Reflex Magnetics Cryptographic Library v.1.0 Security Policy

FIPS 140-2
Level 1

Version 1.44
15th May 2007

Contact:

Check Point Software Technologies Ltd.
31-33 Priory Park Road
London NW6 7HP
United Kingdom

Tel: +44 (0)20 7372 6666
Fax: +44 (0)20 7372 2507

Website: http://www.checkpoint.com

Document control

Document Title	Reflex Magnetics Cryptographic Library v1.0 Security Policy	
Version	**Date**	**Description**
0.A	2nd December 2005	First draft for BT review
0.B	21st February 2006	Various revisions after BT meeting
0.C	21st March 2006	Various revisions after BT review, specifically to review Authentication, add Security Rules, more Key attributes and new AES driver APIs.
0.D	13th May 2006	Complete SP revision before submission to BT
0.E	20th May 2006	AES driver API added
1.0	30th May 2006	Final updates
1.1	27th July 2006	Modified in accordance to BT evaluation notes
1.11	31st July 2006	Updated version number
1.2	10th August 2006	Updated according to PR 1-6
1.3	21st August 2006	Updated according to PR3.1, PR5.1, PR6.1
1.31	30th August 2006	Updated according to PR2.1, PR7.0, PR3.2, PR6.2, PR8.0
1.32	6th September 2006	Updated according to PR9, PR11
1.33	13th September 2006	Updated according to PR11.1
1.34	14th September 2006	Updated according to PR10.2, PR12.0
1.38	3rd November 2006	Updated according to PR16, PR17, PR18
1.39	7th November 2006	Updated according to PR18-PR20
1.40	10th November 2006	Updated according to PR19.1-PR20.1
1.41	16th November 2006	Updated according to PR21
1.42	22nd November 2006	Updated according to PR22
1.43	14th May 2007	Updated according to PR23
1.44	15th May 2007	Updated according to PR23.1, logo change

Table of Contents

1. Introduction

1.1. Product and FIPS Validation Identification

SP Title: Reflex Magnetics Cryptographic Library v1.0 Security Policy

SP Version: Version 1.43

Product Name: Reflex Magnetics Cryptographic Library (RCL)

Product Version: 1.0

FIPS Validation Identification: FIPS 140-2
Validation Level: 1

1.2. Purpose

This is a non-proprietary Cryptographic Module Security Policy for the Reflex Magnetics Cryptographic Library v1.0. This Security Policy describes how the Reflex Magnetics Cryptographic Library v1.0 meets the Level 1 security requirements of FIPS 140-2. The product will be validated on Windows XP Professional SP2; it is also capable of running on Microsoft Windows 2000 (and 2003).

This policy was prepared as part of FIPS 140-2 validation of the Reflex Magnetics Cryptographic Library v1.0. FIPS 140-2 (Federal Information Processing Standards Publication 140-2 – Security Requirements for Cryptographic Modules) details the U.S. Government requirements for cryptographic modules. More information about the FIPS 140-2 standard and validation program is available on the NIST website at http://csrc.nist.gov/cryptval/ .

1.3. References

This document deals only with operations and capabilities of the Reflex Magnetics Cryptographic Library v1.0 in the technical terms of a FIPS 140-2 cryptographic module security policy.

Acronym	Definition
AES	Advanced Encryption Standard
API	Application Programming Interface
CBC	Cipher-Block Chaining
CSP	Critical security parameter
DLL	Dynamic Link Library
FIPS	Federal Information Processing Standards
NIST	National Institute of Standards and Technology
OS	Operating System
RCL	Reflex Magnetics Cryptographic Library v1.0
RCL DLL	DLL component of Reflex Magnetics Cryptographic Library v1.0
RCL Driver	NT Kernel Driver component of Reflex Magnetics Cryptographic Library v1.0
RNG	Random Number Generator
RSA	An algorithm for public-key encryption (that takes its initial letters from its 'inventors')
SHA-1	Secure Hash Algorithm 1
SP	Security Policy
SYS	System file (i.e. a kernel mode driver)
TCB	Trusted Control Base
USB	Universal Serial Bus

Table 1-1: Acronyms used within this Security Policy document

2. Reflex Magnetics Cryptographic Library v1.0

2.1. Overview

The Reflex Magnetics Cryptographic Library v1.0 provides cryptographic support for the Check Point Software Technologies Ltd. software products. The Cryptographic Library is used to perform various cryptographic services including encryption/decryption with symmetric and asymmetric algorithms and pseudo random number generation.

The cryptographic keys which are used in these operations are provided by the client applications or generated using the contained pseudo-random number generator. The Reflex Magnetics Cryptographic Library does not have the key storage or key management ability, this functionality is to be provided by the client application.

For the purposes of FIPS 140-2 validation, Reflex Magnetics Cryptographic Library v.1.0 is provided as two binary files:

- dynamic link library rxcrf100.dll (RCL DLL)
- driver rxaes100.sys (RCL driver)

to be installed and used on a computer running Microsoft Windows operating systems.

2.2. Cryptographic Module

The Reflex Magnetics Cryptographic Library v1.0 is classified as a multi-chip standalone module for FIPS 140-2 purposes. The cryptographic module is capable of running on any commercially available IBM compatible PC running the following list of Operating Systems (OS):

- Microsoft Windows XP SP2
- Microsoft Windows 2000 SP4
- Microsoft Windows 2003

The module was tested for FIPS 140-2 compliance on a generic PC running Windows XP Professional Service Pack 2 configured in the single user mode.

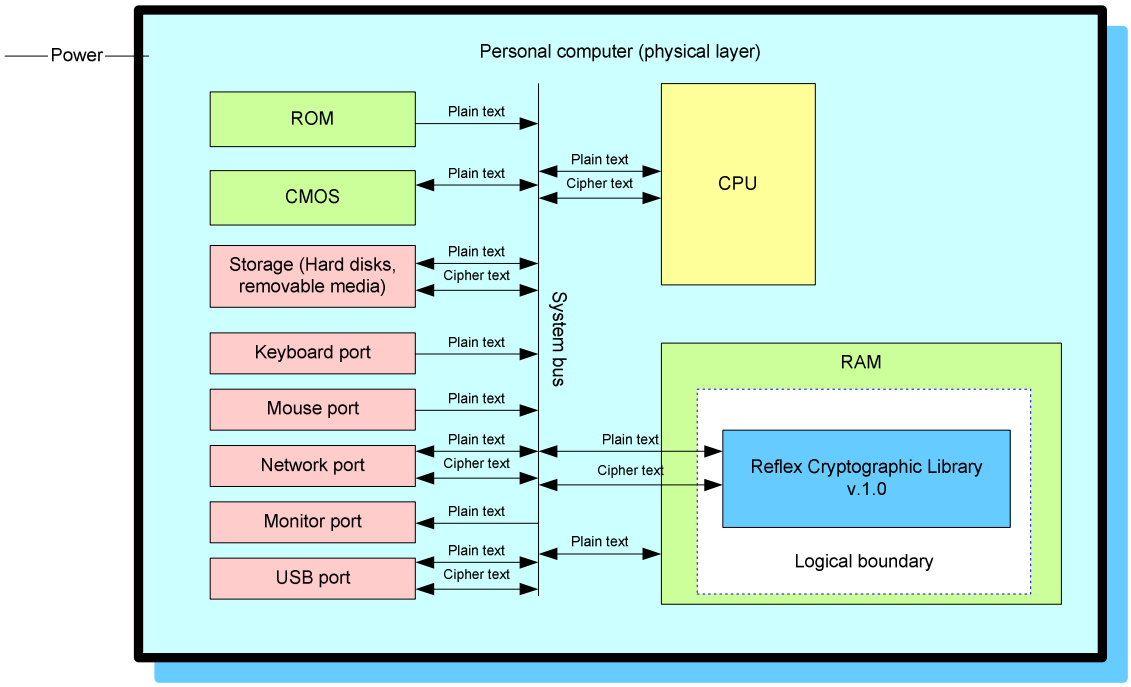
No special configuration is required to operate the module in a FIPS 140-2 mode. The module performs only approved cryptographic algorithms and security functions. Therefore the module is in the approved mode of operation at all times.

