wolfCrypt

Version 4.5.4

FIPS User Guide

Document Version 1.8

6 April, 2021

wolfSSL Inc.
10016 Edmonds Way
Suite C-300
Edmonds, WA 98020
https://www.wolfssl.com/
mailto:support@wolfSSL.com
+1 425.245.8247

Table of Contents

List of Tables

1 Overview

The wolfCrypt v4.0 FIPS User Guide addresses the FIPS 140-2 documentation requirements for:

- Maintaining security when distributing the module
- Secure installation of the module
- Secure initialization of the module

2 Secure Distribution and Installation Instructions

2.1 Linux (Traditional) Installation

NOTE: For HP Imaging & Printing Linux 4.9 see section 2.4
Operation of wolfCrypt in the FIPS-140-2 Approved mode requires the wolfCrypt FIPS library version 4.0 or later. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.0.0-commercial-fips-linux.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.0.0-commercial-fips-linux.7z
7-Zip...
Extracting archive: wolfssl-4.0.0-commercial-fips-linux.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS:

```
$ ./configure --enable-fips=v2
```

Or to allow use of AES-NI and RDSEED

```
$ ./configure --enable-fips=v2 --enable-intelasm
$ ./wolfcrypt/test/testwolfcrypt
$ sudo make install
```

If you have not received the library with FIPS support the *./configure* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the FIPS enable option. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

make check will verify the build and that the library is operating correctly. If *make check* fails this probably means the In Core Integrity check has failed, which is expected. To verify this do:

```
$ ./wolfcrypt/test/testwolfcrypt

...
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
```

```
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

Copy the value given for the hash in the output, and replace the value of verifyCore[] in *./wolfcrypt/src/fips_test.c* with this new value. After updating verifyCore[], recompile the wolfSSL library by running *make check* again.

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.2 Windows 10 Installation

wolfCrypt in FIPS mode for Windows 10 requires the wolfCrypt FIPS library version 4.0 or later. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.

To verify the fingerprint of the package, calculate the SHA-256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
% shasum –a 256 wolfssl-4.0.0-commercial-fips-windows.7z
02da35d0a4d6b8e777236fe30da7a6ff93834fb16939ea16da663773f1b34cf0
```

And compare the sum to the sum provided with the package. If for some reason the sums do not match stop using the Module and contact wolfSSL.

A GUI-based 7-zip extraction may be used. To unpack the bundle from a command shell:

```
% 7za x wolfssl-4.0.0-commercial-fips-windows.7z
7-Zip...
Extracting archive: wolfssl-4.0.0-commercial-fips-windows.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt for FIPS 140-2 compliance:

1. In Visual Studio open IDE\WIN10\wolfssl-fips.sln.
2. Select the Release DLL and x64 as the build type and target.
3. Build the solution
4. The library should be in the directory IDE\WIN10\DLL Release\x64 as a pair of files: wolfssl-fips.lib is the linking library and wolfssl-fips.dll is the shared library proper, it can be added to your project.
5. In your application project, add a preprocessor macro for HAVE_FIPS. This will ensure the compiler is loading all the correct settings from the user_settings.h header file in the library.
6. Build the solution.
7. Run the code from the DLL Release\x64 directory, the default FIPS check failure should be output in the shell.

The enable and disable options required for approved FIPS mode are automatically set in the user settings file mentioned in step 5 above. Any encryption algorithms that are not enabled by the FIPS mode **must** not be enabled separately. Options affecting wolfSSL usage are still allowed.

The first run should indicate the In Core Integrity check has failed:

```
in my Fips callback, ok = 0, err = -203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
```

```
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.3 STM32L4 Installation

wolfCrypt in FIPS mode for STM32L4 requires the wolfCrypt FIPS library version 4.0.1 or later. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.

