# CRYPTO OFFICER GUIDE and USER GUIDE

## Crypto++™ Library
## Version 5.0.4 and 5.2.3
## FIPS 140-2 Level 1 Validation

http://www.cryptopp.com

**Vendor**: **Wei Dai**

**Version Date:** 12/16/2004
**Revision**: 0.1

## Revision History

| Date | Revision | Description |
|---|---|---|
| December 16, 2004 | 0.1 | Creation Date |

# TABLE OF CONTENTS

# 1 Introduction

## 1.1 Audience

This document is written for developers and administrators who use or administer the FIPS *validated* Crypto++ library, versions 5.0.4 and 5.2.3. The library has been formally evaluated by a NIST-approved National Voluntary Laboratory Accreditation Program and has been certified as conforming with FIPS standard 140-2 level 1.

For those administering the installation and configuration of the library, the *Crypto Officer Guide* provides critical information describing the necessary steps to ensure that the *validated* library is used.

For application developers seeking FIPS validation[1] of their application, use of the Crypto++ library can greatly simplify the evaluation process. By strictly following the procedures in the *User Guide*, developers can ensure that the application is using the library's cryptographic algorithms and security functions in a FIPS compliant manner.

## 1.2 Required Reading

This document refers to information and sections in the following documents:

- ***Crypto++ Security Policy* —** Intended for use by anyone planning to use or administer the Crypto++ library in a FIPS approved manner, this non-proprietary document explains how the library meets requirements of FIPS 140-2 level 1.
- ***Crypto++ API Reference*** — Intended for use by developers, this document defines the Crypto++ library object classes, methods, and parameters that must be instantiated in an application to implement cryptographic algorithms and security functions.

In addition to the Crypto++ documents listed above, readers can educate themselves on the requirements, spirit, and intent of FIPS 140-2. For details, refer to FIPS 140-2 *Security Requirements for Cryptographic Modules*.

## 1.3 FIPS Glossary

This section defines terms from the FIPS standard that are used in this guide.

| | |
|---|---|
| **Critical security parameter (CSP)** | Security-related information (e.g., secret and private cryptographic keys, and authentication data such as passwords and PINs) whose disclosure or modification can compromise the security of a cryptographic module. |
| **Cryptographic module** | The executable form of the loaded Crypto++ library. This includes the executing cryptographic algorithms and security functions and hardware devices like the CPU and registers that may store or operate using critical security parameters. The cryptographic module does not include the calling application |
| **FIPS-approved** | For this standard, a security function (e.g., cryptographic algorithm, |

[1] Other validating or certifying programs may also recognize FIPS validations. Check with the specific certification program before commencing your application development.

| | |
|---|---|
| **security function** | cryptographic key management technique, or authentication technique) that is either<br>▪ specified in an Approved standard,<br>▪ adopted in an Approved standard and specified either in an appendix of the Approved standard or in a document referenced by the Approved standard, or<br>▪ specified in the list of Approved security functions |
| **Approved mode of operation** | A mode of the cryptographic module that employs only Approved security functions (not to be confused with a specific mode of an Approved security function, e.g., DES CBC mode |
| **Crypto Officer** | An entity (e.g., person, service) that installs or configures the Crypto++ library |
| **User** | An entity (e.g., application, process) that accesses the services. |

# 2 Crypto Officer Guide

This section describes the responsibilities of the Crypto Officer to ensure that the *validated* Crypto++ library is used. The library can either be downloaded directly from the Crypto++ website or be installed as part of an application's installation package.

## 2.1 Downloaded Library

This section describes the steps to follow if the library is downloaded directly from the Crypto++ website.