2.3. Master component list

2.3.1. Hardware Components

The following components are to be considered hardware components of the module:

- PC case
- CPU
- RAM
- ROM
- CMOS
- Hard Drives
- I/O ports

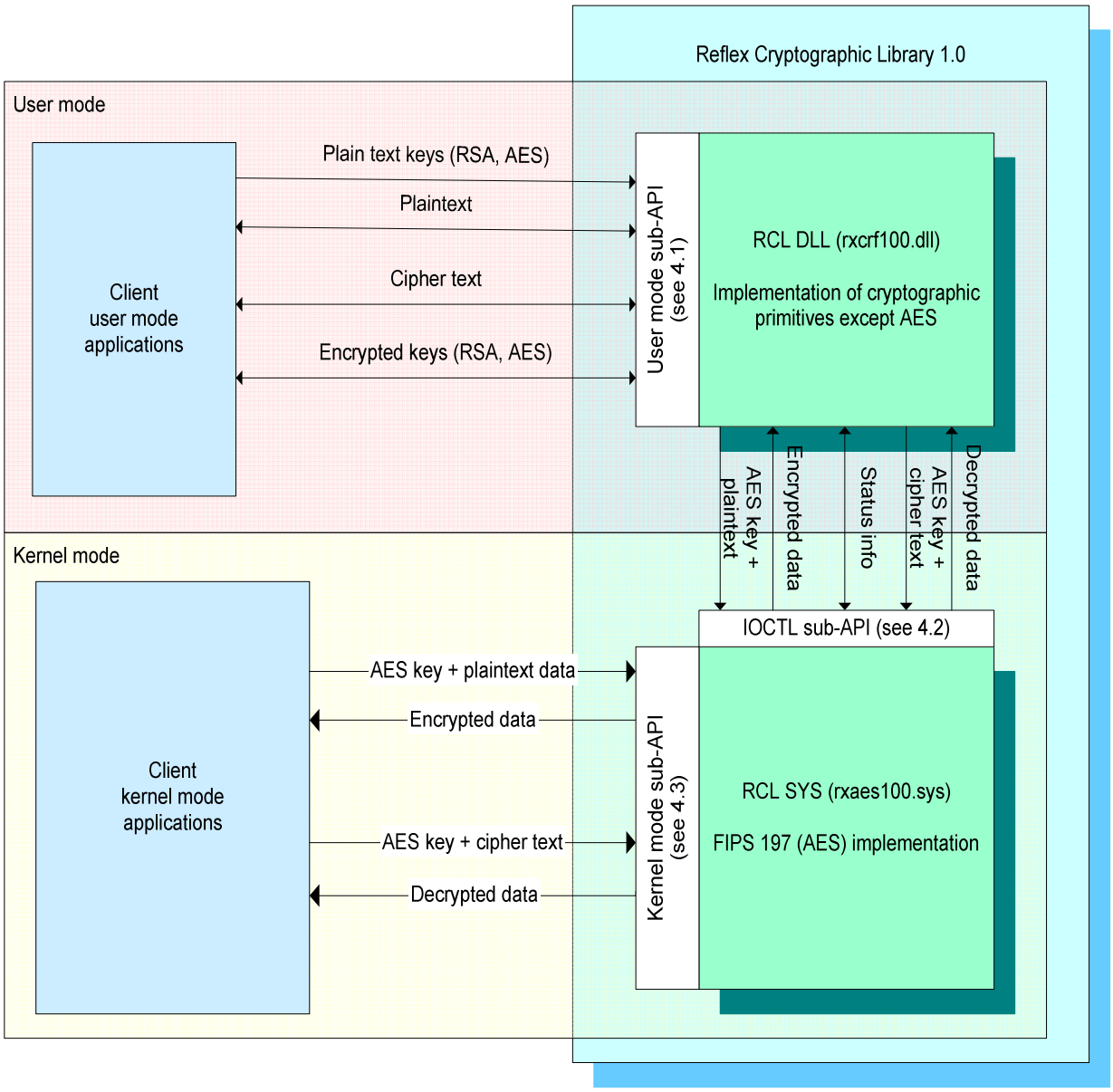


RCL executable code is loaded into the Random Access memory (RAM) and executed by a Central Processing Unit (CPU). All parameters to the module API must be loaded into RAM by a calling process, and data output and status information will also be stored in RAM.

2.3.2. Software components

The following software components comprise Reflex Magnetics Cryptographic Library v. 1.0:

- **RCL DLL (rxcrf100.dll)** is a DLL library containing all cryptographic algorithms implementation except AES. For AES encryption the library calls API exposed by **rxaes100.sys**.
- **RCL Driver (rxaes100.sys)** is executed as a kernel mode driver and encapsulates AES algorithm implementation, accessible by other kernel mode drivers or user mode applications.



For detailed information about the module API see Library API definition.

2.4. Module Ports and Interfaces

The Reflex Magnetics Cryptographic Library v1.0 is classified as a *multi-chip standalone module* for FIPS 140-2 purposes. As such, the module's logical cryptographic boundary includes the library (rxcrf100.dll) and driver (rxaes100.sys) binaries.

The physical boundary is a PC running an operating system and the physical ports of the module include keyboard port, mouse port, serial ports, parallel ports, monitor ports, network port, USB ports and power plug.

The Reflex Magnetics Cryptographic Library v1.0 provides a logical interface via an Application Programming Interface (API). The API provided by the module is mapped to the FIPS 140-2 logical interfaces: data input, data output, control input, and status output. All of these physical interfaces are separated into the logical interfaces from FIPS as described in the following table:

FIPS 140-2 Logical Interface	Physical port	Module logical interface
Data Input Interface	Keyboard port, network port, mouse port, USB port	Parameters passed to the module via API calls
Data Output Interface	Monitor port, network port	Data returned by the module via the API
Control Input Interface	n/a	API function calls
Status Output Interface	Monitor port, network port	Information returned via API return codes. Return code equal to zero indicates a failure.
Power Interface	n/a	Does not provide a separate power or maintenance access interface beyond the power interface provided by the computer Itself

Table 2-1. FIPS 140-2 logical interfaces

The module performs no communications with the physical ports directly, creates no files and does not establish any network connections. All communications between the module and the physical ports of the PC are performed by Microsoft Windows operating system and a process using the module services.

Detailed description of the APIs used is provided in 4.

2.5. Cryptographic Algorithms and Support Functions

The following is a list of algorithms used by the module.

Algorithm \ Support function	Implementation details
AES in CBC mode (128, 192 & 256 bit keys)	FIPS 197
RSA signature	FIPS 186-2, RSASSA_PKCS1_V1_5 with 1024-4096 bit keys
RSA key wrapping	PKCS #1 (RSAES-OAEP or RSAES-PKCS-1_5) used for key wrapping of symmetric keys, as

	allowed by FIPS 140-2 Annex D. with 1024-4096 bit keys.
Key wrapping of AES keys (RFC 3394)	FIPS 197 algorithm is used according to http://csrc.nist.gov/encryption/kms/key-wrap.pdf . Wrapped key must be 128, 192 or 256 bits and therefore is only useful when used as part of AES key establishment technique. The key establishment technique is not a part of the RCL DLL.
SHA-1, SHA-256, SHA-384, SHA-512	FIPS 180-1
Pseudo-random number generation	ANSI X9.31 Appendix A
HMAC-SHA1 (used in module integrity verification only, not available to library users)	FIPS 198 (with Block size 64 bytes and key length 64 bytes or shorter)
RSA key generation	Using ANSI X9.31 Appendix A PRNG (see 2.9.2 for details)

Table 2-2. Supported algorithms

Encryption/decryption keys and random number generator internal state are stored in the library's internal data structures, which are not exposed to external access and protected by operating system protection mechanisms. When a key or random number generator context is set for deletion, the key/generator state is zeroized by overwriting it with 0 bytes to ensure it cannot be recovered.

