Becrypt Ltd

Becrypt Cryptographic Library

FIPS 140-2 Non-Proprietary Security Policy

12 October 2018 version 2.3

Table of Contents

List of Tables

List of Figures:

Introduction

This document is the FIPS 140-2 non-proprietary Security Policy detailing how the Becrypt Cryptographic Library software version 3.0/hardware version Intel Core i5-4300Y meets the Security Level 1 requirements of FIPS 140-2. This security policy also describes how the library is to be used in a secure manner to comply with FIPS 140-2 when compiled into applications.

FIPS 140-2 (Federal Information Processing Standards Publication 140-2) specifies the security requirements for a cryptographic module protecting sensitive information. Based on four security levels for cryptographic modules this standard identifies requirements in eleven sections. For more information about the standard please visit csrc.nist.gov/groups/STM/cmvp/index.html.

In this document, the Becrypt Cryptographic Library is also referred to as "the module".

For information about Becrypt Ltd. please visit www.becrypt.com

Modes

The 32/64 bit library may operate in one of two *Approved* modes of operation:

FIPS Mode 1:	All defined approved and allowed algorithms enabled.
FIPS Mode 2:	RSA algorithms which involve private keys are not enabled, all other approved and allowed algorithms are enabled.

There is one *Unapproved* mode of operation for the 32/64 bit library in which the PRNG algorithm is enabled and non-compliant AES keys are generated.

The 16 bit library has only a single *Approved* mode of operation and no *Unapproved* mode.

The 32/64 bit library is initialized to FIPS Mode 1 by default. To specify the mode a parameter that requests a set of desired algorithms is passed as a parameter. The most appropriate mode for the requested algorithms is selected.

Cryptographic Module Specification

The Becrypt Cryptographic Library is a software-hybrid module providing core cryptographic functionality for software applications. This document describes software version 3.0 and hardware version Intel Core i5-4300Y of this module.

The module comprises two software sub-modules: a 32/64-bit sub-module supporting all the functionality, and a 16-bit sub-module for use in Intel Real-Mode environments supporting a subset of the functionality.

The module supports the following FIPS approved and non-approved algorithms:

Table 1 Approved Algorithms in 32/64 bit sub-module

Algorithm	Implementation details	Certificates
AES (FIPS 197)	CBC, ECB, CTR, OFB Mode 128, 192, 256 Key length	#2883
AES Key Wrap (SP 800-38F)	128, 192, 256 Key length. The key establishment methodology provides between 128 and 256 bits of encryption strength.	#2883
RSA (FIPS 186-4)	ANSI X9.31 (Sig Ver); PKCS v1.5 (Sig Ver); Modulus sizes 2048 and 3072 bits with SHA-256.	#1516
	ANSI X9.31 (Sig Ver); PKCS v1.5 (Sig Ver); Modulus sizes 2048 and 3072 bits with SHA-1.	#1516 Allowed in FIPs mode for legacy systems only
	ANSI X9.31 (Sig Ver); PKCS v1.5 (Sig Ver); Modulus size 1024 bits.	#1516 Allowed in FIPs mode for legacy systems only
RSA (FIPS 186-4)	FIPS 186-4 Key Gen using probable primes with conditions (B.3.6) ANSI X9.31 (Sig Gen); PKCS v1.5 (Sig Gen); Modulus sizes 2048 and 3072 bits with SHA-256.	#1516
HMAC (FIPS 198-1)	SHA-1, SHA-256	#1817
Random Bit Generators (SP800-90A)	CTR_DRBG using AES	#520
SHS (FIPS 180-4)	SHA-1, SHA-256	#2423

Note: Shaded (olive green) RSA algorithm row above is not used in FIPS Mode 2.

Table 2 Non-approved Algorithms in 32/64 bit sub-module

Algorithm	Implementation details	Certificates
Random Number Generators	ANS X9.31 PRNG (No longer allowed in Approved mode and has been moved to the RNG Historical Validation List)	#1285
AES (non-compliant)	CBC, ECB, CTR, OFB Mode 128, 192, 256 Key length (keys generated by PRNG)	

Table 3 Non-approved but Allowed Algorithms in 32/64 bit sub-module

Algorithm	Implementation details
RSA (FIPS 186-4)	Public key operations for key establishment. The key establishment methodology provides 112 or 128 bits of encryption strength. Modulus sizes 2048 and 3072 bits. The encryption strength depends on the size of the modulus; a 3072-bit RSA key is the minimum required to protect an AES-128 key.
RSA Key Wrapping (FIPS 186-4: OAEP, PKCS#1 v1.5)	(wrap operations) The key establishment methodology provides between 112 and 150 bits of encryption strength. Any key length which is an integer multiple of 256 from 2048 to 4096. The encryption strength depends on the size of the modulus; a 3072-bit RSA key is the minimum required to wrap an AES-128 key.
RSA (FIPS 186-4)	Private key operations for key establishment. The key establishment methodology provides 112 or 128 bits of encryption strength. Modulus sizes 2048 and 3072 bits. The encryption strength depends on the size of the modulus; a 3072-bit RSA key is the minimum required to protect an AES-128 key.
RSA Key Wrapping (FIPS 186-4: OAEP, PKCS#1 v1.5)	(unwrap operations) The key establishment methodology provides between 112 and 150 bits of encryption strength. Any key length which is an integer multiple of 256 from 2048 to 4096.
NDRNG	Non-deterministic random number generator for passively seeding the PRNG and DRBG via an application through an API. The module enforces the minimum amount of entropy & returns an error/exception if it doesn't receive it.