To verify the fingerprint of the package, calculate the SHA-256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
% shasum -a 256 wolfssl-4.0.1-commercial-fips-stm32l4.7z
02da35d0a4d6b8e777236fe30da7a6ff93834fb16939ea16da663773f1b34cf0
```

And compare the sum to the sum provided with the package. If for some reason the sums do not match stop using the Module and contact wolfSSL.

A GUI-based 7-zip extraction may be used. To unpack the bundle from a command shell:

```
% 7za x wolfssl-4.0.1-commercial-fips-stm32l4.7z
7-Zip...
Extracting archive: wolfssl-4.0.1-commercial-fips-stm32l4.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt for FIPS 140-2 compliance:

1. Upon receiving wolfssl-4.0.1-commercial-fips-stm32l4.7z drop it into the APPLICATION_ROOT directory and extract it. The Firmware directory structure is expected to be as below after extraction:
   FIRMWARE_ROOT/directoryA/APPLICATON_ROOT/wolfssl-4.0.1-commercial-fips-stm32l4/
2. Change directory into wolfssl-4.0.1-commercial-fips-stm32l4 and use this command to execute the script that will flatten the wolfCrypt FIPS boundary resources into the APPLICATION_ROOT:
   ./scripts/stm32l4-v4_0_1_build.sh
3. The stm32lv-v4_0_1_build.sh script will flatten both wolfCrypt FIPS sources and non-FIPS boundary related resources into the APPLICATION_ROOT which the APPLICATION_ROOT/Makefile will need to build. If this is a followup release from a previous install the script will remove old source files regarless of FIPS or non-FIPS sources prior to flattening. This is to ensure that updates made to non-FIPS source files are carried over however the FIPS source files will never change once validated. Once the sources have been flattened this script will also run a make clean, make, and make install-target to flash the device with the newly built firmware. One additional note, this script is pre-configured with settings for the WOLCRYPT library that were used during testing, for non-FIPS related settings changes apply them to the script as user_setting.h is re-written each time the script is executed.
4. Table 1 below details the source files that will be flattened and the order they should be added to the Makefile in along with Makefile rules for include paths

Description
wolfCrypt Makefile Rules for STM32L4

```
WOLF_ROOT := $(ROOT_DIR)/directoryA/Application/wolfssl-4.0.1-commercial-fips-stm32l4/
WOLF_CRYPT_ROOT := $(WOLF_ROOT)/wolfcrypt/src

INC_DIRS += $(WOLF_ROOT)
INC_DIRS += $(WOLF_CRYPT_ROOT)

SRCS :=
SRCS += <application file(s)>.c
# start of FIPS boundary
SRCS += wolfcrypt_first.c
SRCS += hmac.c
SRCS += random.c
SRCS += sha256.c
SRCS += rsa.c
SRCS += ecc.c
SRCS += aes.c
SRCS += des3.c
SRCS += sha.c
SRCS += sha512.c
SRCS += sha3.c
SRCS += dh.c
SRCS += cmac.c
SRCS += fips.c
SRCS += fips_test.c
SRCS += wolfcrypt_last.c
#end of FIPS boundary
# non FIPS boundary wolfCrypt Sources: Start
SRCS += asn.c
SRCS += coding.c
SRCS += dsa.c
SRCS += error.c
SRCS += hash.c
SRCS += logging.c
SRCS += md5.c
SRCS += memory.c
SRCS += signature.c
SRCS += tfm.c
SRCS += wc_encrypt.c
SRCS += wc_port.c
SRCS += wolfmath.c
# non FIPS boundary wolfCrypt Sources: End
#test app (Uncomment if you wish to build and run the wolfCrypt test application)
#SRCS += test.c
#test app
```

Table 1: wolfCrypt Sources Makefile Rules for STM32L4

5. A change was made to fips.c for v4.0.1 to allow for configuration of two external peripherals on the STM32L4 device during the Power Up Self Test and priory to fipsEntry. The function documented below must be present in the firmware main.c file for invocation by the FIPS power on self test. The wolfSSL_POS_SystemClock_Config function was specifically designed to be minimalistic in that unlike the default SystemClock_Config function it will only turn on the absolute bare minimum to configure the real time clock (RTC).