2.6. Roles, Services and Authentication

The Reflex Magnetics Cryptographic Library v1.0 provides a number of APIs to access its cryptographic services from applications running in both user mode and kernel mode of the Windows operating system. Detailed information about the API is present in 4 below.

The library inherits authentication from the Microsoft Windows operating system upon which it runs. Microsoft Windows requires authentication from a trusted control base (TCB) before a user is able to access system services. Every user that has been authenticated by Microsoft Windows is assigned one of the following two roles supported by the library.

User is any entity that can start a process in the operating system and access services of the Reflex Magnetics Cryptographic Library. The User role is implicitly selected when a process calls any API function in the Reflex Magnetics Cryptographic Library. The User has access to all services of Reflex Magnetics Cryptographic Library.

Crypto Officer is any entity that can install the RCL onto the computer system, configure the operating system, or access services provided by the RCL. All security services of the RCL are available to Crypto Officer. The Crypto Officer has no special access to any keys or data. The Crypto Officer role is implicitly selected when installing the Module or configuring the operating system.

RCL does not support a *Maintenance* role.

There are no audit events nor audit data produced directly by the RCL.

2.7. Physical Security

The Reflex Magnetics Cryptographic Library v1.0 is a software module intended for use with Microsoft Windows XP, Microsoft Windows 2000 or Microsoft Windows 2003 in single user modes on a PC. Since the module is implemented solely in software, the physical security section of FIPS 140-2 is not applicable.

2.8. Operational Environment

The Reflex Magnetics Cryptographic Library is implemented as a loadable module compiled into a Windows DLL binary for the implementation of cryptographic primitives, and a Windows SYS binary for AES implementation.

The Reflex Magnetics Cryptographic Library v1.0 is comprised of two binary files. A software integrity check is run when the binary files are loaded to help ensure that the code has not been accidentally or ineptly modified from its validated configuration.

2.8.1. Multiple Concurrent Operator Roles and Services

The RCL doesn't allow concurrent operators.

For Security Level 1, the operating system has been restricted to a single operator mode of operation, so concurrent operators are explicitly excluded (FIPS 140-2 Sec. 4.6.1).

According to FIPS 140-2 Implementation guidance, when a crypto module is implemented in a server environment, the server application is the user of the cryptographic module. The server application makes the calls to the cryptographic module. Therefore, the server application is the single user of the cryptographic module, even when the server application is serving multiple clients.

If multiple processes that use RCL services are started by an operator, every process will work with separate independent copy of the RCL DLL which does not communicate to any other copy in any way. RCL Driver contains a mutex that ensures that services of the RCL Driver are available to only one process at a time.

2.9. Cryptographic Key Management

The following cryptographic keys are used by Reflex Magnetics Cryptographic Library:

- AES (128, 192 and 256 bits), and created with ***c_AESCreate*** API
- RSA (1024 to 4096 bits) are generated with ***c_RSAGenerate*** API or imported using **c_RSALoad** API (both AES-encrypted and plaintext keys are allowed)

The following pseudorandom number generators are available via library interfaces:

- ANSI X9.31-1998 (can be initialized by seed manually, or seed can be generated automatically from NT performance data), created with ***c_RandomCreate*** method.

Default pseudorandom number generator uses ANSI X9.31-1998 random number generator with AES-256.

Seed key and seed are generated by taking first 384 bits of XOR operation on pseudorandom 512 bit string returned by CryptGenRandom API provided by the Windows operating system's CryptoAPI library, SHA-512 hash value of various NT performance data and optional user-supplied data:

1. NTPERF=collected NT performance data, such as system time, number of running processes, process times, memory usage of every process etc.
2. R=SHA512(NTPERF) XOR CryptGenRandom(64 bytes) XOR OptionalUserData
3. SEEDKEY= bits 0..255 of R
4. SEED= bits 256..383 of R
5. First 128 bits of the SEEDKEY are compared to the first 128 bit of SEED. Random number generator fails to initialise if these two values are equal with an appropriate status code returned to the calling application.

Default library key generator is reseeded automatically every 30 minutes.

SEEDKEY of the Approved RNG is 256 bits and the keys generated by the RCL have from 80 bits (for RSA-1024) to 256 bits (for AES-256) of security (according to NIST Special Publication (SP) 800-57 Part 1, Tables 2-3 in Section 5.6.1 on page 63.).

As key generation uses the approved PRNG, SEEDKEY is of sufficient length, all bits of SEEDKEY contain the randomness from the output from Microsoft CryptoAPI random number generator, compromising the security of the key generation method (e.g., guessing the seed value to initialize the RNG) requires at least as many operations as determining the value of the generated key.

Crypto Officers and library users do not have any direct control over the seed and seed key values chosen in this process.

2.9.1. Key Material

Crypto library uses keys provided by the caller for the following algorithms: AES-128, AES-192, AES-256 and RSA 1024-4096 bits.

2.9.2. Key Generation

The Library generates keys per FIPS requirements, using the Module's Approved RNG (specified in ANSI X9.31–1998, Appendix A) with AES-256 key in CBC mode, in the following manner:

- RSA keys are generated using Module's Approved RNG (ANSI X9.31–1998, Appendix A) using **c_RSAGenerate** API. Key generation procedures described in 4.1.2 of the ANSI X9.31 are not followed.
- AES keys are generated by generate a random string of bytes using either **c_RandomGet** API and passing the generated string to **c_AESCreate**. Alternatively this can be achieved in one call by specifying a NULL key value to **c_AESCreate** API.

2.9.3. Key Entry and Output

AES keys can be imported into the Library DLL via ***c_AESCreate*** in plain text format by calling application providing a RAM address where an octet string of 128, 192 or 256 bits long representing an AES key is stored. As keys are not directly imported to the RCL by means of typing, error detection codes and duplicate key entry tests are not performed.

When imported, AES key can only be exported from the library if encrypted with an RSA key for key transport purposes (***c_RSAEncryptKey*** API).

RSA keys are generated using ANSI X9.31-1998 Appendix A pseudorandom number generator via ***c_RSAGenerate*** API.

The RSA keys can be exported from the library using ***c_RSASave*** interface method with or without private key. When an RSA key is exported with a private key, the export operation succeeds only if an AES-128/192/256 key is specified to encrypt the key.

Previously exported keys can be then imported using ***c_RSALoad*** method.

Each instance of the RCL DLL is owned by a single operator (the *module owner*) and all secret and private keys are thus associated with the module owner. Each key is assigned a unique ID which is returned from **c_RSACreate** / **c_SHACreate** / **c_AESCreate** / **c_RSADecryptKey** API call and is used by other API to uniquely identify a certain instance of the key.

2.9.4. Key Storage

RCL keeps all keys in RAM only and does not store them into persistent storage.

While inside the cryptographic module, the keys are protected from unauthorised disclosure, modification and substitution using memory protection mechanism of the Microsoft Windows XP operating system.

All key copies inside the RCR DLL are destroyed when **c_AESDestroy**/**c_RSADestroy** API is called. Internal copies of the keys are overwritten with zero bytes at this time.

To prevent a key data leak if a client application fails to destroy the key context with **c_AESDestroy**/**c_RSADestroy**, the module will automatically zeroize all keys when the DLL is unloaded.

HMAC keys used for module integrity check are zeroized when RCL DLL is unloaded from memory. RCL DLL is unloaded from memory if module power up self-tests fail or a process using the RCL DLL services terminates.

RCL Driver which contains RCL AES implementation does not provide any key storage facility between calls. All operations with the AES keys are executed in one call, with an AES key and data to be encrypted/decrypted sent to the Driver, with Driver responding with encrypted/decrypted data appropriately. The AES key is never stored in the Driver between the calls; RCL DLL is responsible for key storage in RAM until it is destroyed by the user.