### *2.1.1 Downloading*

The Crypto++ library can be downloaded from http://www.cryptopp.com. There are multiple forms and versions of the library available on the website so the Crypto Officer should ensure that the correct FIPS *validated* version of the library is downloaded. The DLL form of the library (*cryptopp.dll*) has undergone the FIPS validation process, while the source code and static library forms of the library have not. The *validated* Crypto++ DLL can be found in the distribution package **cryptopp504-DLL-msvc6.zip** for version 5.0.4 and **cryptopp523-DLL-msvc2003.zip** for version 5.2.3. Note that the *debug* versions of the DLL, included in the packages for development aid, are *not* validated.

### *2.1.2 Verifying Integrity*

After downloading the library, its integrity should be verified to ensure that it is intact and not tampered with over the wire. This step is not required for FIPS compliance, but is recommended by the vendor.

The library is digitally signed using the author's PGP private key. If the Crypto Officer's system does not have a PGP verification utility, you can obtain a free utility called GNU Privacy Guard from http://www.gnupg.org. The following steps show how to use GNU Privacy Guard Version 1.06 to verify the public key and integrity of the library:

1. Import the public key (provided with the library) into your keyring, as follows:

```
C:\cryptopp >gpg --import pubkey.asc
gpg: key 04549843: public key imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)

C:\cryptopp >
```

2. Verify the signature of the library file, as follows:

```
C:\cryptopp >..\gpg cryptopp504-DLL-msvc6.zip.sig
gpg: Signature made 05/09/02 12:29:03  using RSA key ID 04549843
gpg: Good signature from "Wei Dai (Crypto++ Code Signing Key)
<cryptopp@weidai.c
Could not find a valid trust path to the key.  Let's see whether we
can assign some missing owner trust values.

No path leading to one of our keys found.

gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
owner.
gpg: Fingerprint: F1F2 7D64 0CAA 3C65 763D  2508 F190 1AEB 0454 9843
```

   The first two lines of output indicate the signature status. Note that the above command produces a *warning* that the public key is not certified. This warning is expected, as other trusted parties do not sign the key. Verify the key in the next step.

3. Verify the public key by comparing the key fingerprint (in the last line of command output above) to the fingerprint specified in the *Crypto++ Security Policy* obtainable from the NIST web site http://csrc.nist.gov/cryptval/.

### 2.1.3 Installing and Protecting the Library

As described in the *Security Policy*, the library relies on certain operating system security features, such as password-based authentication. A crypto officer may impose additional authentication requirements such as smart card or biometric identification before allowing access to the library.

To ensure that the library remains intact and unmodified, the Crypto Officer should install it in a protected location (e.g., in a protected file system or secure configuration management system). It should be in a location that is read-accessible to only authorized users and write-accessible to only authorized crypto officers of the library.

### 2.1.4 Building and Running a Test Application

The Crypto Officer can confirm correct installation of the library by building and running a test application. For version 5.0.4 of the library, Microsoft Visual C++ 6.0 is required to build the application, and for version 5.2.3 of the library, Microsoft Visual C++ .NET 2003 is required. Refer to "APPENDIX A: *Test Application Guide*" for more information.

### 2.2 Application-Installed Library

If the library is installed as part of application's installation package, the Crypto Officer needs to follow the instructions described only in Section 2.1.3 "*Installing and Protecting the Library"*.

# 3 User Guide

This guide describes the responsibilities of and guidance for Users of the Crypto++ library to ensure that the library is used in a FIPS compliant manner. Developers can use this guide as a FIPS compliance check-list for application development using the *validated* Crypto++ library.

### 3.1 Approved Mode of Operation

The FIPS standards refer to an *Approved Mode of Operation*. A FIPS compliant application is allowed to execute non-Approved security functions. However, whenever the application does this, it is not running in an Approved Mode of Operation. In other words, an application is *not* operating in an Approved mode when it executes non-Approved security functions.

Note that there are some cryptographic algorithm types that do not have an Approved algorithm, but commercially available algorithms are *Allowed* to be used. For example, no FIPS standard for key establishment exists, but Diffie-Hellman key agreement and RSA encryption for key transport are *allowed* in Approved mode.

