SECURITY POLICY

Crypto++™ Library
Version 5.0.4
FIPS 140-2 Level 1 Validation

http://www.cryptopp.com

Vendor: **Wei Dai**

Version Date: 08/11/2003
Revision: 1.9

Revision History

Date	Revision	Description
January 22, 2002	0.1	Snapshot Draft Document
February 8, 2002	0.2	Snapshot Draft 2 Document
March 15, 2002	0.3	Snapshot Draft 3 Document
May 14, 2002	0.4	Trial Evaluation Draft
June 6, 2002	0.5	Evaluation Draft: Expanded acronyms, Win2K Service Pack 1, Finite State Model updated, Fixed copyright, Crypto Boundary figure clarified, Added Win ME, Skipjack is now “approved”, Added modes (CBC, OFB, etc), clarified listing of approved Encryption / Signing algs, EDC need not be in a protected location. Corrected name of FIPS RNG object, Stated PGP uses 2048 bit keys for signing /verification. Clarified key generation information.
June 27, 2002	0.6	Incorporated CygnaCom comments (CEAL Comments 2 dated 6/21/2002)
July 26, 2002	0.7	Incorporated CygnaCom comments (CEAL Comments 3 dated 7/9/2002)
August 16, 2002	0.8	Incorporated CygnaCom comments (CEAL Comments 4 dated 8/2/2002)
October 16, 2002	0.9	Added RNG test to Power on self test, increment v# and date.
May 1, 2003	1.0	Incorporated NIST/CSE comments.
May 7, 2003	1.1	Fixed copyright notice, Module Interfaces, and Roles and Services
May 27, 2003	1.2	Incorporated more NIST/CSE comments.
June 13, 2003	1.3	Incorporated NIST/CSE verbal comments.
June 26, 2003	1.4	Changed static library to DLL. Draft sent to CEAL.
June 27, 2003	1.5	Fixed previous version. 2nd draft sent to CEAL.
June 30, 2003	1.6	3rd draft sent to CEAL.
July 8, 2003	1.7	Incorporated CEAL’s comments dated 7/7/03. Removed DES and SHA-2 algorithms.
July 16, 2003	1.8	Incorporated CEAL’s comment dated 7/16/03. Added CBC-MAC/TDES in section 8.1.
August 11, 2003	1.9	Incorporated NIST/CSE comments dated 8/1/03.

Copyright Notice

Table of Contents

SECURITY POLICY
FIPS 140-2 LEVEL 1 VALIDATION
CRYPTO++ LIBRARY

1. Introduction

1.1. Purpose

This document specifies the *Security Policy* for the Crypto++ library. This Security Policy was produced as part of the Federal Information Processing Standard (FIPS) 140-2 Level 1 validation of the Crypto++ library version 5.0.4. This document is non-proprietary.

1.2. Documents

The Crypto++ *Security Policy* is provided as part of the following submission package:

- *Security Policy* contains:
  - *Security Policy*
  - *Master Components List* (in Appendix)
  - *Finite State Model* (in Appendix)
- *Crypto Officer and User Guide* contains:
  - *Crypto Officer Guide*
  - *User Guide*
  - *Source Code Description* (in Appendix)
- *Vendor Evidence*
- *API Reference*
- Source Code

A Protection Profile is not required or provided as supporting documentation for this validation.

1.3. Crypto++ Library

Crypto++ is a free open source C++ class library of general-purpose cryptographic algorithms and schemes. It provides a C++ Application Programming Interface (API) for cryptographic functionality such as digital signing and verification, encryption and decryption, hashing, key agreement schemes, key derivation functions, secret sharing, random number generation, and more. The library may be compiled for a number of platforms and operating systems, including Microsoft Windows 95/98/NT/2000/XP, MacOS, Linux, FreeBSD, and other Unix-type operating systems. However only the Windows DLL version has undergone the FIPS 140-2 validation process. For more information, please refer to http://www.cryptopp.com.