It is the user's responsibility to maintain the security of AES and RSA keys when the keys are outside of the crypto module.

2.9.5. Self Tests

The Reflex Magnetics Cryptographic Library v1.0 performs several power-up self-tests during initialisation.

The following functions are critical to the secure operation of the library:

- AES encryption in CBC mode
- SHA-1, SHA-256, SHA-384 and SHA-512
- Generation and validation of RSA signatures
- Generation of pseudo-random numbers using an Approved PRNG
- HMAC-SHA1, used internally for module integrity verification

The library performs a self-integrity check to verify the module has not been damaged or tampered with. Both RCL Driver and RCL DLL perform their integrity and known answer tests. Should any of the components of the library fail test the module will enter error state as explained in 3.3.

2.9.6. RCL Driver power-up self-tests

When the driver is loaded, a number of self-tests are performed. If any of the tests fail, the driver switches to "Driver self-test failed" state and returns with an error code ACCESS_DENIED in reply to all further requests for its cryptographic services.

2.9.6.1. Integrity Check

The code section of the RCL driver contains a 160 bits value of HMAC-SHA1 checksum. The key used for HMAC-SHA1 operation is stored in the module itself.

During the software integrity self-test, the module first executes HMAC-SHA1 KAT test, then calculates HMAC-SHA1 value over the SYS image, excluding the checksum value, and compares result to the value stored in the SYS file.

2.9.6.2. Cryptographic Algorithm Test

The module performs HMAC-SHA1, AES-128, AES-192 and AES-256 CBC encrypt/decrypt Known Answer Tests.

2.9.7. RCL DLL power-up tests

When the module is loaded into a process, it switches to "Power Up" state (see 3.2).

At this time the tests described in the following sections are performed. If any of the tests fail, module switches to "Self-test failed" state and all further requests to the module will fail with an appropriate error code returned.

To verify if power-up tests have been successful, a user can verify the module status output interface by executing **c_GetStatus** API and checking whether returned value equals CR_STATUS_READY.

2.9.7.1. Integrity Check

The code section of the Rxcrf100.dll driver contains a 160 bits value of HMAC-SHA1 checksum. The key used for HMAC-SHA1 operation is stored in the module itself. During the software integrity self-test, the module the module first executes HMAC-SHA1 KAT test, calculates HMAC-SHA1 value over the DLL image, excluding the checksum value, and verified to match the value stored in the DLL file.

Please note that HMAC-SHA1 implementation is only used for module integrity verification purposes and is not available as service of the RCL library through the library APIs.

2.9.7.2. Cryptographic Algorithm Test

The module performs the following Known Answer Tests:

- HMAC-SHA1 (for module integrity verification only)
- AES-128, AES-192, AES-256 CBC encrypt/decrypt Known Answer Test
- AES-Wrap encrypt/decrypt KAT (for 128,192 and 256 bit keys)
- SHA-1, SHA-256, SHA-384, SHA-512 KAT test
- RSA-1024, RSA-1536, RSA-2048, RSA-3072, RSA-4096 signature verification KAT test
- RSA-1024 key generation

For each algorithm, the tests operate on known values, comparing plaintext, ciphertext, and intermediate data to determine whether the algorithms perform in the Approved manner.

A known answer test for the RNG sets all input parameters to specified values and checks for a specific output value.

2.9.8. Conditional Self Tests

In addition to the initialization self-tests described above, the RCL DLL performs on-going tests during execution as described below. If any of these conditional tests fails, the corresponding module API returns an appropriate error code.

2.9.8.1. Pair-wise Consistency Test

Pairwise consistency test is performed upon each invocation of RSA key generation or when a RSA private key is loaded into the library with **c_RSALoad** API.

During the test the Module signs a sample message using the private key and verifies the signature using the corresponding public key. Then Module encrypts a message with the public key, verifies that the ciphertext differs from the plaintext, decrypts the ciphertext with the private key, and verifies that the decrypted value equals the original message.

2.9.8.2. Continuous Random Number Generator Test

The Module implements a continuous RNG test (as specified in FIPS 140-2, section 4.9.2) that runs each time the Approved RNG is called.

During the test, the previously generated random number is stored as a variable in memory (not on persistent media). The variable is protected by standard operating system protection mechanisms. As the Module's RNG consistently generates fewer than 16 bits (typically as low as 8 bits), the test runs as follows:

1. It stores the first 128 bits for comparison against the next 128 generated bits.
2. It compares each subsequently generated 128 bits against the previously generated 128 bits.
3. It fails if two compared 128-bit sequences are equal.

2.10. Design Assurance

The Reflex Magnetics Cryptographic Library is designed, developed, and deployed in a manner that protects its integrity throughout the process. Check Point Software Technologies Ltd. uses a secure configuration management system, Microsoft Visual SourceSafe 6.0, to ensure the integrity of the Library throughout its development. The system stores distinct versions of the Library source code and documentation.

2.10.1. Internal versioning of the source code and documentation files.

When a source code or a documentation file is added to Visual SourceSafe, it is automatically assigned a unique version number. This version number can be used by developers of the module to identify different versions of the source file or to rollback to a specific version if necessary.

2.10.2. Versioning used for FIPS validation purposes

A FIPS testing laboratory is provided with a number of ZIP files containing the two Library binaries, the test application, source code and other supporting documents serving validation needs.

The file name of each submitted zip file contains the date, which is used as version identifier. All contained files are considered to be of that version.

Testers can use the date/time stamp attribute of individual files inside the zip file to determine whether an individual file has been modified.

Documentation files also have a date and revision number on the title page.

2.10.3. Versioning used for library binaries

Each public release of the library binary files has a unique version number to allow Library users and Crypto Officers to distinguish between different versions of the library. This version number is included in the filenames of the library binaries (for version 1.0 the filenames will be rxcrf**100**.dll, rxaes**100**.sys.

In addition to the public version in the filename, the library binaries have VERSION resources, which contain a unique build number of the binary in format 1.0.0.ymmdd. For example, 1.0.0.60901 means that the library was compiled on 1st of September 2006.

2.10.4. Configuration list

Configuration list is provided in a separate document.

2.11. Security Rules

The RCL adheres to the following security rules:

- The RCL shall only provide NIST-approved cryptographic algorithms.
- The RCL shall support two roles: the Cryptographic Officer and the User role.
- The RCL shall rely on the operating system to provide user authentication and role assignment
- All services implemented within RCL shall be available to both the User and Crypto-officer roles.
- The RCL shall not require human intervention to perform power-on self-tests.
- The RCL shall not support a bypass capability.
- The RCL shall not support a maintenance role or maintenance interface.
- The RCL shall inhibit all data output during self-tests, zeroisation and error states.
- The RCL shall use only Approved algorithms for key generation
- The RCL shall keep all keys and data in memory, without saving them to a persistent storage
- The RCL shall seed its pseudorandom number generation via invoking a noise generator specific to the Windows platform. Pseudorandom number generator shall be seeded with noise derived from the execution environment such that the noise is not predictable.
- The RCL pseudorandom number generator shall be periodically reseeded with unpredictable noise.
- The RCL shall perform a continuous random number generator test upon each invocation of the pseudorandom number generator
- The RCL shall automatically destroy and wipe from memory all copies of keys, pseudo-random generator states and data when no longer needed, or when entering "Power Down" state.

2.12. Mitigation of Other Attacks

The Reflex Magnetics Cryptographic Library v1.0 does not provide security mechanisms to defend against attacks beyond those required by FIPS 140-2 level 1 for monitoring the integrity of the Module.