Note: Shaded RSA algorithm rows above are not used in FIPS Mode 2.

RSA modulus encryption strengths are taken from Table 2 of NIST SP800-57 Part 1 Rev 3.

Table 4 Approved Algorithms in 16 bit sub-module

Algorithm	Implementation details/related standard	Certificates
AES (FIPS 197)	CBC, ECB Mode 128, 192, 256 Key length	#2885
AES Key wrap (SP 800-38F)	128, 192, 256 Key length. The key establishment methodology provides between 128 and 256 bits of encryption strength.	#2885
HMAC (FIPS 198-1)	SHA-256	#1819
SHS (FIPS 180-4)	SHA-1, SHA-256	#2426

A module may not generate keys in a non-Approved mode of operation and then switch to an Approved mode of operation and use the generated keys for Approved services and vice versa. The module is implemented in software as an object file that can be linked with other software in order to execute on a general purpose computer (GPC) with either Intel x86/x64 compatible processors or ARM v6 (or above) compatible processors. The module is embodied as multiple-chip standalone. The test environments against which the module was tested were:

Table 5 Test Platforms

Hardware Architecture	Operating System	Exact specification of test platform
Intel x86 16 bit real mode	MS-DOS 6.22	Fujitsu LifeBook S7020 laptop with Intel Pentium M 740 processor
Intel x86	Microsoft Windows 7 Ultimate Edition	Dell D630 with Intel Centrino Duo processor
Intel x64	Microsoft Windows 7 Enterprise Edition	Dell Vostro 1500 with Intel Centrino Duo processor
Intel x64 AES-NI	Microsoft Windows 8.1 Professional	Dell Venue 11 Pro (7130) with Intel Core i5-4300Y (AES-NI) processor
Intel x86	Ubuntu Linux 12.04 LTS	Dell D630 with Intel Centrino Duo processor
Intel x64	Ubuntu Linux 12.04 LTS	Dell Vostro 1500 with Intel Centrino Duo processor
ARM v6	Android v4.2.2	Google Nexus 7 (2012) with NVidia Tegra 3 processor

The vendor confirmed not-tested platforms supported by the tested module are as follows.

- Microsoft Windows XP and above; Intel x86 (32 bit) and x86-64 compatible processors (with and without AES-NI) on platforms supporting PE object format;

- ARM Architecture v6 and above compliant processor platforms supporting the ELF object format;
- Other Intel x86/x64 platforms supporting the ELF object format.

The module continues validation compliance by recompilation as per section G.5 'Maintenance validation compliance of software or firmware cryptographic modules' subsection 1 c) of the FIPS 140-2 Implementation Guidance.

The CMVP makes no statement as to the correct operation of the module or the security strengths of the generated keys when so ported if the specific operational environment is not listed on the validation certificate.

The product meets the overall requirements applicable to FIPS 140-2 level 1.

Users of the library should review transition tables in SP 800-131A and on the CMVP Web site at http://csrc.nist.gov/groups/STM/cmvp/. The data in the tables informs users of the risks associated with using a particular algorithm and a given key length.

Table 6 Security Requirement Levels

Security Requirements Section	Level
Cryptographic Module Specification	1
Cryptographic Module Ports and Interfaces	1
Roles and Services and Authentication	1
Finite State Machine Model	1
Physical Security	1
Operational Environment	1
Cryptographic Key Management	1
EMI/EMC	1
Self-Tests	1
Design Assurance	1
Mitigation of Other Attacks	NA
Overall Level of Validation	1

Cryptographic and Physical Boundaries

The boundaries of the module with respect to FIPS 140-2 cover the physical cryptographic boundary which is the General Purpose Computer (GPC) on which the module is executing and the logical cryptographic boundary of the module itself.