2. Module Specification

For the purposes of FIPS 140-2 validation, the Crypto++ library version 5.0.4 is provided as a dynamic link library (DLL), available as cryptopp.dll with build version 5.0.4.0, running on Microsoft Windows operating systems. Henceforth, the DLL package of Crypto++ will simply be referred to as the "Crypto++ library" (or for brevity, just "library").

2.1. Module

In FIPS 140-2 terminology, the Crypto++ library is classified as a *multi-chip standalone Module*. The library was validated as meeting all FIPS 140-2 level 1 physical security and operating system requirements on Microsoft Windows 2000 in single-user mode. The Crypto++ source code and compilations of the Crypto++ source code that have not undergone FIPS validation are not considered FIPS validated. Only version 5.0.4 of the library is FIPS validated.

The Crypto++ library contains only Approved cryptographic algorithms. Non-Approved algorithms implemented in the Crypto++ product are not included in the FIPS *validated* DLL package. Section 4.2 *Services* provides a list of implemented Approved algorithms. The APIs to these functions are exported from the DLL so they can be used by a calling application.

2.2. Boundary

The *physical boundary* for the Module is defined as the enclosure of the computer system on which the functions of the Module execute. The *logical boundary* contains the software modules that comprise the Crypto++ library.

2.3. Hardware Platform

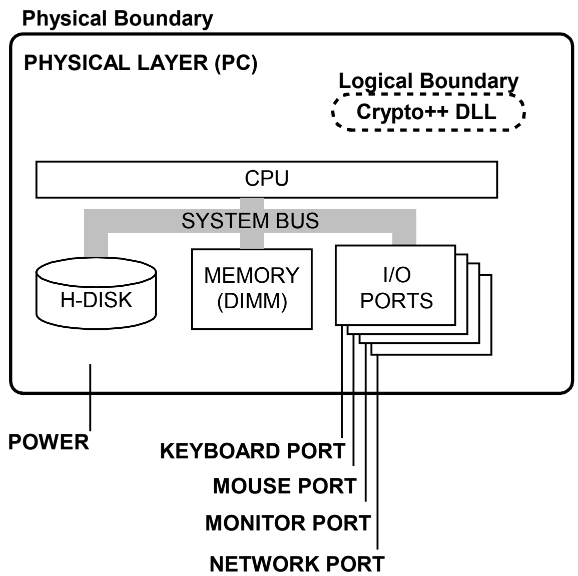
For FIPS 140-2 testing, the library was installed and tested on a Dell OptiPlex GX1 Personal Computer system with:

- an Intel Pentium III 450 MHz processor
- 128 MB system RAM (DIMM)
- 2 serial ports and 1 parallel port
- 4.3 GB hard drive
- Windows 2000 Professional Operating System, Service Pack 1 (operating in single operator mode).

The hardware platform enclosure completely surrounds the entire Module. The enclosure material is standard production-grade material customarily used for this purpose.

2.4. Module Block Diagram

The following block diagram shows the keyboard and mouse ports as physical ports for data or control input, and the monitor port as the physical port for data and status output.



2.5. Software Environment

The software environment in which the Module was validated is the Microsoft Windows 2000 Professional operating system.

The execution platform is a standard commercial off-the-shelf (COTS) computing platform running the Microsoft Windows 2000 operating system. For level 1 Operating System Security, the software Module remains compliant with the FIPS 140-2 validation when operating on any general purpose computer (GPC) since the software of the Module does not require modification when ported (excluding platform specific configuration modifications).

Although not officially tested by the FIPS testing laboratory, the FIPS *validated* Crypto++ library can also execute (without modification) on Windows '95, '98, NT, 2000, and XP. The library does not require use of specialized cryptographic hardware.

2.6. Approved Mode of Operation

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. The list of implemented Approved security functions are described in Section 4.2 *Services*.

3. Module Ports and Interfaces

The *logical interfaces* to the Module consist of a C++ Application Programming Interface (API) exported by the Crypto++ DLL. The *physical interfaces* are standard I/O ports found on a computer for connecting external devices such as monitors and keyboards. (Note: These external devices are outside the physical boundary of the crypto Module and are not part of the Crypto++ validation.)