Reflex Magnetics Cryptographic Library v.1.0 Security Policy

Appendices A, B

FIPS 140-2
Level 1

Version 1.44
15th May 2007

3. Finite State Machine

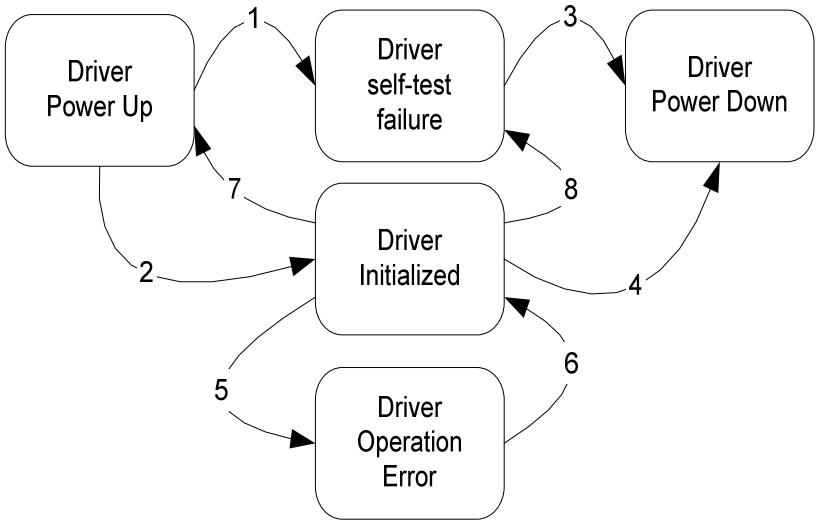
As the cryptographic library is comprised of two separate binary files each exposing a separate set of APIs, finite state machines are provided for RCL DLL and RCL driver separately.

Transitions between states can be automatic or result from user intervention.

3.1. RCL Driver states

State	Description
Driver Power Up	The Driver Power Up state is entered when operating system loader calls the RXAES100.SYS driver entry point function DriverEntry() during system boot.
Driver Power Down	The Driver Power Down state is entered when OS kernel calls the RXAES100.SYS driver's unload function which was set in the DriverUnload field of the DriverObject representing RXAES100.SYS during the Power Up state.
Driver Self-test failure	This state is entered any of the driver self-tests (see 2.9.6) fail.
Driver Initialized	This state is entered if all driver self-tests succeed.
Driver Operation Error	This state is entered when an error occurs while performing a cryptographic operation or loading a key. After reporting the error to the calling application via an appropriate error code, the driver returns to Initialized state.

Table 3-1: RCL Driver States



	Current State	Input	Output	Next State
1	Driver Power Up	Module integrity test fail	Self-test failure	Driver Self-test failure
1	Driver Power Up	Cryptographic tests fail	Self-test failure	Driver Self-test failure
2	Driver Power Up	All self-tests succeeded	No output	Driver Initialized
3	Driver Self-test	Module unloaded by OS	No output	Driver Power Down

	failure	kernel		
4	Driver Initialized	Module unloaded by OS kernel	No output	Driver Power Down
5	Driver Initialized	Invalid parameters passed to the driver's APIs or an internal error occurs.	Operation specific error message	Driver Operation Error
6	Driver Operation Error	Automatic transition	No output	Driver Initialized
7	Driver Initialized	RCL DLL uninitialised	No output	Driver Power Up
8	Driver Initialized	RCL DLL self test failed	Self-test failure	Driver Self-test failure

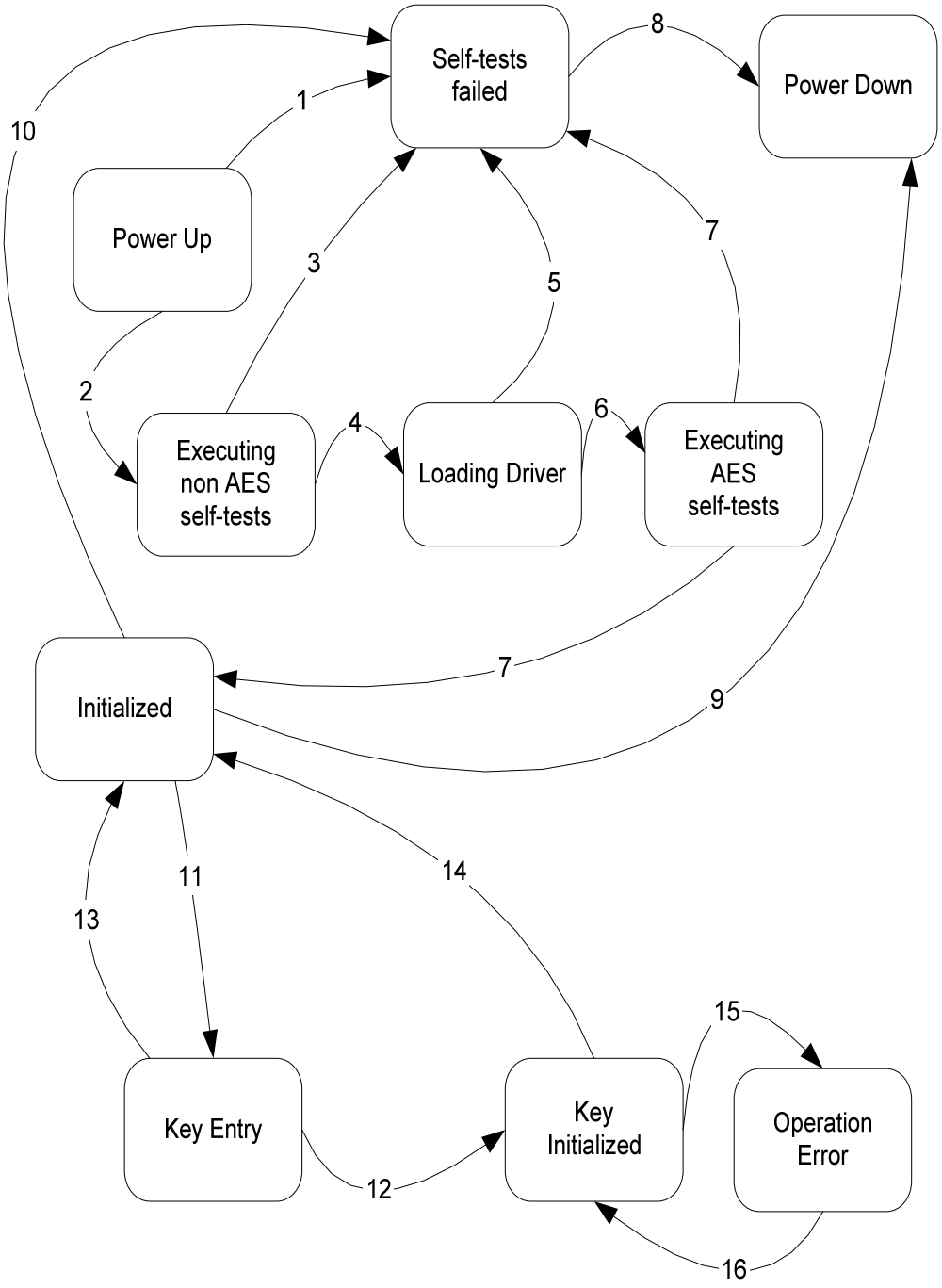
Table 3-2: RCL Driver State transitions

3.2. RCL DLL states

As RCL DLL utilizes services of the RCL Driver, it states are affected by the state of RCL Driver state.

State	**Description**
Power Up	Entered when a client application loads the library DLL into memory.
Loading Driver	Entered when RCL DLL attempts to establish a connection to a RCL Driver
Executing non-AES Self-tests	Entered when RCL DLL executes power-up self-tests
Executing AES self-tests	Entered when RCL DLL has successfully established connection to the driver
Power Down	Entered when a client application unloads the DLL from memory.
Self-test failed	Entered when a library power-up self tests fail. Any request to any library API with fail with an appropriate error code
Initialized	This state is entered if library has loaded successfully and all self-tests have been completed successfully. In this state, the Module is idle, waiting for an operation from the calling application. Successful operations return the Module to this state.
Key Entry	This state is entered when a cryptographic context is created with c_AESCreate/c_RSACreate/c_SHACreate API.
Key Initialized	This state is entered when a valid cryptographic key is loaded for RSA/AES algorithms, or entered automatically for SHA hash algorithm.
Operation Error	This state is entered when an error occurs while performing a cryptographic operation or loading a key. After reporting the error to the calling application via an appropriate error code, the module returns to Initialized state.