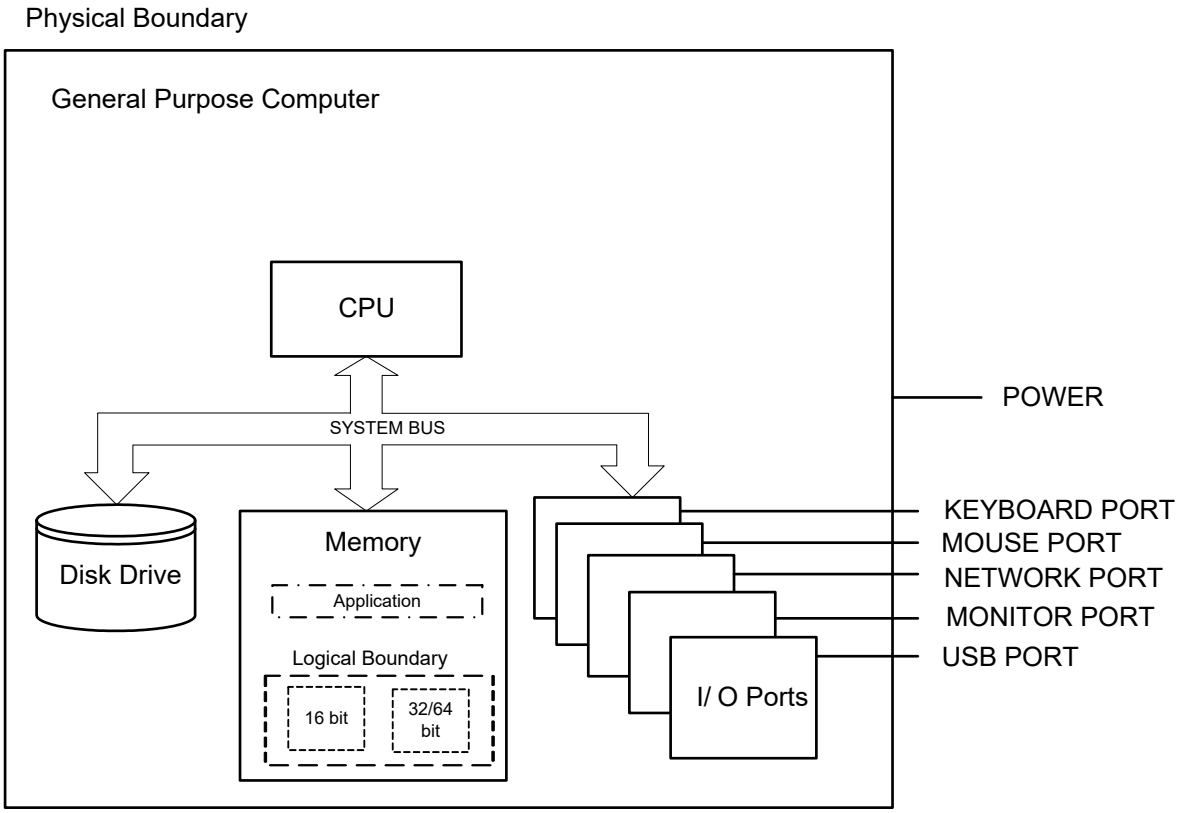

Figure 1 Block diagram of the cryptographic module

Figure 1 illustrates the components related to, and included within, the physical cryptographic boundary of the module. The 16 and 32/64-bit subcomponent objects are illustrated as logical components of an application.

The physical cryptographic boundary includes all the module components within the logical boundary as well as the hardware components of the computing platform the module is installed on which include:

- CPU
- System Bus
- Disk drive
- Memory
- I/O Ports

The hardware component portion of the hybrid cryptographic module is the Intel Core i5-4300Y processor (See Table 6). The processor utilizes AES-NI instruction set implemented in the device for support of the AES algorithm and all the AES related services found in the Roles and Services Tables.

Figure 2 Physical Cryptographic Hardware for AES-NI

The logical cryptographic boundary includes all the software sub-components portion of the hybrid cryptographic module which include:

- 32/64 bit cryptographic object
  - libfipscore.a on Linux / Android / Unix
  - fipscore.lib on Microsoft Windows / UEFI
- 16 bit cryptographic object
  - Becrypt.lib

The 32/64 bit sub-module has a number of distinct platform-related logical embodiments. For example, there are distinct versions of fipscore.lib for 64-bit Windows and 32-bit Windows; these versions can be used on the corresponding UEFI environments, but are distinct from those built for Linux 32 and 64, and also for ARM. All versions are built from identical source code, however.

Module Ports and Interfaces

The 32/64-bit cryptographic module is a single-threaded software-hybrid module which interfaces only with other software. The only physical ports are those of the general purpose computer (GPC) on which it is executing.

The logical interface is an application program interface (API) with C language bindings for the 32 /64 bit sub component and assembler language binding for the 16 bit sub component.

Note that the module must be initialized, by calling the Initialization defined entry point, before any interfaces can be used. No cryptographic interfaces are available until the initialization has taken place. The Initialization API is the module Defined Entry Point for the purposes of FIPS.

Table 7 Interface Table

Interface type	manifests as (Logical Interfaces)	Maps to GPC physical interfaces
Data input	API function input parameters	Keyboard port, Mouse port, Network port, USB port, optical drive, floppy drive
Data output	API function output parameters	Network port, USB port, optical drive, floppy drive
Control input	API function name and control parameters	Keyboard port, Mouse port, Network port, USB port, optical drive, floppy drive
Status output	API function return status and status parameters	PC monitor

In addition to the API interfaces by which applications request service, there are some call-back interfaces (also part of the API) which might be regarded as logical ports. These are:

- RSA private key operation call-back: optional external functions called by the module to perform RSA private key operations when the private key is held externally, e.g. on a smart card.