The following table describes the *logical interfaces* and *physical ports* in more detail:

Interface	Logical Interface	Physical Port
Data Input	Data passed to the API calls to be used by the Module.	Standard Input Port (e.g. Keyboard)
Date Output	Data returned from API calls, generated by the Module	Standard Output Port (e.g. Monitor)
Control Input	Exported API calls	N/A
Status Output	C++ exceptions and the GetPowerUpSelfTestStatus() function	Standard Output Port (e.g. Monitor)
Power	N/A	Supplied by PC

Data entered into and output from the Module are kept separated throughout the software of the Module. The software design maintains separation of the Module's *logical paths*. These *logical paths* are used for *output data* exiting the Module during functions such as key generation and zeroization of cryptographic keys and Critical Security Parameters (CSPs).

The *Crypto++ API Reference* document identifies separate parameters (objects and functions) that behave as the Module's *control input* and *status output*.

4. Roles, Services, and Authentication

The Security Policy states the security rules and operations by which the Module operates. By specifying roles, access controls, services, and security-relevant data items, the Security Policy defines the data items that operators can access while performing specific services in specific roles.

4.1. Roles

The Module supports a *User* role and a *Crypto Officer* role as defined in FIPS 140-2 standard as follows:

User The User is any entity that can access services provided by the Module. The User role is implicitly selected when a process calls any API function in the Module.

Crypto Officer The Crypto Officer is any entity that can install the Module onto the computer system, configure the operating system, or access services provided by the Module. The Crypto Officer may access all services, the same as a User. 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.

The Module does not support a *Maintenance* role.

4.2. Services

The following table provides information about the services available within the Module. To see the detailed interface descriptions for these services, look up the respective implementation object class of function in the *Crypto++ API Reference*.

Service Type	Algorithm	FIPS	Implementation Object Class or Function
Symmetric Cipher	AES	FIPS 197	ECB_Mode<AES>, CTR_Mode<AES>, CBC_Mode<AES>, CFB_Mode<AES>, OFB_Mode<AES>
	Triple DES (2-key)	FIPS 46-3	ECB_Mode<DES_EDE2>, CTR_Mode<DES_EDE2>, CBC_Mode<DES_EDE2>, CFB_Mode<DES_EDE2>, OFB_Mode<DES_EDE2>
	Triple DES (3-key)	FIPS 46-3	ECB_Mode<DES_EDE3>, CTR_Mode<DES_EDE3>, CBC_Mode<DES_EDE3>, CFB_Mode<DES_EDE3>, OFB_Mode<DES_EDE3>
	Skipjack	FIPS 185	ECB_Mode< SKIPJACK>, CTR_Mode< SKIPJACK>, CBC_Mode< SKIPJACK>, CFB_Mode< SKIPJACK>, OFB_Mode< SKIPJACK>
Digital Signature and Key Generation	RSA Signature	--------	RSASSA<PKCS1v15, SHA>[1]
	DSA	FIPS 186-2	DSA
	ECDSA	FIPS 186-2	ECDSA<ECP, SHA>, ECDSA<EC2N, SHA>
Message Digest	SHA-1	FIPS 180-2	SHA
	CBC-MAC/TDES	FIPS 113	CBC_MAC<DES_EDE2>, CBC_MAC<DES_EDE3>
	HMAC/SHA-1	FIPS 198	HMAC<SHA>
Random Number Generator	ANSI X9.31-1998 - Appendix A	-----	AutoSeededX917RNG<DES_EDE3>[2]
Key Establishment	Diffie-Hellman Key Agreement	-----	DH
	RSA Key Transport	-----	RSAES<OAEP<SHA> >
Other Functions	Self-Test	N/A	DoPowerUpSelfTest

[1] The RSA signature algorithm implemented is specified in PKCS #1 version 2.0 as RSASSA-PKCS1-v1_5.
[2] The RNG is seeded using the CryptGenRandom API provided by the Windows operating system's CryptoAPI library.