Table 3-3: RCL DLL states


Self-tests failed
Power Down
Power Up
Executing non AES self-tests
Loading Driver
Executing AES self-tests
Initialized
Key Entry
Key Initialized
Operation Error
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

	Current State	Input	Output	Next State
1	Power Up	Automatic transition if an error occurs during library initialization	No output	Self-test failed
2	Power Up	Automatic transition if library has loaded into memory successfully	No output	Executing Self-tests
3	Executing non AES Self-tests	Automatic transition if RCL DLL integrity test fails	No output	Self-tests failed
3	Executing non AES Self-tests	Automatic transition if any of the cryptographic tests fails	No output	Self-tests failed
4	Executing non AES Self-tests	Automatic transition if integrity and cryptographic tests succeeded	No output	Loading driver
5	Loading Driver	Automatic transition if a connection to a running instance of RCL Driver cannot be established or the Driver is in invalid state or of incorrect version	No output	Self-tests failed
6	Loading Driver	Automatic transition if RCL Driver is running and is in valid state	No output	Executing AES Self-tests
7	Executing AES Self-tests	Automatic transition if all self tests, AES and non-AES succeeded	No output	Initialized
8	Self-tests failed	DLL is unloaded by client application	No output	Power Down
9	Initialized	DLL is unloaded by client application	No output	Power Down
10	Initialized	Automatic transition if driver DLL has failed	Operation failed	Self-tests failed
10	Initialized	c_SelfTest executed by user failed	DLL initialization error	Self-tests failed
11	Initialized	Algorithm context created with c_AESCreate/c_RSACreate/c_SHACreate API.	No output	Key entry
12	Key entry	Entered key is valid	Success	Key initialized
13	Key entry	Entered key is of invalid length or format	Error specific error message	Initialized
14	Key initialized	algorithm context destroyed with c_AESDestroy/c_RSADestroy/c_SHADestroy	Success	Initialized
15	Key initialized	An error occurs while performing a cryptographic operation.	Operation specific error message	Operation Error
16	Operation Error	Automatic transition	No output	Key Initialized

Table 3-4: RCL DLL State Transitions

3.3. Interaction between RCL DLL and RCL Driver

In order to provide AES encryption services to a User, RCL DLL translates all calls to RCL c_AES* functions to RCL Driver calls.

RCL DLL establishes a connection to RCL Driver while being in “Loading Driver” state, which it automatically enters after being loaded by the library User. If an IOCTL connection to the driver cannot be established, driver not running, or in any other state except “Driver power up”, or another error occurs, transition to “Self-tests failed” state occurs and all subsequent calls to the library will fail.

Because RCL DLL and RCL Driver comprise one cryptographic module, in order to inhibit data output from either part of the cryptographic module in error states, their error states are synchronised an initialisation of the module is performed as follows:

1. RCL Driver is loaded on system startup and switches to "Driver Power Up" state. AES services of the driver are not available in this state.
2. RCL Driver exposes a global event **rxAESStatus** to signal the state of the Driver to the RCL DLL. The RCL DLL checks the status of this event before any data output operation.
4. RCL DLL performs part of the power up tests and instructs the driver to perform its part of the power-up self tests by calling IOCTL_RXAES_SELFTEST API.
5. RCL DLL establishes a connection to the RCL driver (via driver API documented in 4.2)
6. RCL DLL verifies the state of the driver by checking the driver state event, created by driver at step 2. If the event is set, RCL DLL switches to "Self-test failed" state and inhibits the data output.
7. RCL DLL switches to “Initialized” state, as both DLL and Driver part have been successfully initialized and successfully completed power-up tests.
8. When the RCL DLL unloads, its handle to the RCL Driver closes and the Driver switches from "Driver Initialized" back to "Driver Power Up" state.

If an RCL DLL from any process enters the error state, the Driver is informed about this situation by IOCTL_RXAES_SELFTEST_FAILED API (see 4.2) and switched to "Driver self-test failure" state, signaling **rxAESStatus** event. The Driver will remain in this state until restarted. Correspondingly all further calls requesting services from the RCL DLL will also fail, and all instances of RCL DLL will switch to “Self-tests failed” state.

4. Library API definition

The RCL API can be divided into three separate sub-APIs for documentation purposes.

The sub-APIs are (see 2.3.2 above for a diagram):

- RCL DLL API (4.1), to be used by user mode applications. It forwards all AES calls to RCL Driver IOCTL API (4.2)
- RCL Driver IOCTL API (4.2), to be used by the RCL DLL to access AES services of the RCL Driver and synchronise status between the Driver and RCL DLL
- RCL Driver kernel mode API (4.3), to be used by kernel mode applications to access driver's AES services.

All three sub-APIs use the same implementation of AES algorithm in the RCL Driver.

4.1. RCL DLL API description

Most RCL DLL API return TRUE if operation has succeeded and FALSE otherwise, with GetLastError() Windows API returning a more detailed error code.

A brief description of the API follows. More detailed definition of the interfaces is available in crypto.h header file.

4.1.1. Cryptographic Module Self test and status

BOOL c_SelfTest ()

This API can be called by client application to repeat library power-up self tests.

As tests include testing of both non-AES algorithms and AES algorithms implemented in the RCL Driver, **c_SelfTest** API tests both DLL and Driver parts of the RCL.

DWORD c_GetStatus ()

This API can be called by client application to get status of the library.

The returned states value is used to identify the library state:

CR_STATUS_FAILURE RCL DLL failed to load or self tests failed.
CR_STATUS_READY RCL DLL is ready

4.1.2. AES encryption and decryption

```
BOOL c_AESCreate (          IN const BYTE *pbKey, IN DWORD cbKey,
                            IN const BYTE *pbIV, IN DWORD cbIV,
                            OUT PCRD_AES *ppAES)
```

Create an AES CBC context using the provided key and, optionally, initialization vector. cbKey value determines whether AES-128, AES-192 or AES-256 is used.

Parameters

pbKey	*AES key. If NULL, the library will generate a random key using a default PRNG*
cbKey	*length of an AES key. Only 128, 192 and 256 bits keys are permitted.*
pbIV/cbIV	*initialization vector (16 bytes).*
ppAES	*pointer to a variable that will receive an AES context handle. This handle should be later released with c_AESDestroy call.*

```
BOOL c_AESDestroy (         IN PCRD_AES pAES)
```

Destroy AES key context and zeroizes internally stored key and initialization vector.

Parameters
pAES AES context handle to be destroyed.

```
BOOL c_AESGetIV (               IN PCRD_AES pAES,
                                OUT LPBYTE pbIV16bytes)
```

Get current value of IV for an AES key in CBC mode.

Parameters

pAES	*AES context handle.*
pbIV16bytes	*A pointer to the buffer that receives 16 bytes initialisation vector.*

```
BOOL c_AESSetIV (               IN PCRD_AES pAES,
                                IN LPBYTE pbIV16bytes)
```

Set value of IV for an AES key in CBC mode.

Parameters
pAES AES context handle.
pbIV16bytes A pointer to the buffer with 16 bytes initialisation vector.

```
BOOL c_AESEncrypt (             IN PCRD_AES pAES,
                                IN BOOL bFinal,
                                IN OUT LPBYTE pbData,
                                IN OUT DWORD *pcbData,
                                IN DWORD cbDataBufLen)
```

Encrypt data with AES-CBC.

Parameters

pAES	*AES context handle.*
bFinal	*TRUE if this is the last block, FALSE otherwise.*
pbData	*Pointer to the data to be encrypted.*
pcbData	*Pointer to DWORD that on input contains a number of plain text bytes pointed by pbData, and on output a number of encrypted bytes.*
cbDataBufLen	*Size of the buffer pointed by pbData.*

```
BOOL c_AESDecrypt (             IN PCRD_AES pAES,
                                IN BOOL bFinal,
                                IN OUT LPBYTE pbData,
                                IN OUT DWORD *pcbData)
```

Decrypt data with AES-CBC.

