk[240], u8 buffer[16]);`	AES

155 Commented function headers are listed in Annex: Functions Header.

8.4 Guidance document

156 Guidance information for both Crypto Officer and User roles is included in [GUI], covering the following aspects:

- For Crypto officer role, the guidance specifies:
  - the administrative functions,
  - security events,
  - security parameters,
  - logical interfaces,
  - procedures on how to administer the cryptographic module in a secure manner,
  - assumptions regarding user behaviour that is relevant to the secure operation of the cryptographic module.

- For User role the guidance specifies:
  - the approved security functions,
  - logical interfaces available to the users of the cryptographic module,
  - all user responsibilities necessary for the secure operation of the cryptographic module.

9 Glossary

AES	Advanced Encryption Standard
API	Application Programming Interface
CI	Configuration Item
CM	Cryptographic Module
CSP	Critical Security Parameter
CTR	Counter
EMC	ElectroMagnetic Compatibility
EMI	ElectroMagnetic Interferences
FIPS	Federal Information Processing Standard
FSM	Finite State Machine
HMAC	Hash based Message Authentication Code
HW	Hardware
IT	Information Technology
IUT	Implementation Under Test
KAT	Known Answer Test
NIST	National Institute of Standards and Technology
OS	Operating System
PGP	Pretty Good Privacy
SCM	SECUWARE Cryptographic Module: Refers to "**Secuware Security Framework – Crypt4000 module**" module FIPS name
SHA	Secure Hash Algorithm
SW	Software
SW	Cryptographic Module SCM
VSS	Visual Source Safe

Annex: Functions Header

157 Below, the header information of the low level and high level functions with appropriate comments is presented.

High level functions

void WINAPI SCM_Init(void);

```
/**
 * \brief          This function calls the SCM_Self_test function and initialized
 *                 the module in FIPS mode of operation.
 *
 * \param output            return 1 if the test has gone well, 0 in another case.
 */
```

void WINAPI SCM_KeySetup(LPBYTE Key);

```
/**
 * \brief          With a given pointer to the key, this function initializes the
 *                 AES subkeys
 *
* \param Key                pointer to where the AES Key is allocated, length must be 32 bytes.
 */
```

void WINAPI SCM_Cipher(DWORD counter, LPBYTE buffer);

```
   /**
    * \brief       This function receives the buffer that the programmer wants to encrypt,
    *              saving the encrypted block in the same buffer. Counter must be
    *              modified according to AES-CTR.
    *
    * \param counter        sequential number in a larger data structure than buffer which the
                            programmer wants to encrypt/decrypt according to AES-CTR
    *
    * \param buffer         input/output buffer for the encrypted/decrypted result. Its length must
                            be 16Bytes
    *
    */
```

int WINAPI SCM_Show_status(void);

```
/**
 * \brief          This function returns true if the SCM is in approved mode of operation
 *                 and there has been no error or false if it isn't
 *
 * \param output            return a integer showing the status of the cipher in the form 2 *
                            system status + key fixed.
 */
```

void WINAPI SCM_Terminate(LPBYTE Key, DWORD *counter);

```
/**
 * \brief          This function must be called at the end of the encryption / decryption,
 *                 doing a zeroization in the memory address that allocates the Key, the counter
                   and the Subkeys.
 *
* \param Key                pointer to where the AES Key is allocated
* \param counter          pointer to the AES-CTR counter.
 *
 */
```

void WINAPI SCM_Self_test(void);

```
/**
 * \brief          Performs the self tests, updating the system status
 *
 */
```

Low level functions

int WINAPI SCM_Check_INT(void);

```
/**
 * \brief          Performs the Integrity Test
 *
 * \param output          return 1 if the test has gone well, 0 in another case.
 */
```

int WINAPI SCM_Check_KAT(void);

```
/**
 * \brief          Performs the Known Answer Test for the AES algorithm implementation
 *
 * \param output          return 1 if the test has gone well, 0 in another case.
 */
```

int CompruebaFirmaInterna(void);

```
/**
 * \brief          This function checks if the stored HMAC-SHA1 is equal to the calculated.
 *
 * \param output          Returns 1 if the integrity test has gone well, 0 in another case.
 */
```

void FirmaInterna(unsigned char *hash);

```
/**
 * \brief          This function calculate the HMAC-SHA1 corresponding to the code between
                   scm_start and scm_end.
 *
 * \param sig             where to save the calculated HMAC-SHA1
 */
```

int AesFastKeySetupEnc256(u32 sk[240], const u8 Key[32]);

```
/**
 * \brief        With a given pointer to the key, this function initializes the AES subkeys
 *               at the specified memory position.
 *
 * \param sk              pointer to the generated subkeys.
 *
 * \param Key             pointer to where the AES Key is allocated. It's 256 bits
 */
```

void AesFastCipherBlock_sk(const u32 sk[240], u32 counter, u8 buffer[16]);

```
/**
 * \brief        This function receives the pointer to the AES subkeys, and the buffer that
 *               the programmer wants to encrypt, saving the encrypted block in the same
 *               buffer.
 *
 * \param sk            pointer to the generated subkeys.
 * \param counter       sequential number in a larger data structure than buffer which the
 *                      programmer wants to encrypt/decrypt
 * \param buffer        input/output buffer where the programmer wants to save the
                        encrypted/decrypted result.
*/
```

int AesFastKeySetupDec256(u32 sk[240], const u8 Key[32]);

```
/**
 * \brief        With a given pointer to the key, this function initializes the AES subkeys for
                 decrypting at the specified memory position.
 *
 * \param sk            pointer to the generated subkeys.
 *
 * \param Key           pointer to where the AES Key is allocated. It's 256 bits
 */
```

void StandAloneECB(const u32 sk[240], u8 buffer[16]);

```
/**
 * \brief        Execute standalone AES-ECB encryption
 *
 * \param sk             pointer to the generated subkeys.
 *
 * \param buffer         buffer to encrypt
 */
```

void StandAloneECBDeciph(const u32 sk[240], u8 buffer[16]);

```
/**
 * \brief        Execute standalone AES-ECB decryption
 *
 * \param sk             pointer to the generated subkeys.
 *
 * \param buffer         buffer to decrypt
 */
```