	Self-Test Status	N/A	GetPowerUpSelfTestStatus

The following table identifies CSPs and types of available access for the supported services.

Service	Cryptographic Keys and CSPs	Type(s) of Access (e.g., RWE)
ANSI X9.31-1998 - Appendix A Random Number Generation	Seed value, seed key, random number	No operator access to the seed, which is generated internally. RW access to the random number.
AES, TDES, Skipjack Encryption and Decryption	Secret key	RW
RSA, DSA, ECDSA Signing	Private key	RW
RSA, DSA, ECDSA Verification	Public Key	RW
RSA Key Transport, DH Key Agreement	Private and Public Keys	RW
SHA-1 Hashing	Checksum	RW
CBC-MAC/TDES, HMAC/SHA-1 Generation	Secret key and checksum	RW

There is presently no Approved secret sharing algorithm.

4.3. Authentication

Within the constraints of FIPS 140-2 level 1, the Module does not directly implement User authentication; it depends on the operating system for operator authentication.

5. Finite State Model

See "*APPENDIX B: Finite State Model*".

6. Physical Security

The Module was tested while executing on a standard Intel-compatible personal computer platform. This platform (and other Intel compatible platforms) and the executing software comprise a multi-chip standalone Module that includes standard, production grade components, standard passivation, and an enclosure of production grade strength, meeting all FIPS 140-2 level 1 physical security requirements.

7. Operational Environment

7.1. Operating System Requirements

Each user process in the operating system has its own virtual address space with its own copy of the executable code. When a process loads the Crypto++ DLL (Module), it maps the Module into its own virtual address space and then calls the DLL's exported functions. The Module uses and allocates memory from the virtual address space of the calling process.

The Module is completely independent and in its own process. The Module itself does not communicate with other processes, for example, using any operating system inter-process communication mechanisms. So no other process can access private and secret keys or other CSPs.

The Module is restricted to a *Single Operator Mode of Operation*, per FIPS requirements. The operating system is responsible for multitasking operations so that other processes cannot intervene when the Module is active at a particular instance in time.

7.2. Module Integrity

The security of the Module does not depend on secrecy of the code contained in the Module. Being open source, the library's code is accessible to all. However, the integrity of the *validated* Module is verified through the self-tests described in Section 9 "*Self-Tests*". These tests limit opportunities for keys or other CSPs to be disclosed inadvertently.

7.3. Other Assumptions

Proper FIPS configuration and usage of the Module requires following instructions in the *Crypto Officer Guide* and *User Guide*, and following the rules described in Section 4 *"Roles, Services, and Authentication"*.

8. Cryptographic Key Management

All keys in the Module may be either imported into the Module or internally generated using the Module's random number generator (RNG). The Module itself keeps these keys in memory only and does not store them in persistent media.

8.1. Key Generation

The Module generates keys per FIPS requirements, using the Module's Approved RNG (specified in ANSI X9.31–1998, Appendix A), in the following manner:

- DSA keys are generated according to procedures described in FIPS 186-2.
- ECDSA keys are generated according to procedures described in ANSI X9.62.
- RSA keys are generated according to procedures described in ANSI X9.31.
- The remaining keys (AES, TDES, CBC-MAC/TDES, Skipjack, HMAC/SHA-1) are generated using the Module's Approved RNG (by generating a random octet string of suitable size).

Intermediate key generation values are not output from the Module during the key generation process. The Crypto++ API header files (*rsa.h*, *dsa.h*, and *rng.h*) provide more information on key formats and structures.

8.2. Key Establishment

In the absence of a FIPS-approved key establishment method, FIPS (Annex D to FIPS 140-2) allows the following commercially available methods to be used in Approved mode of operation: RSA Key Transport and Diffie-Hellman Key Agreement. Crypto++ provides APIs for the calling application to use these algorithms.