- Entropy input call-backs: external functions called by the module to obtain entropy of various strengths for use in DRBG seeding, cryptographic padding and probabilistic primality testing.

- Allocator call-backs: external functions providing memory allocation and memory releasing services for the module.

These have equivalent interface types; the cryptographic module supplies the control and data input and responds to the status and data output. The API restricts the information flowing across this interface to the minimum required for the correct operation of the call. The implementation of such call-backs is outside of the scope of this document; however, FIPS-compliant applications providing such call-back implementations must ensure they use only approved cryptographic functionality.

Roles, Services and Authentication

The module supports a crypto officer (CO) role and a user role. The crypto officer and user may be different operators or they may be the same operator performing role-specific module operations. Both the crypto officer and user roles are implicitly assumed. The crypto officer role is implicitly assumed by the operator configuring the module for use at installation time. Crypto officer operations consist of configuring the GPC in single user mode and installing and building applications that use the cryptographic module.

Identification and Authentication

Multiple concurrent operators are not allowed as the module is restricted to single user operation. Operators may change roles while operating the module. The module does not support authentication of the module roles. Becrypt policy concerning the use of the module can be stated thus:

- Operating System administrative privileges are required for an application containing the module to be installed or uninstalled (by the Crypto Officer).
- Once installed, unattended use of the module may continue without further authentication.

Roles and Services

The module supports the services listed in the table below, which identifies the Roles, Cryptographic Keys and CSPs associated with the services. The User role has access to all the services of the module of the CO with the exception of initialization with self tests.

The modes of access are also identified per the explanation.

R - The item is read or referenced by the service (input parameter).

W - The item is written or updated by the service (output parameter).

E - The item is executed by the service. (The item is intrinsic to the cryptographic module.)

Note all Cryptographic keys and CSPs of all modes of access are held in volatile memory.

Table 8 Roles and Services 32/64 bit module

Service	Role	Cryptographic keys and CSPs	Access Type
Install Crypto Module	CO	Becrypt corporate RSA public key	R
Uninstall Crypto Module	CO	None	
Initialization	CO User	See keys involved in Crypto Algorithm Tests and Software Integrity Tests	
Crypto Algorithm Tests	CO User	AES known answer test keys RSA public and private known answer test keys HMAC known answer test key DRBG known answer test PRNG known answer test seed key	E E E E E

Service	Role	Cryptographic keys and CSPs	Access Type
Software Integrity Test	CO User	HMAC integrity test key	E
Show Status	CO User	None	
AES Key Generation	CO User	AES key (plaintext internal format) [DRBG] AES Seed Key (plaintext) [DRBG] DRBG V [DRBG]	W R W
AES Key Creation	CO User	AES key (plaintext) [DRBG] AES key (plaintext internal) [DRBG]	R W
AES Encrypt / Decrypt	CO User	AES data encryption key (plaintext internal format) [DRBG]	R
AES Key Wrap	CO User	AES key encryption key (plaintext external format) AES data encryption key (plaintext external format) AES data encryption key (encrypted)	R R W
AES Key Unwrap	CO User	AES key encryption key (plaintext external format) AES data encryption key (plaintext external format) AES data encryption key (encrypted)	R W R
RSA Key Creation	CO User	RSA public key (plaintext internal format)	W
		RSA key pair (plaintext external format) RSA key pair (plaintext internal format)	W
RSA Key Generation	CO User	RSA private/public key (plaintext external format)	W
RSA Sign	CO User	RSA private key (plaintext internal format)	R
RSA Verify	CO User	RSA public key (plaintext internal format)	R
RSA Key Wrap	CO User	RSA public key (plaintext internal format)	R
RSA Key Unwrap	CO User	RSA private key (plaintext internal format)	R
SHA-1 Message Digest	CO User	None	
SHA-256 Message Digest	CO User	None	
HMAC-SHA-1 Keyed Hash	CO User	HMAC SHA-1 key (plaintext external format)	R
HMAC-SHA-256 Keyed Hash	CO User	HMAC SHA-256 key (plaintext external format)	R
DRBG random number service	CO User	V, Key	None
Memory Allocator services (Non-Crypto)	CO User	None	
Integrity Diagnostic	CO User	HMAC integrity test key	E