Parameters

pAES	AES context handle.
bFinal	TRUE if this is the last block, FALSE otherwise.
PbData	Pointer to the data to be decrypted.
pcbData	Pointer to DWORD that on input contains a number of encrypted bytes pointed by pbData, and on output a number of plain text bytes.

4.1.3. AES key wrap

```
BOOL c_AESWrap (                IN const BYTE *pbKEK,
                                IN DWORD cbKEK,
                                IN const BYTE *pbKey,
                                IN DWORD cbKey,
                                OUT LPBYTE pbWrapped,
```

```
                                    OUT DWORD *pcbWrapped,
                                    IN DWORD  cbWrappedBufSize)
```

Wrap AES key with another AES key, as per http://csrc.nist.gov/encryption/kms/key-wrap.pdf.

Parameters

pbKEK/cbKEK	AES Key-Encryption-Key, that will be used to wrap another key.
pbKey/cbKey	Key to be wrapped
pbWrapped	Pointer to a memory buffer that will receive the wrapped key
pcbWrapped	Pointer to a DWORD, that will receive a number of bytes copied to pbWrapped.
cbWrappedBufSize	Size of the output buffer pointed by *pbWrapped*.

```
BOOL c_AESUnwrap (                  IN const BYTE *pbKEK,
                                    IN DWORD cbKEK,
                                    IN const BYTE *pbWrapped,
                                    IN DWORD cbWrapped,
                                    OUT LPBYTE pbKey,
                                    OUT DWORD *pcbKey,
                                    IN DWORD cbKeyBufSize)
```

Unwrap AES key with another AES key, as per http://csrc.nist.gov/encryption/kms/key-wrap.pdf.

Parameters

pbKEK/cbKEK	AES Key-Encryption-Key, that will be used to unwrap another key.
pbWrapped	Pointer to a buffer containing a wrapped key
cbWrapped	Size of the wrapped key
pbKey	Pointer to a memory buffer that will receive the unwrapped key
pcbKey	Pointer to a DWORD, that will receive a number of bytes copied to *pbKey*.
cbKeyBufSize	Size of the output buffer pointed by *pbKey*.

4.1.4. RSA operations

```
BOOL c_RSACreate (                  IN DWORD dwFlags,
                                    OUT PCRD_RSA *ppRSA)
```

Create an RSA key handle.

Parameters

dwFlags	Flags that currently only affect encryption operations, not signature/verification operations. CR_RSA_OAEP to use encryption in OAEP mode, 0 – PKCS #1 mode.
ppRSA	Pointer to a RSA context.

```
BOOL c_RSAReset (                   IN PCRD_RSA pRSA)
```

Reset all RSA key parameters and zeroize any public or private key used.

Parameters

pRSA	RSA key context.

```
BOOL c_RSADestroy (                 IN PCRD_RSA pRSA)
```

Destroy and zeroize RSA key.

Parameters

pRSA RSA key context to be destroyed.

```
BOOL c_RSAGetKeyHash (          IN PCRD_RSA pRSA,
                                OUT LPBYTE hash20bytes)
```

Get RSA public key SHA-1 hash, as a unique identifier of the used key.

Parameters

pRSA RSA key context.

hash20bytes Pointer to a buffer that will receive a public key hash of the RSA key.

```
BOOL c_RSAGenerate (            IN PCRD_RSA pRSA,
                                IN PCRD_RNG pRNG,
                                IN DWORD dwLength)
```

Generate RSA private and public key pairs using the provided random number generator (or default X9.31 generator initialized from NT performance data). Pairwise consistency test is performed on the generated key.

Parameters

pRSA	RSA key context.
pRNG	RNG generator context (received from c_RandomCreate). Can be NULL to use default RNG.
dwLength	Key length, in bits, of the key to be generated.

```
BOOL c_RSASign (                IN PCRD_RSA pRSA,
                                IN const BYTE *pbHash,
                                IN DWORD cbHash,
                                OUT LPBYTE pbSignature,
                                OUT DWORD *pcbSignature,
                                IN DWORD cbSignatureBufLen)
```

Sign data with RSA private key according to PKCS #1 v.1.5.

Parameters

pRSA	RSA key context.
pbHash	Pointer to SHA-1/SHA-256/SHA-384/SHA-512 hash
cbHash	Size of the hash pointed by *pbHash* in bytes.
pbSignature	Pointer to a variable that will receive a produced signature.
pcbSignature	Pointer to a variable that will receive a length of the produced signature, in bytes.
cbSignatureBufLen	Size of the buffer pointed by *pbSignature*.

```
BOOL c_RSAVerify (              IN PCRD_RSA pRSA,
                                IN const BYTE *pbHash,
                                IN DWORD cbHash,
                                IN const BYTE *pbSignature,
                                IN DWORD cbSignature)
```

Verify RSA signature.

Parameters

pRSA	RSA key context.
pbHash	Pointer to SHA-1/SHA-256/SHA-384/SHA-512 hash
cbHash	Size of the hash pointed by *pbHash* in bytes.
pbSignature	Pointer to a signature.
cbSignature	Length of the signature, in bytes.

```
BOOL c_RSAEncrypt (             IN PCRD_RSA pRSA,
                                IN OUT LPBYTE pbData,
```

```
                                IN OUT DWORD *pcbData,
                                IN DWORD cbBufLen               )
```

Wrap a symmetric key with RSA public key.

Parameters

pRSA	RSA key context.
pbData	Pointer to a first byte of the symmetric key to be encrypted
pcbData	On input, the number of bytes in the symmetric key. On output, the number of encrypted bytes.
cbBufLen	Length of the buffer pointed by pbData.

```
BOOL c_RSADecrypt (             IN PCRD_RSA pRSA,
                                IN OUT LPBYTE pbData,
                                IN OUT DWORD *pcbData)
```

Unwrap an encrypted symmetric key with RSA private key.

Parameters

pRSA	RSA key context.
pbData	Pointer to a first byte of the symmetric key to be decrypted
pcbData	On input, the number of encrypted data. On output, the number of bytes in the decrypted key.

```
BOOL c_RSAEncryptKey(           IN PCRD_RSA                     pRSA,
                                IN PCRD_AES                     pAES,
                                OUT LPBYTE                      pbData,
                                OUT DWORD*                      pcbData,
                                IN DWORD                        cbBufLen);
```

Encrypt an AES key with RSA public key (key transport).

Because RSA encrypted data size does not depend on the size of the input data,

```
DWORD size=0;
c_RSAEncryptKey(pRSA,0,&size,0);
```

may be used to estimate the required encrypted buffer size.

Parameters

pRSA	RSA key handle.
pAES	AES key handle, to be encrypted with RSA
pbData	pointer to the buffer to place encrypted data. Can be NULL to estimate required output buffer size
pcbData	size of the encrypted blob
cbBufLen	size of buffer pointed by pbData.

```
BOOL c_RSADecryptKey(           IN PCRD_RSA pRSA,
                                IN LPBYTE pbData,
                                IN DWORD cbData,
                                OUT PCRD_AES *ppAES)
```

Decrypt an AES key, previously encrypted with c_RSAEncryptKey API.

Parameters

pRSA	RSA key handle.
pbData	pointer to encrypted key data.
cbData	on input - size of encrypted data in the buffer
ppAES	*pointer to an AES key handle.*

```
BOOL c_RSASave (                IN PCRD_RSA pRSA,
                                IN PCRD_AES pAES,
                                IN DWORD dwFlags,
                                OUT PBYTE pbData,
                                OUT PDWORD pcbData,
                                IN DWORD cbBufLen)
```

Save RSA key to a memory blob, with or without private key.