8.3. Key Entry and Output

Keys are entered into and output from the Module in plaintext form through the C++ API. The Module also provides APIs for a calling application to wrap keys for output using RSA key transport. APIs are also provided for a calling application to sign and verify signatures on keys. The module does not create certificates for keys.

The module generates seeds and seed keys for its random number generator using the CryptGenRandom API provided by the Windows operating system's CryptoAPI library. Neither Users nor Crypto Officers have any direct control over generation and entry of these seeds and seed keys.

8.4. Key Storage

The Module does not store or archive keys in any persistent storage media.

8.5. Key Destruction

The Module stores keys while they are in use *in memory* only. When the C++ object that encapsulates a key is destroyed, the Module automatically zeroizes the key.

It is possible that the operating system may swap memory that contains keys to disk. To zeroize those keys, the User must wipe the swap files. One way to accomplish this is to reformat the hard drive(s) containing the swap file.

9. Self-Tests

The Crypto++ library implements both power-up and conditional self-tests to ensure proper operation of the Approved cryptographic algorithms and security functions.

9.1. Power-up Self-tests

When the Crypto++ DLL is loaded into a process, it performs a suite of power-up self-tests to ensure the integrity and correct operation of the cryptographic services. The self-tests always run automatically when the DLL is loaded, and do not require any inputs from or actions by the operator. If any self-test fails, the Module enters an error state and prevents any cryptographic operation from being performed. "*APPENDIX B: Finite State Model"* provides detail of the state transitions. This section describes the power-up self-tests implemented by the Module.

9.1.1. Cryptographic Algorithm Test

The Module performs known answer tests for AES, TDES, Skipjack, SHA-1, HMAC/SHA-1, and RSA Signature. 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.

9.1.2. Software Integrity Check

The read-only data section of the Crypto++ DLL contains an HMAC/SHA-1 secret key and value. The value is computed by applying an HMAC over the DLL image, *excluding* the stored HMAC value in the DLL. During the software integrity self-test, this value is recomputed and verified to match the value stored in the DLL.

9.1.3. Pair-wise Consistency Test

The Module performs pair-wise consistency tests on DSA and ECDSA as described in Section 9.2.1. In contrast to the conditional testing, this power-up self-test verifies the operations on fixed (hard-coded) key pairs. (Since the output of the RSA Signature algorithm is deterministic, it is tested as part of the known answer test described in Section 9.1.1.)

9.1.4. On-Demand Self-test

The Module exports an API routine, *DoPowerUpSelfTest*, which can be called to initiate the self-tests on demand. Minimally, Users can manually initiate the power up self-tests by resetting (restarting) the application.

9.2. Conditional Tests

In addition to the power-up self-tests described above, the Module performs on-going tests during execution as described below. If any of these conditional tests fails, the Module throws an exception.

9.2.1. Pair-wise Consistency Test

The Module runs a pair-wise consistency test (as specified in FIPS 140-2, section 4.9.2) each time an asymmetric key pair is generated. For the signature keys (i.e., DSA, ECDSA, RSA signature), the Module signs a message using the private key and verifies the signature using the corresponding public key. For key transport keys (i.e., RSA encryption), the 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. For key agreement keys (i.e., DH), the Module creates a second compatible keypair, performs both sides of the key agreement algorithm, and verifies that the resulting secret keys are equal.

9.2.2. 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.

10. Design Assurance

The Module is designed, developed, and deployed in a manner that protects its integrity throughout the process. Guidance is provided to Crypto Officers and Users of the Module.

10.1. Configuration Management

10.1.1. Source Code Management System

The vendor uses a secure configuration management system, Microsoft Visual SourceSafe 6.0, to ensure the integrity of the Module throughout its development. The system stores distinct versions of the Module's source code. While in storage, files are protected against unauthorized modification, with password-based authentication required. Currently, only authorized developers at Groove Networks, Inc. are allowed modification access to the source files.

10.1.2. Versioning

Internal. Whenever a new version of a file is stored in the configuration management system, it is labeled with a unique version number. These version numbers are used internally to allow developers to roll back to previous versions of a Module, if necessary. These internally managed version numbers are not seen by end-users.