Service	Role	Cryptographic keys and CSPs	Access Type
Memory services (set, copy, compare) (Non-Crypto)	CO User	None	
Structure validation services (Non-Crypto)	CO User	None	
Module configuration	CO User	None	
Zeroization (see Table 12 for details)	CO User	AES data encryption key AES key encryption key RSA public key HMAC SHA-1 key HMAC SHA-256 key DRBG key values	W
		RSA key pair	W
PRNG random number service	CO User	Seed, Key	None
AES Key Generation	CO User	AES key (plaintext internal format) [PRNG] AES Seed Key (plaintext) [PRNG] PRNG 'V' value [PRNG]	W R W
AES Key Creation	CO User	AES key (plaintext) [PRNG] AES key (plaintext internal) [PRNG]	R W
AES Encrypt / Decrypt	CO User	AES data encryption key (plaintext internal format) [PRNG]	R
Zeroization (see Table 12 for details)	CO User	AES data encryption key (generated by PRNG) AES key encryption key (generated by PRNG) PRNG key values	W

Note: All service rows above are used in FIPS Mode 1 except shaded (orange) rows which are only used in non-Approved mode. Shaded (olive green) RSA service rows above are not used in FIPS Mode 2.

Table 9 Roles and Services 16 bit module

Service	Role	Cryptographic keys and CSPs	Access Type
Install Crypto Module	CO	Becrypt corporate RSA public key	R
Uninstall Crypto Module	CO	None	
Initialization	CO User	See keys involved in Crypto Algorithm Tests and Software Integrity Tests	
Crypto Algorithm Tests	CO User	AES known answer test keys HMAC known answer test key	E
Software Integrity Test	CO User	HMAC integrity test key	E
AES Key Creation	CO User	AES key (plaintext) AES key (plaintext internal)	R W

Service	Role	Cryptographic keys and CSPs	Access Type
AES Encrypt / Decrypt	CO User	AES data encryption key (plaintext internal format)	R
AES Key Wrap	CO User	AES key encryption key (plaintext external format) AES data encryption key (plaintext external format) AES data encryption key (encrypted)	R R W
AES Key Unwrap	CO User	AES key encryption key (plaintext external format) AES data encryption key (plaintext external format) AES data encryption key (encrypted)	R W R
SHA-1 Message Digest	CO User	None	
SHA-256 Message Digest	CO User	None	
HMAC-SHA-256 Keyed Hash	CO User	HMAC SHA-256 key (plaintext external format)	R
Zeroization (see Table 13 for details)	CO User	AES data encryption key AES key encryption key HMAC SHA-256 key	W

Physical Security

The module is a software-hybrid library and is a multi-chip standalone cryptographic module with the physical security being that of the GPC it is executing on. The module is contained in a hard plastic and metal enclosure which is defined as the cryptographic physical boundary of the module. The module's enclosure is opaque within the visible spectrum. The enclosure of the module has been designed to satisfy Level 1 physical security requirements.

Cryptographic Key Management

This section describes how cryptographic keys are managed by both the 16-bit and the 32/64-bit cryptographic sub-modules. Pre-loaded keys are discussed separately from run time cryptographic keys, even though the former are effectively run-time keys which are not zeroized at runtime.

Pre-loaded cryptographic keys

The following keys are preloaded during the manufacturing process into the binary. They are zeroized by re-formatting the HDD on which the executable containing the library resides.

Table 10 Pre-loaded cryptographic keys

Description	Key type	Used
32/64-bit sub module		
Integrity Check Key	32-byte HMAC-SHA256 key	Key used for approved integrity checking technique used for checking the integrity of the 32/64-bit cryptographic sub-module binaries.
16-bit sub-module		
Integrity Check Key	64-byte HMAC-SHA256 key	Key used for approved integrity checking technique used for checking the integrity of the 16-bit cryptographic sub-module binaries.

Run-time cryptographic keys

The following tables describe how keys generated or established at run-time are handled by the module. The 32/64-bit and 16-bit sub-modules are described in distinct tables.

Memory Management

At run-time, the 32/64-bit sub-module has no writable memory of its own where key values and CSPs can be stored between calls to the module; of necessity, key values and CSPs are held in volatile memory supplied by the caller. Depending on the operation in question, this may be in a buffer directly provided by the caller (e.g. on the stack or the heap), or in memory indirectly provided (normally on the heap) by use of a caller-provided 'allocator' call-back structure. This contains pointers to functions which can respond to memory allocation and freeing requests issued by the sub-module. The sub-module expects newly-allocated memory to be set to all zeroes, but takes responsibility for zeroizing sensitive values prior to freeing them.

The 16-bit sub-module has its own volatile writable memory where it maintains internal forms of Keys and hash contexts.

Zeroizing of Keys, CSPs and Sensitive Data

Unless otherwise stated in the table, the caller of either sub-module is responsible for zeroizing all key values and CSPs after use, using mechanisms appropriate to each item to be zeroized.

When a relevant zeroizing API call is available such as with HMAC keys, that call must be used. Otherwise, zeroizing must be performed by writing a known value (usually 0) to all the bytes of the data/key buffer. The 32/64-bit sub-module provides a function **fips_memset_ext**() while the 16-bit sub-module provides a function **Blank_Key** which may be used for this purpose. (The standard memset() is not safe to use for this purpose as it may be optimized away by the compiler.)