Description
wolfSSL peripheral configuration API's on STM32L4
wolfSSL_POS_SystemClock_Config

```
/* wolfSSL added for FIPS testing to turn clock speed up to 120MHz for crypto
 * tests taking far too long */
void wolfSSL_POS_SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
```

```
  /**Configure the main internal regulator output voltage
  */
  if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1_BOOST) != HAL_OK)
  {
    Error_Handler();
  }
  /**Initializes the CPU, AHB and APB busses clocks, nothing else, we're in
   * Power Up Self tests so we only want the bare minimum for clock at this
   * point, no other peripherals.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLM = 6;
  RCC_OscInitStruct.PLL.PLLN = 90;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
  RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /**Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
  {
    Error_Handler();
  }

  /* The only other peripheral we need at this point is the RNG */
  MX_RNG_Init();
}
```

MX_RNG_Init

```
  /* wolfSSL RNG Addition */
  void MX_RNG_Init(void)
  {

    /* Peripheral clock enable */
    LL_AHB2_GRP1_EnableClock(LL_AHB2_GRP1_PERIPH_RNG);

    LL_RCC_PLLSAI1_ConfigDomain_48M(LL_RCC_PLLSOURCE_MSI,       LL_RCC_PLLM_DIV_1,       24,
  LL_RCC_PLLSAI1Q_DIV_2);
    LL_RCC_PLLSAI1_Enable();
    LL_RCC_PLLSAI1_EnableDomain_48M();
    while(LL_RCC_PLLSAI1_IsReady() != 1) {};
    LL_RCC_SetRNGClockSource(LL_RCC_RNG_CLKSOURCE_PLLSAI1);

    LL_RNG_Enable(RNG);
    LL_RNG_EnableClkErrorDetect(RNG);
    LL_RNG_Disable(RNG);

  }
  /* wolfSSL RNG Addition END */
```

Table 2: wolfSSL minimal peripheral init API's for STM32L4

6. Any other minor adjustments to the Firmware default files was documented with a comment "wolfSSL Addition", these changes were of the nature, including an additional header or peripheral source code. Please contact wolfSSL if you have any questions about these changes. The headers added were already present in the firmware bundle but were not yet being included. The headers were stm32l4xx_ll_rng.h and stm32l4xx_ll_rtc.h, the source additions were restricted to changes in the firmware main.c.

The enable and disable options required for approved FIPS mode are automatically set in the user settings file mentioned in step 3 above. Any encryption algorithms that are not enabled by the FIPS mode **must** not be enabled separately. Options affecting wolfSSL usage are still allowed.

The first run should indicate the In Core Integrity check has failed:

```
in my Fips callback, ok = 0, err = -203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.4 HP Imaging & Printing Linux 4.9 Installation

Operation of wolfCrypt in the FIPS-140-2 Approved mode on Linux 4.9 requires the wolfCrypt FIPS library version 4.1.0 or later. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.1.0-commercial-fips-linux-HP-IPL-4.9.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.1.0-commercial-fips-linux-HP-IPL-4.9.7z
7-Zip...
Extracting archive: wolfssl-4.1.0-commercial-fips-linux.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS when building with PAA the configure option --enable-armasm is required and without PAA requires --disable-armasm or simply remove --enable-armasm (it is not on by default)

wPAA:

```
$ ./configure --enable-fips=v2 --enable-armasm
$ make
$ ./wolfcrypt/test/testwolfcrypt
$ sudo make install
```

woPAA:

```
$ ./configure --enable-fips=v2 --disable-armasm
Or
$ ./configure --enable-fips=v2
$ make
$ ./wolfcrypt/test/testwolfcrypt
$ sudo make install
```

If you have not received the library with FIPS support the *./configure* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the FIPS enable option. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

make check will verify the build and that the library is operating correctly. If *make check* fails this probably means the In Core Integrity check has failed, which is expected. To verify this do:

```
$ ./wolfcrypt/test/testwolfcrypt

...
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

Copy the value given for the hash in the output, and replace the value of verifyCore[] in *./wolfcrypt/src/fips_test.c* with this new value. After updating verifyCore[], recompile the wolfSSL library by running *make check* again.