FIPS Testing. A FIPS testing laboratory is provided with a zip file containing the Crypto++ DLL, the test application, source code and other supporting documents serving evaluation needs. The file name of each submitted zip container file contains the date and time stamp. All contained files are considered to be of that version. Evaluators can use the date/time stamp attribute of individual files to determine whether an individual file has been modified. Documentation files will have a date and revision number on the title page.

User Delivery. Each public release of the product carries a unique product version number. This allows users to distinguish the *validated* product from other product versions. The product uses a conventional version numbering scheme (for example "version 5.0.4") in which major releases are noted by incrementing the unit digit and minor updates to a release are noted by incrementing the decimal digit(s). A change log available with each product version describes the associated changes made for that release.

10.2. Delivery and Operation

The *Source Code Description* and *Crypto++ API Reference* documents provide more information on the design and implementation of the Module.

In addition to understanding the versioning information provided in the previous section, application developers should ensure that the proper version of the Module is delivered by verifying a Pretty Good Privacy (PGP) signature on the Crypto++ DLL. The *Crypto*

Officer Guide explains how to verify the signature on the library. The following *fingerprint*[3] identifies the PGP (2048-bit RSA) public key that is used to sign the library:

F1F2 7D64 0CAA 3C65 763D 2508 F190 1AEB 0454 9843

10.3. Guidance Documents

The *Crypto Officer Guide* instructs Crypto Officers to properly install, configure, and maintain the Module. The *User Guide* and *Crypto++ API Reference* explain the proper and complete use of the library's Approved cryptographic services and functions.

11. Mitigation of Other Attacks

The Module 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.

12. References

For more information about the Crypto++ library, please visit the product website at http://www.cryptopp.com. The following documents were used to support validation of the Crypto++ library.

[1] National Institute of Standards and Technology, *Security Requirements for Cryptographic Modules*. FIPS 140-2, 25 May, 2001.

[2] National Institute of Standards and Technology, *Derived Test Requirements for FIPS PUB 140-2, Security Requirements for Cryptographic Modules*, Draft, November 15, 2001.

[3] FIPS 197 *Advanced Encryption Standard (AES)*

[4] FIPS 46-3 *Data Encryption Standard (DES)*

[5] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, FIPS 186-2, October 5, 2001.

[6] American Bankers Association, *Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA),* ANSI X9.31-1998.

[7] American Bankers Association, *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA),* ANSI X9.62-1998.

[8] National Institute of Standards and Technology, *Secure Hash Standard*, FIPS 180-1, April 17, 1995.

[9] National Institute of Standards and Technology, *Keyed-Hash Message Authentication Code (HMAC)* FIPS 198, issued March 6, 2002.

[10] RSA Laboratories. *PKCS #1: RSA Encryption Standard.* Version 2.0, September 1998.

[3] A fingerprint is a SHA1 hash of a public key. It uniquely identifies the key and is easier for humans to read than the public key value that is much longer.

APPENDIX A: Master Components List

The Crypto++ library is software that is intended to operate as part of an application on a personal computer platform under the Windows® 2000 Professional operating system.[4] The Module includes the *validated* Crypto++ DLL and the hardware elements of the personal computer platform. Neither the operating system nor the calling application is a component of the Module.

A.1. Hardware Components

The following listed hardware elements are components of the Module. The components are standard production-quality integrated circuits or components designed to meet commercial-grade specifications for power, temperature, reliability, shock, vibration, and so on.

- The PC enclosure
- The central processing unit (CPU)
- The hard drive
- Memory
- CD-ROM drive
- Floppy disk drive

A.2. Software Components

The following listed software elements are components of the Module.

- The Crypto++ 5.0.4 DLL

The *Source Code Description* lists all the software Modules that make up the Crypto++ DLL.