Table 11 Key Management 32/64-bit sub module

Key type and length	Generation	Input	Output	Zeroization	Storage
AES Key 128, 192, 256 bit	Either using NIST SP800-90A DRBG [Approved mode] Or using ANS X9.31 PRNG [legacy use, non-approved mode]	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
RSA Keys or Key Pair [FIPS 186-4] 2048, 3072 bits	FIPS 186-4 section B3.6	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
RSA Keys or Key Pair [FIPS 186-4: OAEP, PKCS#1 v1.5] 2048 to 4096 bits	FIPS 186-4 section B3.6	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
RSA Public Key [FIPS 186-4] 2048, 3072 bits	N/A	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
RSA Public Key [FIPS 186-4: OAEP, PKCS#1 v1.5] 2048 to 4096 bits	N/A	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext

Key type and length	Generation	Input	Output	Zeroization	Storage
RSA Public Key [FIPS 186-4] 1024 bits	N/A	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
HMAC Key (SHA-1) 25, 40, 64, 72, 100 bytes (SHA-256) 16, 30, 64, 80, 128 bytes	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call HMAC finish API	Volatile Memory, plaintext
DRBG V 256, 320, 384 bit	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
DRBG CTR Key 128, 192, 256 bit	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
DRBG Personalisation String Any length	N/A	Electronic Input by API, plaintext	N/A	Zeroized upon HDD re-format.	Volatile Memory, plaintext
Entropy input string	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext
DRBG nonce	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call fips_memset_ext()	Volatile Memory, plaintext

Key type and length	Generation	Input	Output	Zeroization	Storage
DRBG Additional Any length	N/A	Electronic Input by API, plaintext	N/A	Zeroized upon HDD re-format.	Volatile Memory, plaintext
Integrity Check Key (32-byte HMAC-SHA256 key)	N/A	Preloaded during the manufactu ring process into the binary	N/A	Zeroized upon HDD re-format.	Volatile Memory, plaintext

Note: All CSP rows above are used in FIPS Mode 1. Shaded RSA CSP rows above (olive green) are not used in FIPS Mode 2.

Table 12 Key Management 16-bit sub module

Key type and length	Generation	Input	Output	Zeroization	Storage
AES Key 128, 192, 256 bit	N/A	Electronic Input by API, plaintext	Electronic Output by API, plaintext	Caller Responsible to call **Blank_Key**	Volatile Memory, plaintext
HMAC Key (SHA-256) 64 bytes	N/A	Electronic Input by API, plaintext	N/A	Caller Responsible to call HMAC clear key API	Volatile Memory, plaintext
Integrity Check Key (64-byte HMAC-SHA256 key)	N/A	Preloaded during the manufactu ring process into the binary	N/A	Zeroized upon HDD re-format.	Volatile Memory, plaintext

Self-Test

The cryptographic module performs an integrity check of the module at power up and performs a set of known answer tests (KAT) of the cryptographic algorithms as listed in the table below. If FIPS Mode 1 is enabled then all KATs are executed. If FIPS Mode 2 is enabled the RSA KATs are not executed. The KATs are executed at power up or on demand. The KATs can be performed against all FIPS approved algorithms or a subset can be selected (i.e. for FIPS Mode 2).

The failure of the integrity test of one or more KATs is indicated by a non-zero status output at power-up or a zero status when running individual KATs after power-up. The failure of any of the run-time conditional tests is indicated by a non-zero status output from the relevant service.

In the event of a failure of a KAT the module will not perform cryptographic operations for that specific approved algorithm; this affects all services which use that algorithm, which will indicate this by a specific non-zero status output. If a subset of algorithms is tested those algorithms untested are deemed to have failed the KAT and will not perform cryptographic operations. The failure of the integrity test causes a non-zero status output for all cryptographic services.

Once the module has entered a failure state it must be re-initialized and self-tests re-run before cryptographic operations are available. The 16-bit sub-module can be re-initialized by a successful call to its `IInit_Library` service. The 32/64-bit sub-module cannot be re-initialized except by re-loading of the containing executable (i.e. effectively creating and initializing a new instance of the sub-module).

Table 13 Self-Tests

Test	Type	Actions Performed
Critical Functions Tests	Power up	Performed as part of the GPC initialization and operation, e.g. RAM Power-On Self-Test (POST) at boot time, and persistent storage (hard drive) sector check-sum tests.
Module integrity test	Power up	Performed when the module is initialized for use in FIPS mode. A HMAC-SHA-256 of the cryptographic module is performed to verify its integrity for both the 16-bit and 32/64-bit cryptographic objects.
Known Answer Tests	Power up	Performed when the module is initialized for use. For the 32/64-bit cryptographic object the following tests are performed in initialization: AES (Encrypt/Decrypt/Key Wrap), SHA-1, SHA-256, HMAC-SHA-256, RSA (FIPS 186-4 Signature Generation/Verification and Key Pair Generation), PRNG. FIPS Mode 2 initialization omits the RSA FIPS 186-4 Signature Generation and Key Pair Generation tests and marks them as unavailable for use.