To simplify the build process and make it easier on HP developer wolfSSL used put together an automated method for the flatboard that was tested during the validation process. wolfSSL provided two scripts in the first release wolfssl-4.2.0-commercial-fips-HPIPL.7z named “build-wPAA.sh” and “build-woPAA.sh” these scripts use the simple test application supplied by HP (aes_encrypt_decrypt.c) to perform the following test in an automated fashion:

1) Configure and build libwolfssl with the HP settings provided for FIPS validation testing, copy library to device specified in the network by $DEVICE_IP variable in the script (Update $DEVICE_IP for each device).
2) Build and link aes_encrypt_decrypt application and copy to device specified by $DEVICE_IP
3) Remote into device and execute “./aes_encrypt_decrypt”, check the output for a new hash and if found store to variable $NEW_HASH in the script.
4) If $NEW_HASH is a non-empty value then updated wolfcrypt/src/fips_test.c variable “verifyCore” with the new hash value, re-build libwolfssl and re-copy/paste libwolfssl to the device again.
    a. If 4 executed ($NEW_HASH was non-empty value) rebuild all apps the script supports (wolfCrypt Test, wolfCrypt Benchmark, and aes_encrypt_decrypt) and copy the updated apps to the device specified by $DEVICE_IP

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.5 Linux socfpgacyclone5 5.3

Operation of wolfCrypt in the FIPS-140-2 Approved mode on Linux socfpgacyclone5 5.3 requires the wolfCrypt FIPS library version 4.3.0 or later. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.3.0-commercial-fips-LSFPGA5-v2.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.3.0-commercial-fips-LSFPGA5-v2.7z
7-Zip...
Extracting archive: wolfssl-4.3.0-commercial-fips-LSFGA5-v2.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS you may contact wolfSSL support for access to an automated build script as a starting point or configure in your own setup with the settings below:

```
$ ./configure -CFLAGS="-DSIZEOF_LONG_LONG=8 -DWOLFSSL_PUBLIC_MP" --
prefix=<INSTALL_LOCATION> --disable-examples --enable-shared --disable-
static --disable-examples --disable-crypttests --enable-sha3 --enable-
sha512 --enable-sha384 --enable-fips=v2 --enable-keygen
$ make
$ make install
```

If you have not received the library with FIPS support the *./configure* step will fail. Please contact wolfSSL.

Once installed cross-compile your application and link it against the cross-compiled library then copy the test app and library over to the target device for testing.

The enable and disable options required for approved FIPS mode are automatically set by the FIPS enable option. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

You should setup a FIPS callback in your application to output the run-time hash during development in case the integrity check fails. The FIPS callback is below and can be copy/pasted into your app then registed on startup with:

```
wolfCrypt_SetCb_fips(myFipsCb);
```

FIPS CALLBACK:

```
static void myFipsCb(int ok, int err, const char* hash)
{
```

```
        printf("in my Fips callback, ok = %d, err = %d\n", ok, err);
        printf("message = %s\n", wc_GetErrorString(err));
        printf("hash = %s\n", hash);

        if (err == IN_CORE_FIPS_E) {
            printf("In core integrity hash check failure, copy above hash\n");
            printf("into verifyCore[] in fips_test.c and rebuild\n");
        }
    }
```

Now when the app runs if the run-time integrity check fails you should see this output:

```
$ ./run-your-app
...
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

Copy the value given for the hash in the output, and replace the value of verifyCore[] in *./wolfcrypt/src/fips_test.c* with this new value. After updating verifyCore[], recompile the wolfSSL library by running *make check* again.

To simplify the build process, as mentioned above, contact wolfSSL support if you would like an automated script to simplify the build process during development.

2.6 FDLARMv7

Operation of wolfCrypt in the FIPS-140-2 Approved mode on Linux 4.12 Yocto Standard requires the wolfCrypt FIPS library version 4.4.1. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.4.1-commercial-fips-linux-FDLARMv7-v2.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.4.1-commercial-fips-linux-FDLARMv7-v2.7z
7-Zip...
Extracting archive: wolfssl-4.1.0-commercial-fips-linux.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS two automation scripts are provided in the releases for this module version. The scripts will be located in "wolfssl-root-directory/v4.4.1-Automation" directory and are named "build-with-PAA.sh" and "build-with-out-PAA.sh" respectively.

wPAA:

```
$ ./v4.4.1-Automation/build-with-PAA.sh
```

woPAA:

```
$ ./v4.4.1-Automation/build-with-out-PAA.sh
```

If you have not received the library with FIPS support the *./configure* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the FIPS enable option. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

make check will verify the build and that the library is operating correctly. If *make check* fails this probably means the In Core Integrity check has failed, which is expected. To verify this do:

```
$ ./wolfcrypt/test/testwolfcrypt

...
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

Copy the value given for the hash in the output, and replace the value of verifyCore[] in *./wolfcrypt/src/fips_test.c* with this new value. After updating verifyCore[], recompile the wolfSSL library by running *make check* again.