Parameters

pRSA	RSA key context.
pAES	AES key context, to encrypt the exported key. Cannot be 0 if CR_RSA_SAVE_PRIVATEKEY flag is specified.
dwFlags	0 to export public key only, CR_RSA_SAVE_PRIVATEKEY to export both RSA public and private keys.
pbData	Buffer to receive the exported key.
pcbData	The number of bytes copied to buffer pointed by *pbData*.
cbBufLen	Size of the buffer pointed by *pbData*.

```
BOOL c_RSALoad (                IN PCRD_RSA pRSA,
                                IN PCRD_AES pAES,
                                IN DWORD dwFlags,
                                IN const BYTE *pbData,
                                IN DWORD cbData)
```

Load RSA key from a memory blob, with or without private key.

Parameters

pRSA	RSA key context.
pAES	AES key context that will be used to decrypt the key blob, if encrypted.
dwFlags	Reserved, must be 0.
pbData	Buffer that contain the key, previously exported with *c_RSASave*.
cbData	Size of the buffer pointed by *pbData*.

```
DWORD c_RSAIsValid (            IN PCRD_RSA pRSA)
```

Verify that RSA key is valid. Return value is:

- 0 if the key is not valid.
- 1 if the public key is available and key can be used for c_RSAEncrypt/c_RSAVerify operations.
- 2 if private key is available and key can be used for all operations.

Parameters

pRSA	RSA key context.

4.1.5. SHA operations

```
BOOL c_SHACreate (              IN DWORD cbSize,
                                IN PCRD_SHA *ppSHA)
```

Create a SHA hash.

Parameters

cbSize	Length of the SHA hash in bytes (20,32,48,64).
ppSHA	Pointer to a variable to receive a hash context handle.

```
BOOL c_SHAUpdate (              IN PCRD_SHA pSHA,
```

```
                                IN const BYTE *pbData,
                                IN DWORD cbData)
```

Hash data with SHA.

Parameters

pSHA	Hash context
pbData/cbData	Data to be hashed

```
DWORD c_SHAFinal (              IN PCRD_SHA pSHA,
                                IN BYTE *pbData,
                                IN DWORD cbDataBufLen)
```

Finalize SHA hash and retrieve hash value. Returns number of bytes copied to *pbData* buffer or 0 if an error occurs.

Parameters

pSHA	Hash context
pbData	Pointer to the buffer to receive a final hash value. c_SHAGetSize can be used to determine the number of bytes.
cbDataBufLen	Size of the buffer in bytes.

```
DWORD c_SHAGetSize(             IN PCRD_SHA pSHA)
```

Get size of the hash value in bytes.

Parameters

pSHA	Hash context.

```
BOOL c_SHADestroy (             IN PCRD_SHA pSHA)
```

Destroy SHA hash.

Parameters

pSHA	Hash context to be destroyed.

4.1.6. Pseudorandom number generation

```
BOOL c_RandomCreate (           IN DWORD dwFlags,
                                IN LPBYTE pbData,
                                IN DWORD cbData,
                                OUT PCRD_RNG *ppRng)
```

Create and initialize a pseudorandom number generator.

Parameters

dwFlags	0 – create a default ANSI X9.31 Appendix A RNG, initialized from NT performance data. If *pbData/cbData* are non zero, the contents of the buffer pointed by pbData will be XOR-ed with the NT perfomance data and CryptoAPI CryptGenRandom API output . 1 - CR_RNG_ANSI931 create an ANSI X9.31 Appendix A RNG, initialized by the SEED pointed by pbData/cbData
pbData/cbData	See above.
ppRNG	Pointer to a variable that receives the RNG handle.

```
BOOL c_RandomGet (              IN PCRD_RNG pRng,
                                OUT LPBYTE pbData,
                                IN DWORD cbData)
```

Generate pseudorandom data.

Parameters
pRNG RNG handle. May be NULL to use default generator.
pbData/cbData Buffer to receive the generated pseudo-random data.

```
BOOL c_RandomDestroy (          PCRD_RNG pRng)
```
Destroy pseudorandom number generator.

Parameters
pRNG RNG handle.

4.2. RCL Driver IOCTL API description (user mode)

RCL Driver user mode API can be accessed by user mode clients, including RCL DLL, for AES CBC encryption/decryption. Calling the driver API does not affect driver internal state, all state information is passed through API parameters.

This API is accessed through DeviceIoControl Windows API calls to device **\\.\rxAES** with the following control codes (notation is simplified, see rxAesAPI.h and rxAesCommon.h for the exact definitions).

```
BOOL IOCTL_RXAES_VERSION (DWORD *pdwVersion)
```
Return installed AES driver version. Current version is 0x0100.

```
BOOL IOCTL_DNP_AES_INITSUCCESS()
```
Instructs the Driver that DLL has executed its part of power-up tests to execute its part of the power up tests.

```
BOOL IOCTL_DNP_AES_INITFAILED()
```
Informs the driver that user mode part of the library entered an error state and therefore the driver must also enter error state.

```
BOOL IOCTL_DNP_AES_INIT (       IN PAES_INIT paesInit,
                                OUT PAES_CONTEXT paescontext)
```
Convert AES_INIT structure containing a variable-length AES key to AES_CONTEXT structure with AES key schedules (FIPS 197, paragraph 5.2 Key Expansion).

```
BOOL IOCTL_DNP_AES_ENCRYPT (    IN OUT PAES_CONTEXT paescontext,
                                IN OUT BYTE *pbBuffer,
                                DWORD cbBufferSize)
```
Encrypt the passed block with an AES key context, previously obtained through IOCTL_DNP_AES_INIT.

Passed block length must be a multiple of 16.

```
BOOL IOCTL_DNP_AES_DECRYPT (    IN OUT PAES_CONTEXT paescontext,
                                IN OUT BYTE *pbBuffer,
                                DWORD cbBufferSize)
```
Decrypt the passed block with an AES key context, previously obtained through IOCTL_DNP_AES_INIT.

Passed block length must be a multiple of 16.

If the API function succeeds, the returned value is non-zero. In case of failure the returned value is zero and the calling application must handle the error appropriately.

4.3. RCL Driver kernel mode API description

RCL driver kernel mode API can be accessed by kernel mode clients for AES CBC encryption/decryption.

In order to use this API a kernel mode user of the RXAES100.SYS driver must be able to reference the API functions before using them. This is accomplished by building a function table request irp (I/O request packet) and then sending the irp to the driver via the IoCallDriver function. Further information on irp and IoCallDriver can be found on Microsoft Windows XP Driver Development Kit.

Calling the driver API does not affect driver internal state, all state information is passed through API parameters.

If the API function succeeds, the returned value is non-zero. In case of failure the returned value is zero and the calling application must handle the error appropriately.

```
int rxaes_set_key(          IN const BYTE in_key[],
                            IN const ULONG n_bytes,
                            IN const int fType,
                            OUT AES_CRYPTO_CONTEXT *cx);
```

Expands provided variable-length AES key to AES_CRYPTO_CONTEXT structure with AES key schedules (FIPS 197, paragraph 5.2 Key Expansion).

fType parameter indicates whether encryption (1) or decryption (2) or both (3) will be required.

```
int rxaes_BlockEncrypt(     const BYTE *pbInput,
                            BYTE *pbOutput,
                            int cbLength,
                            BYTE *pbIv,
                            AES_CRYPTO_CONTEXT *cx);
```

Encrypt cbLength (must be a multiply of 16) bytes pointed by pbInput with the provided key and initialization vector (16 bytes), and store the result to the buffer pointed by pbOutput parameter.

pbOutput may be equal to pbInput for in-place encryption.

```
int rxaes_BlockDecrypt(     const BYTE *pbInput,
                            BYTE *pbOutput,
                            int cbLength,
                            BYTE *pbIv,
                            AES_CRYPTO_CONTEXT *cx);
```

Decrypt cbLength (must be a multiply of 16) bytes pointed by pbInput with the provided key and initialization vector (16 bytes), and store the result to the buffer pointed by pbOutput parameter.

pbOutput may be equal to pbInput for in-place decryption.