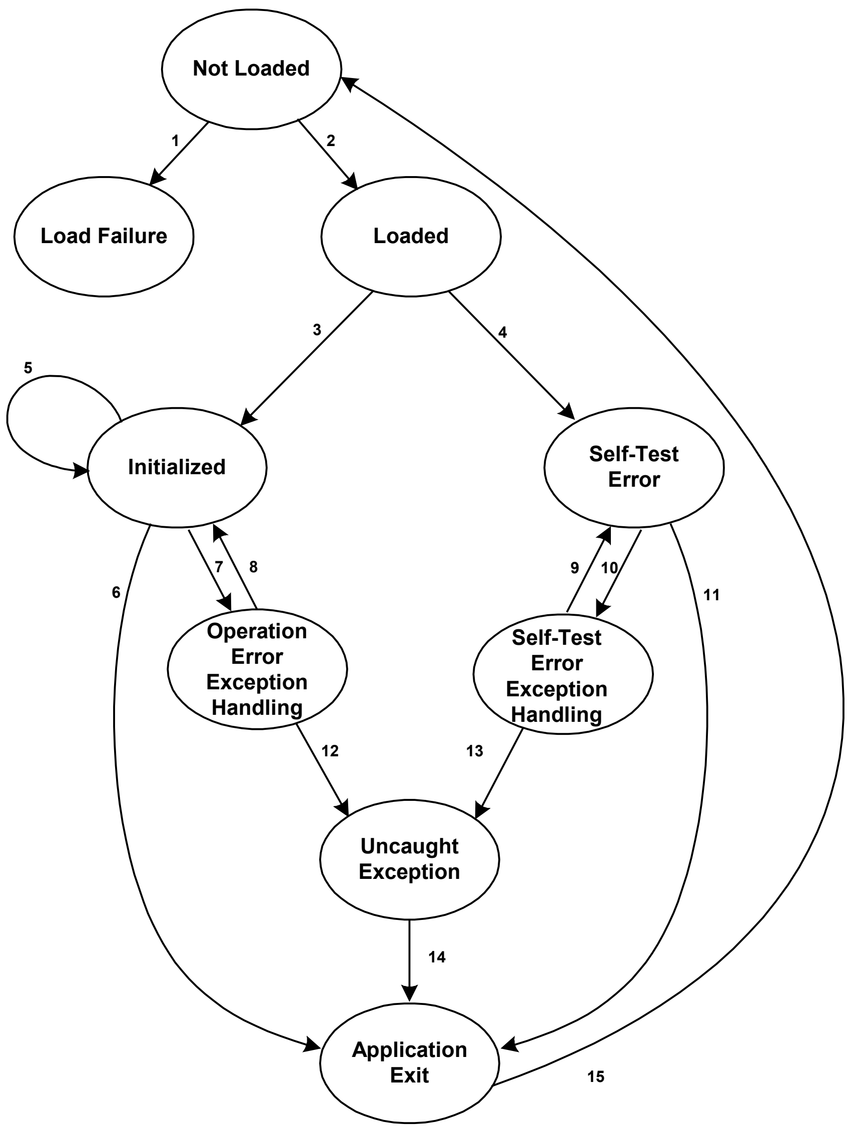
[4] See Section 2.5 *Software Environment* for other qualified operating systems.

APPENDIX B: Finite State Model

This section describes the finite state model of the Crypto++ library. The Module can be in only one state at a time. State transitions are driven by various software events or actions.

Crypto Officer operations are outside the scope of this state model.

B.1. Diagram



B.2. Descriptions

B.2.1. Not Loaded

The Module is in the *Not Loaded* state when it has not been loaded into memory. The *Not Loaded* state corresponds to the *Power-Off* state defined in FIPS 140-2.

B.2.2. Loaded

The *Loaded* state corresponds to the *Power-On* state defined in FIPS 140-2.

In the *Loaded* state, the Module is running but the power-on self-test has not yet run. The Module immediately runs the power-on self-test and automatically transitions to the next state. If the self-test runs successfully, the Module transitions to the *Initialized* state. On self-test failure, the Module transitions to the *Self-Test Error* state.