The *validated* Crypto++ DLL implements only those algorithms allowed to be used in Approved mode of operation. The Crypto++ Security Policy lists the security functions and their object classes as implemented in the library. To see the detailed interface descriptions for these services, look up the respective implementation object class in the *Crypto++ API Reference*.

If your application needs to use any non-Approved algorithm, you can use the other forms of the Crypto++ library (static library or source code) that have not been validated. However, to claim that your application includes a *validated* cryptographic module operating in an Approved mode, **your application must use the *validated* version of the library and call *at least one* Approved security function.**

Note that the *validated* Crypto++ DLL does not include either the DES or SHA-2 algorithm. Although DES is allowed in Approved mode of operation to support legacy systems, certain cipher modes of the algorithm are not allowed. For example, the (relatively new) CTR mode of DES is not allowed. For the SHA-2 algorithms, although they are included in a FIPS standard, their algorithm tests have not been defined at the time of this FIPS submission. Hence, neither of these two algorithms are included in the *validated* library.

### 3.2 Management of CSPs

The "*Cryptographic Key Management"* section of the Security Policy describes requirements related to management of CSPs that must be satisfied by applications. This section provides further additional guidance.

### 3.2.1 Identifying Critical Security Parameters

Each application needs to determine the CSPs that it generates, stores, uses, etc. For example, passwords, integrity checksums, secret and private keys are all considered different types of CSPs. These need to be generated, stored, and destroyed in an Approved manner.

### 3.2.2 Generation

Use available Crypto++ classes in the *validated* library for key generation since they have been validated as described in the *Security Policy*.

### 3.2.3 Storage

If a key needs to be stored in persistent media, follow FIPS requirements for protecting the CSP, as appropriate for the application.

### 3.2.4 Destruction

CSPs should be deleted and *wiped* from the system when they are no longer needed. This includes CSPs that are kept in memory, a registry, or a file. Per FIPS requirements, they should be overwritten such that they cannot be recovered.

The in-memory representations of keys and other CSPs in the library are automatically *wiped* since they are zeroized in the C++ destructors of their respective Crypto++ objects. However, applications should ensure that class destructors are in fact called for all CSPs. Destructors are always called (by the compiled code) for CSP objects allocated on the stack. However, **if a CSP is memory-allocated on the heap, the application should make sure to call its *delete* operator**, even if an exception occurs**.**

## 3.3 Self-Tests

As described in the "*Self-Tests*" section of the *Security Policy*, the Crypto++ library automatically runs power-up self-tests when the DLL is loaded. Additionally, although not FIPS required, application developers, if they choose, can initiate these tests on-demand by calling the *DoPowerUpSelfTest* function.

Applications can check the status of the self-tests by calling the *GetPowerUpSelfTestStatus* function. A return value of *POWER_UP_SELF_TEST_PASSED* indicates that all self-tests have succeeded. If any one of the self-tests has failed, the function returns the value *POWER_UP_SELF_TEST_FAILED*. In the failure case, any API calls to cryptographic functions will throw an error. As described in section 3.4 "*Error Handling*", applications should gracefully handle errors thrown by the library.

## 3.4 Error Handling

Crypto++ reports errors by throwing C++ exceptions. See the *Crypto++ API Reference* for a list of possible exceptions. Applications may handle errors by catching exceptions, reinitializing the Crypto++ object that threw the exception, correcting the condition that caused the error, and trying again.

The "*Finite State Model*" in the *Security Policy* describes various states, including error states, of the library.

## 3.5 Development Tips

This section provides general Crypto++ development-related guidance that is not necessarily relevant to FIPS compliance.

### *3.5.1 Heap Memory Management*

Because it's possible for the Crypto++ DLL to delete objects allocated by the calling application, they must use the same C++ memory heap. Three methods are provided to achieve this.