		For the 16-bit cryptographic object the following tests are performed: SHA-1, SHA-256, HMAC-SHA-256, AES (Encrypt/Decrypt/Key Wrap).
DRBG health tests	Instantiation	DRBG health checks according to Section 11.3 of SP800-90A. Instantiate and Generate function tests are performed at every instantiation following power-up; an error is returned on failure, and the instance handle is invalid. Since the Generate function tests cover both PR and no-PR cases, the Reseed function is not tested separately at instantiation. Explicit tests for the Instantiate, Generate and Reseed functions may be invoked on a healthy instance; failure makes the instance 'unhealthy' - unusable except for uninstantiation. The Uninstantiate function is tested whenever the above function tests are performed. Failures of the entropy and/or nonce input functions also cause an instance to become unhealthy. This behaviour is tested as part of the Instantiate test. The built-in interval for testing the Generate() function is identical to the interval between operational instantiations; long-running applications should explicitly test the Generate function on an instance at least once an hour if no new instance has been created in that time.
Key generation Conditional test	Conditional	Tests generated entropy data against the previous block generated for the PRNG and DRBG, rejecting if the same. During power-up test, a fixed seed value (PRNG-based key generation) and a fixed-output "test DRBG" (DRBG-based key generation) are used to force a failure of this test to ensure that it is operating as required.
RSA Pair wise consistency test	Conditional	Performs pair wise consistency test each time RSA keys are generated for key wrap, signature generation/verification. (FIPS Mode 1 only; although RSA public keys may be used in FIPS mode 2, there is no private key available against which to perform this test.)

Crypto-Officer and User Guidance

This section describes the configuration, maintenance, and administration of the cryptographic module.

Secure Setup and Initialization

The module must be validated and initialized by successful completion of self tests as documented in the section entitled *Self-Test* above. The self tests must be performed as part of the module initialization and can also be performed on demand by either COs or Users.

The steps to securely initialize the module are as follows:

- Install the module on the target platform (CO only)
- Initialize the required sub-modules: for the 32/64-bit sub-module use `fipsCoreInit`(); for the 16-bit sub-module use `Crypto_Library` specifying the service `IInit_Library`.

`fipsCoreInit()` and `Crypto_Library` are the Default Entry Points (DEP) for the respective sub-modules. If successfully initialized, both functions return a 0 status value and access to the cryptographic functionality of the sub-module is enabled; otherwise a non-zero error status is returned and access to the cryptographic functionality is denied.

Initialization of FIPS Modes in the 32/64 bit sub-module

The 32/64-bit sub-module may be initialized into one of two modes, depending on the algorithms required by the program acting on behalf of the CO/User.

FIPS Mode 1 makes all the approved and allowed algorithms available; FIPS Mode 2 is intended for use where RSA-2 private key functionality is not required, which significantly reduces the time spent in power-up self-test.

The first argument to the `fipsCoreInit()` function is a pointer to a structure which is used to configure the library for use. (The remaining function arguments must be valid for initialization to succeed but do not otherwise affect the configuration.)

The important points for module initialization are:

- correct initialization of the fipscore_config structure;
- provision of an allocator;
- provision of a location in which to place the service table pointer on success;
- checking the return status of fipsCoreInit(), the module's Defined Entry Point.

The following C code fragment illustrates how FIPS MODE 2 may be selected in a user-mode program:

```
#include "fipscore.h"

const struct fipscore_functions * coreAccess = 0;
      /* Pointer to cryptographic service function table */

{
      struct fips_allocator alloc = { calloc, free };
            /* User-mode allocator used during initialization */

      struct fipscore_config config = {}; /* initialize config to zeroes
                                             (FIPS Mode 1 by default) */
      unsigned status;

      config.algs = FIPS_USING_MODE_2;    /* change to request Mode 2 */

      /* Initialize cryptographic library via DEP */

      status = fipsCoreInit(&config, &alloc, 0, &coreAccess);
      if (status != 0) {
            /* crypto library is not successfully initialized */
            ASSERT(!coreAccess);
            exit(1);
      }

      ASSERT(coreAccess);

      /* Use the library via coreAccess.
         Check the established mode as an example. */

      if (coreAccess->core.get_mode() != FIPS_MODE_2) {
            /* This is not the mode we requested. Bail out. */
            exit(1);
      }

      /* use library ... */
```

Figure 3 32/64-bit sub-module initialization - C fragment

The code fragment in Figure 2 is intended for illustrative purposes only; names of variables may differ, and calls to ASSERT() and exit() are not required and in any case would need extra header files to be included. FIPS_USING_MODE_2 and FIPS_MODE_2 are macros representing integer constants and are defined via fipscore.h. What makes this example 'user-mode' is the use of calloc() and free() for memory allocation; a kernel program would use equivalent kernel functions.