B.2.3. Load Failure

The Module enters the *Load Failure* state for any load failure such as file not found, incorrect parameter passed or other error.

B.2.4. Self-Test Error

The Module enters the *Self-Test Error* state if any self-test fails, setting a global self-test error flag. This flag lets the application detect the *Self-Test Error* state and handle the error, performing (for instance) a graceful program termination if that is the appropriate action.

If the application does not check the flag and tries to perform a cryptographic operation, an exception is thrown and the Module enters the *Self-Test Exception Handling* state.

B.2.5. Initialized

The Module enters the *Initialized* state after all self-tests pass. In this state, the Module is idle, waiting for an operation from the calling application. Successful operations return the Module to this state.

B.2.6. Operation Error Exception Handling

The Module enters the *Operation Error Exception Handling* state by throwing a C++ exception after any security operation fails. In this state the calling application may attempt to catch the exception. If the exception is caught, then the Module transitions back to the *Initialized* state. Otherwise the Module transitions to the *Uncaught Exception* state.

B.2.7. Self-Test Error Exception Handling

The Module enters the *Self-Test Error Exception Handling* state by throwing a C++ exception when the calling application attempts to perform a security operation while the

Module is in the *Self-Test Error* state. In this state, the calling application may attempt to catch the exception. If the exception is caught, the Module transitions back to the *Self-Test Error* state. Otherwise the Module transitions to the *Uncaught Exception* state.

B.2.8. Uncaught Exception

The Module enters the *Uncaught Exception* state when the calling application fails to catch the C++ exception thrown by the Module when it entered the *Operation Error Exception Handling* state or the *Self-Test Error Exception Handling* state. In this state, the Module throws the exception up the stack until it is caught and handled by a higher layer in the stack or it is not caught and the program exits.

B.2.9. Application Exit

The Module enters the *Application Exit* state when the user exits the application normally from the *Initialized* state. Abnormal conditions causing transitions to this state are uncaught exceptions, and self-test errors.

B.3. Transition Conditions and Events

This section describes the conditions or events that cause transitions (numbered) in the State Diagram.

1. Load fails due to file not found error, incorrect parameter, or other error.
2. Load succeeds.
3. Self-test succeeds. This transition occurs when all self-tests pass.
4. Self-test fails. This transition occurs when any self-test fails.
5. User operation succeeds.
6. User exits application.
7. User operation failure due to invalid key, algorithm parameters, or other error. Exception thrown.
8. User operation exception caught.
9. Self-test error exception caught.
10. Operation attempted while in *Self-Test error* state. Exception thrown.
11. Self-test error flag detected. Program terminates.
12. User operation exception not caught.
13. Self-test error exception not caught.
14. Automatic Transition. Program terminates.
15. Automatic Transition. Program unloaded from memory.

	Current State	Input	Output	Next State
1	Not Loaded	Load Failure	Load Failure Error Message	Load Failure
2	Not Loaded	Load Success	Load Success Message	Loaded
3	Loaded	Run self-test automatically.	Self-Test Success	Initialized
4	Loaded	Run self-test automatically.	Self-Test Failure	Self-Test Error
5	Initialized	Cryptographic operation	Operation success	Initialized
6	Initialized	User exits program	Operation success	Application Exit
7	Initialized	Cryptographic operation	Operation failure	Exception Handling

8	Operation Error ExceptionHandling	Catch exception	Exception caught	Initialized
9	Self-Test Error Exception Handling	Catch exception	Exception caught	Self-Test Error
10	Self-Test Error	Attempt Cryptographic operation	Throw Exception	Self-Test Exception Handling
11	Self-Test Error	Check Self-Test Error Flag	Error flag detected	Application Exit
12	Operation Error ExceptionHandling	Not catch exception	Exception not caught	Uncaught Exception
13	Self-Test Error Exception Handling	Not catch exception	Exception not caught	Uncaught Exception
14	Uncaught Exception	Automatic transition	No output	Application Exit
15	Application Exit	Automatic transition	Application and crypto Module unloaded from memory	Not Loaded