1. The calling application can tell Crypto++ what heap to use. This method is required when the calling application uses a non-standard heap.
2. Crypto++ can tell the calling application what heap to use. This method is required when the calling application uses a statically linked C++ Run Time Library. (Method 1 does not work in this case because the Crypto++ DLL is initialized before the calling application's heap is initialized.)
3. Crypto++ can use the heap provided by the calling application's dynamically linked C++ Run Time Library. The calling application must make sure that the dynamically linked C++ Run Time Library is initialized before Crypto++ is loaded.

When Crypto++ attaches to a new process, it searches all modules loaded into the process space for exported functions "GetNewAndDeleteForCryptoPP" and "SetNewAndDeleteFromCryptoPP". If one of these functions is found, Crypto++ uses methods 1 or 2, respectively, by calling the function. Otherwise, method 3 is used.

### *3.5.2 Block Ciphers*

Symmetric block ciphers (AES, DES, or TDES) cannot be used alone. They should be combined with a cipher mode such as cipher block chaining (CBC) mode in order to overcome known weaknesses of the block cipher's native mode, "electronic code book" (ECB) mode.

A block cipher can be combined with an appropriate cipher mode by constructing each object separately (e.g., AES and CBC) and then calling them one after the other. Alternatively, a templatized version of the cipher mode class, which takes in a block cipher object as a parameter, can be used to perform both operations automatically.

# APPENDIX A: Test Application Guide

The Crypto++ library is accompanied with a test application package that contains sample code demonstrating how to use the Crypto++ API on the *validated* Crypto++ library. The package also contains a set of build files to build the test application on the Windows development environment, which can be used to verify that the Crypto++ library is properly installed.

On startup, the library automatically runs FIPS-required self-tests that confirm correct operation of FIPS-approved cryptographic algorithms and security functions. The sample application demonstrates how to check for the status of these self-tests. It also contains other tests and calls to Crypto++ exported functions demonstrating use of various FIPS-approved cryptographic operations.

## A.1 Purpose of this Guide

This guide provides instructions for compiling the test application's source code to produce a ***DLLTest.exe*** application. The procedures in this guide assume that the Crypto++ DLL package has been downloaded, unzipped, and verified into a project directory as described in Section 2.1 *Downloaded Library*.

## A.2 Packaged Files

The test application package is delivered as a Microsoft Visual C++ project to be installed in a Microsoft Visual C++ Integrated Development Environment. It contains the following files:

- **DLLTest.dsp or DLLTest.vcproj —** Microsoft Visual C++ project file
- **DLLTest.cpp —** Test application source code

## A.3 Building the Application

This section describes the steps required to build the test application.

### *A.3.1 Opening the Test Application Project File*

1. **Important**: For version 5.0.4, ensure that Microsoft Visual Studio 6.0 has the following service packs installed:
   - Visual Studio 6.0 Service Pack 5
   - Processor Pack for Visual Studio 6.0 SP5.
2. Start the Microsoft Visual C++ program and open the **DLLTest.dsp** or **DLLTest.vcproj** project file.

### *A.3.2 Examining the sample code*

The test application exercises various cryptographic functionality and outputs the results to the console. It also simulates a failure of the power up self-test, catching the exception and re-running the self-test successfully before using any FIPS-approved cryptographic algorithms or security functions.

### *A.3.3 Running the Build Command*

1. Select the **DLLTest Win32 Debug** configuration using the **Build / Set Active Configuration** command.
2. Run the **Build DLLTest.exe** command in Microsoft Visual Studio to compile the source code to produce **DLLTest.exe**.
3. Make sure the cryptop.dll is in the same directory location as **DLLTest.exe**. If not, make a copy of cryptop.dll in that location.