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.7 VF500

Operation of wolfCrypt in the FIPS-140-2 Approved mode on Nucleus 3.0 version 2013.08.1 requires the wolfCrypt FIPS library version 4.4.2. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.5.0-commercial-fips-linux-VF500-v2.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.5.0-commercial-fips-linux-VF500-v2.7z
7-Zip...
Extracting archive: wolfssl-4.5.0-commercial-fips-linux-VF500-v2.7z
...
```

```
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS both a “user_settings.h” header and a “Makefile” are provided in the releases for this module version. The user settings header and Makefile will be located in “wolfssl-root-directory/v4.4.2-SP-and-user-guide” directory. First remove past versions of wolfSSL from the L3-Harris directory:

```
$ rm -rf L3-Harris/common_gpp/LMR/third_party/wolfSSL
```

Next unzip the new release and re-name it to “wolfSSL”

```
$ 7za x -P<PASSWORD> wolfssl-4.5.0-commercial-fips-linux-VF500-v2.7z
$ mv wolfssl-4.5.0-commercial-fips-linux-VF500-v2 L3-
Harris/common_gpp/LMR/third_party/wolfSSL
```

Now copy/paste the user_settings.h and Makefile to the root directory and you are ready to build:

```
$ cp L3-Harris/common_gpp/LMR/third_party/wolfSSL/v4.4.2-SP-and-user-
guide/user_settings.h L3-
Harris/common_gpp/LMR/third_party/wolfSSL/user_settings.h
$ cp L3-Harris/common_gpp/LMR/third_party/wolfSSL/v4.4.2-SP-and-user-
guide/Makefile L3-Harris/common_gpp/LMR/third_party/wolfSSL/Makefile
$ cd L3-Harris/common_gpp/LMR
$ make daytona
```

If you have not received the library with FIPS support the *make daytona* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the user_settings.h configuration. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

Any call into the crypto module can be used to verify the module is working. If a call into the module produces output like below be sure to copy the new hash, paste it into third_party/wolfSSL/wolfcrypt/src/fips_test.c in the “verifyCore[]” array, re-compile, and run the test again.

```
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.8 CT8200

Operation of wolfCrypt in the FIPS-140-2 Approved mode on CodeOS v1.4 requires the wolfCrypt FIPS library version 4.5.2. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.

To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.5.0-commercial-fips-CT8200-v2.7z
746341ac6d88b0d6de02277af5b86275361ed106c9ec07559aa57508e218b3f5
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.5.0-commercial-fips-CT8200-v2.7z
7-Zip...
Extracting archive: wolfssl-4.5.0-commercial-fips-CT8200-v2.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS both a “user_settings.h” header and a “Makefile” are provided in the releases for this module version. The user settings header and Makefile will be located in “wolfssl-root-directory/v4.5.2-SP-and-user-guide” directory. First remove past versions of wolfSSL from the library/libwolfssl/ directory

```
$ rm -rf library/libwolfssl/wolfssl-*
```

Next unzip the new release and move it to the library/libwolfssl/ directory:

```
$ 7za x -P<PASSWORD> wolfssl-4.5.0-commercial-fips-CT8200-v2.7z
$ mv wolfssl-4.5.0-commercial-fips-CT8200-v2 ~/cr2700-codeOS-
1.4/library/libwolfssl/wolfssl-4.5.0-commercial-fips-CT8200-v2
```

Now copy/paste the user_settings.h and Makefile to the root directory and you are ready to build:

```
$ cd ~/cr2700-codeOS-1.4/library/libwolfssl/wolfssl-4.5.0-commercial-
fips-CT8200-v2/
$ cp v4.5.2-SP-and-user-guide/user_settings.h ../user_settings.h
$ cp v4.5.2-SP-and-user-guide/Makefile ../Makefile
```

Before we can build however we must update the Config.inc with the new release version. Go ahead and open ~/cr2700-codeOS-1.4/Config.inc and change the variable WOLF_RELEASE_VERSION to be “WOLF_RELEASE_VERSION=wolfssl-4.5.0/commercial-fips-CT8200-v2

```
$ cd ~/cr2700-codeOS-1.4
$ make bootloader
```

If you have not received the library with FIPS support the *make bootloader* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the user_settings.h configuration. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

Any call into the crypto module can be used to verify the module is working. If a call into the module produces output like below be sure to copy the new hash, paste it into

third_party/wolfSSL/wolfcrypt/src/fips_test.c in the “verifyCore[]” array, re-compile, and run the test again.