In the example, the coreAccess variable is defined at file scope so that, once the library is initialized, any part of the program may use it; this scope is not required, but any part of the application needing access to the module services must have a means to get its value.

To request FIPS Mode 1, config.algs should be set to either 0 or the macro constant FIPS_USING_MODE_1.

Initialization of the 16-bit sub module.

The MASM code fragment in Figure 3 illustrates how the 16-bit sub-module may be initialized.

```
.MODEL TINY
.586
.CODE
INCLUDE cryptoh.inc
; ...
      mov dx, IInit_Library         ; set up service number
      call Crypto_Library           ; perform service via DEP
      cmp ax, NO_ERROR
      jne init_error                ; library not initialized, bail out
;
; we can use the library now ...
```

Figure 4 16-bit sub-module initialization - MASM fragment

As before, this code is ilustrative only; NO_ERROR is defined as 0 in cryptoh.inc. 'init_error' represents a location where the failure to initialize the library is handled. All access to the cryptographic services of the 16-bit sub-module requires calling `Crypto_Library`, which returns a non-zero status output if the library has not been initialized successfully.

Re-initialization

The 16-bit sub-module may be re-initialized by calling `Crypto_Library` specifying the service `IInit_Library`.

The 32/64-bit sub-module cannot be re-initialized without re-loading the executable containing it; in the kernel, this may require rebooting the GPC. An attempt to invoke `fipsCoreInit()` after successful initialization in an executable results in an error status being returned; however, the sub-module state is not affected by this.

Module Security Policy Rules

COs must observe the following practice:

- When performing key generation services, COs are required to use an entropy source with sufficient entropy for generating the seed value(s) to be provided to the Approved DRBG algorithm. See section 'NDRNG guidance' below for details.
- Keys that are entered into or output from the cryptographic boundary must be handled appropriately.
- Long-running processes using a DRBG instance must observe the guidance concerning explicit Generate function tests described opposite 'DRBG health tests' in the table of the next section.

NDRNG guidance

Key generation requires a source of entropy to ensure that the cryptographic strength of generated key meets the FIPS requirements.

The cryptographic module does not have its own source of entropy; it requires a source of entropy outside of the module's logical boundary. For the purpose of this document, this source is called a Non-Approved non-Deterministic Random Number Generator ('NDRNG'); the strength of cryptographic keys generated by the module depends on the entropy output by the NDRNG used. Hence, there is no assurance of the minimum strength of generated keys.

During key generation, NDRNG output is passed as seed entropy to either a DRBG (AES and RSA key generation) or a PRNG (legacy; AES only; unapproved mode). The output of the DRBG (or PRNG) is used in the generation of the keys themselves.

When using DRBG-based key generation, the module specifies the amount of entropy it needs from the NDRNG via the DRBG's `entropy_source` call-back; if the NDRNG is unable to satisfy the request, it must return an error so that key generation fails rather than use insufficient entropy. DRBG requests for nonce values via the same mechanism are counted as requests for entropy.

When using PRNG-based key generation, the entropy input string is passed as data input of the same size as the key to be generated; consequently the data input **must** represent full entropy. The PRNG-based mechanism does not permit the introduction of sufficient additional entropy for generating subsequent AES-256 keys and therefore the PRNG must be re-seeded before generating each AES-256 key. PRNG-based key-generation is **disallowed** after 2015.

The table below provides an indication of the minimum entropy required for generating keys using a single instance of the DRBG or PRNG. The security strength values are based on Table 2 in NIST SP800-57. The entropy required values are the minimum due to the possibility of discarding entropy during the generation process; this particularly affects RSA key generation owing to the probabilistic nature of the method used. Differences in the entropy required for first and subsequent keys reflect the entropy overhead in initializing the DRBG-based key generation mechanism.

Table 14 Minimum entropy requirements for key generation

<table>
<tr><th rowspan="2">Key type and size</th><th rowspan="2">Security strength (bits)</th><th rowspan="2">Deterministic RBG used within module (security strength)</th><th colspan="2">Minimum entropy required (bits)</th></tr>
<tr><th>First key</th><th>Subsequent keys</th></tr>
<tr><td rowspan="2">AES-128</td><td rowspan="2">128</td><td>DRBG (128)</td><td>448</td><td>128</td></tr>
<tr><td>PRNG (128)</td><td>128</td><td>128 (via DT input)</td></tr>
<tr><td>AES-192</td><td>192</td><td>DRBG (192)</td><td>672</td><td>192</td></tr>
<tr><td rowspan="2">AES-256</td><td rowspan="2">256</td><td>DRBG (256)</td><td>896</td><td>256</td></tr>
<tr><td>PRNG (256)</td><td>256</td><td>128 (via DT input); do not use - see above.</td></tr>
<tr><td>RSA, 2048</td><td>112</td><td rowspan="2">DRBG (128)</td><td rowspan="2">960</td><td rowspan="2">768</td></tr>
<tr><td>RSA, 3072</td><td>128</td></tr>
</table>