## A.4 Running the Application

When you run the test application from a command prompt, an output of the power-up self-tests and conditional tests is presented. The test application also simulates some failures demonstrating how exceptions are caught and handled. Here's a sample output:

```
C:\cryptopp\DLLTest\CTDebug>DLLTest.exe

0. Automatic power-up self test passed.
1. Caught expected exception when simulating self test failure.
Exception message follows: Cryptographic algorithms are disabled
after a power-up self test failed.
2. Re-do power-up self test passed.
3. DES-CBC Encryption/decryption succeeded.
4. SHA-1 hash succeeded.
5. DSA key generation succeeded.
6. DSA key encode/decode succeeded.
7. DSA signature and verification succeeded.
8. DSA signature verification successfully detected bad
signature.
9. Caught expected exception when using invalid key length.
Exception message follows: DES: 5 is not a valid key length

FIPS 140-2 Sample Application completed normally.

C:\cryptopp\DLLTest\CTDebug>
```

The following describes the output in more detail:

- Line number 0 indicates that the automatic self-test passed.
- Line number 1 occurs because the test application substitutes an incorrect known value in place of a correct known value, causing a test to fail. This throws an exception. The test application outputs the message shown.
- Line number 2 indicates that the self-test was redone and passed.
- Lines numbers 3 through 8 indicate success of the sited Approved operations.
- Line number 9 indicates a key with an invalid length was used.

# APPENDIX B: Source Code Description

The Crypto++ library is a collection of open source cryptographic algorithms and security functions, many of which have been freely available in the public domain for some time. The DLL form of the library, which is FIPS *validated*, contains only algorithms that are either FIPS-approved or allowed to be used in a FIPS mode of operation. For more information, see the Crypto++ *Security Policy*.

For convenience, the library provides a C++ interface for all of the native interfaces (most of which were written in the C language), reducing the programming burden for developers writing applications in C++. The most recent version includes a filter interface that brings even more consistency to the programming level.

This document describes the design of the software and source code modules from which the DLL form of the library is generated. The source code modules are provided for inspection along with the DLL.

## B.1 Purpose of this Document

This document provides design information needed by evaluators certifying the Crypto++ DLL for FIPS 140-2 level 1. This document also fulfills the FIPS requirement for a Software Design Specification.

The "*Source Code Files"* section lists all of the files that make up the Crypto++ DLL. Short descriptions identify the purpose of files directly relevant to the evaluation.

The description in the "*Correspondence"* section helps evaluators map claims made in the *Security Policy* to their concrete implementations in the source code.

## B.2 Source Code Files

This section lists all of the files that make up the Crypto++ DLL. Short descriptions identify the purpose of files that are directly relevant to the evaluation. Some general purpose files support operations in many files. For example, *algebra.cpp* performs various math and algebra operations. Such files do not have specialized descriptions.

| | |
|---|---|
| aes.h | |
| algebra.h, algebra.cpp | |
| algparam.h, algparam.cpp | |
| argnames.h | |
| asn.h, asn.cpp | |
| basecode.h, basecode.cpp | |
| cbcmac.h, cbcmac.cpp | CBC-MAC/TDES |
| channels.h, channels.cpp | |
| config.h | Compile-time configuration options |
| cryptlib.h, cryptlib.cpp | Abstract base classes and other interface definitions |
| des.h, des.cpp | Triple-DES |
| dessp.cpp | |
| dh.h, dh.cpp | Diffie-Hellman |

| | |
|---|---|
| dll.h, dll.cpp | |
| dsa.h, dsa.cpp | DSA |
| ec2n.h, ec2n.cpp | Elliptic Curve over GF(2^n) |
| eccrypto.h, eccrypto.cpp | Elliptic Curve Cryptographic, including ECDSA |
| ecp.h, ecp.cpp | Elliptic Curve over GF(p) for prime p |
| eprecomp.h, eprecomp.cpp | |
| files.h, files.cpp | |
| filters.h, filters.cpp | Implementation of various filter classes |
| fips140.h, fips140.cpp | FIPS-140 related classes and functions, excluding the power-on self tests. |
| fipstest.cpp | Power-on self tests required by FIPS-140 |
| fltrimpl.h | |
| gf2n.h, gf2n.cpp | |
| gfpcrypt.h , gfpcrypt.cpp | Implementation of schemes based on DL over GF(p), including DSA. |
| hex.h, hex.cpp | |
| hmac.h, hmac.cpp | HMAC/SHA-1 |
| integer.h, integer.cpp | |
| iterhash.h, iterhash.cpp | |
| mdc.h | |
| misc.h, misc.cpp | |
| modarith.h | |
| modes.h, modes.cpp | Block cipher modes of operation |
| modexppc.h, modexppc.cpp | |
| mqueue.h, mqueue.cpp | |
| mqv.h | |
| nbtheory.h, nbtheory.cpp | |
| oaep.h, oaep.cpp | |
| oids.h | |
| osrng.h, osrng.cpp | OS provided/seeded RNGs, including AutoSeededX917RNG |
| pch.h, pch.cpp | |
| pkcspad.h, pkcspad.cpp | |
| pubkey.h, pubkey.cpp | Classes for implementing public key schemes |
| queue.h, queue.cpp | |
| randpool.h, randpool.cpp | |
| rdtables.cpp | |
| rijndael.h, rijndael.cpp | Rijndael (AES) |
| rng.h, rng.cpp | |
| rsa.h, rsa.cpp | RSA |
| secblock.h | |
| seckey.h | |

| | |
|---|---|
| sha.h, sha.cpp | SHA-1 |
| simple.h, simple.cpp | |
| skipjack.h, skipjack.cpp | SkipJack |
| smartptr.h | |
| stdcpp.h | |
| strciphr.h, strciphr.cpp | |
| tea.cpp | |
| trdlocal.h, trdlocal.cpp | |
| words.h | |
| cryptopp.rc | |

## B.3 Correspondence

This section specifies the correspondence between the design of the software components of the cryptographic module and (a) its security policy and (b) its services.

### *B.3.1 Correspondence to Security Policy*

The cryptographic module design collects publicly available cryptographic algorithms and security functions into a C++ object library. For evaluation purposes, FIPS approved and allowed algorithms are separately compiled into a DLL form, which can be used by a calling application.

The DLL does not itself contain any cryptographic keys or critical security parameters (CSPs). A calling application may use services within the cryptographic module to operate on cryptographic keys and CSPs. Thus the application is responsible for key and CSP management.

The cryptographic module specifies 2 separate roles (a Crypto officer and a User). Access control (to available services) and separation of roles is specified in the Security Policy.

### *B.3.2 Correspondence to Services*

Software modules within the DLL implement various services that are FIPS-approved or allowed. This section describes the main modules implementing these services. Use these modules as starting points for examining all of the modules related to the specific service. For example, the main module for DES is *des.cpp*. The evaluator can examine this module and all included header and sources files to determine whether the complete set properly implements the given service.

| Service Type | Algorithm | Main module |
|---|---|---|
| Symmetric Cipher | AES | *rijndael.cpp* and *modes.cpp* |
| | Triple DES (2-key) | *des.cpp* and *modes.cpp* |
| | Triple DES (3-key) | *des.cpp* and *modes.cpp* |
| | Skipjack | *skipjack.cpp* |
| Digital Signature and Key Generation | RSA Signature | *rsa.cpp* |
| | DSA | *dsa.cpp* |
| | ECDSA | *eccrypto.cpp* |

| | | |
|---|---|---|
| Message Digest | SHA-1 | *sha.cpp* |
| | CBC-MAC/TDES | *cbcmac.cpp* |
| | HMAC/SHA-1 | *hmac.cpp* |
| Random Number Generator | ANSI X9.31-1998 - Appendix A | *osrng.cpp* |
| Key Transport | Diffie-Hellman Key Agreement | *dh.cpp* |
| | RSA Encryption | *rsa.cpp* |
| Other Functions | On-demand Self-Test | *fipstest.cpp* |
| | Self-Test Status | *fips140.cpp* |