```
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

2.9 ARMv8 [A | B]

Operation of wolfCrypt in the FIPS-140-2 Approved mode on Linux 4.14 or Linux 4.19 requires the wolfCrypt FIPS library version 4.5.4. The wolfCrypt FIPS releases can be obtained with a link provided by wolfSSL through direct email.
To verify the fingerprint of the package, calculate the SHA256 sum using a FIPS 140-2 validated cryptographic module. The following command serves as an example:

```
$ shasum -a 256 wolfssl-4.4.1-commercial-fips-ARMv8-A-v2.7z
9109651b2e3f967d968f7e7ce7ad31a75e7b9a8d5af373f5302edb05e0412b62

$ shasum -a 256 wolfssl-4.4.1-commercial-fips-ARMv8-B-v2.7z
9109651b2e3f967d968f7e7ce7ad31a75e7b9a8d5af373f5302edb05e0412b62
```

Compare the sum to the sum provided with the package. If the sums do not match stop using the Module and contact wolfSSL.

To unpack the bundle:

```
$ 7za x wolfssl-4.4.1-commercial-fips-ARMv8-B-v2.7z
7-Zip...
Extracting archive: wolfssl-4.4.1-commercial-fips-ARMv8-B-v2.7z
...
Enter password (will not be echoed):
```

When prompted, enter the password. The password is provided in the distribution email.

To build and install wolfCrypt with FIPS two automation scripts are provided in the releases for this module version. The scripts will be located in “wolfssl-root-directory/v4.5.4-Automation” directory and are named “build-wPAA.sh” and “build-woPAA.sh” respectively.

wPAA:

```
$ ./v4.5.4-Automation/build-wPAA.sh
```

woPAA:

```
$ ./v4.5.4-Automation/build-woPAA.sh
```

If you have not received the library with FIPS support the *./configure* step will fail. Please contact wolfSSL.

The enable and disable options required for approved FIPS mode are automatically set by the FIPS enable option. Any encryption algorithms that are not enabled by the FIPS mode must not be enabled separately. Options affecting wolfSSL usage are still allowed.

make check will verify the build and that the library is operating correctly. If *make check* fails this probably means the In Core Integrity check has failed, which is expected. To verify this do:

```
$ ./wolfcrypt/test/testwolfcrypt

...
in my Fips callback, ok = 0, err = 203 message = In Core Integrity
check FIPS error
hash = 622B4F8714276FF8845DD49DD3AA27FF68A8226C50D5651D320D914A5660B3F5
In core integrity hash check failure, copy above hash into verifyCore[]
in fips_test.c and rebuild
```

Copy the value given for the hash in the output, and replace the value of verifyCore[] in *./wolfcrypt/src/fips_test.c* with this new value. After updating verifyCore[], recompile the wolfSSL library by running *make check* again.

The In Core Integrity checksum will vary with compiler versions, runtime library versions, target hardware, and build type.

Also included in the ARMv8[A | B] bundles will be the intial performance results observed between wPAA and woPAA. These will be located in the v4.5.4-SP-and-user-guide/ performance-wPAA-vs-woPAA/ directory.

3 Secure Installation

3.1 Linux (Traditional)

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.2 Windows 10

The library uses the DllMain() function to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.3 STM32L4

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner on each reset without power loss or full power cycle.

3.4 HP Imaging & Printing Linux 4.9

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.5 Linux socfpgacyclone5 5.3

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.6 VF500

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.7 CT8200

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

3.8 ARMv8 [A | B]

The library uses an allocator method to initialize itself after loading, without programmer interaction. The library will perform its own self-test in a thread safe manner.

4 wolfCrypt FIPS 140-2 Services – API entry points and Usage

This section provides usage information for wolfCrypt 4.0 library FIPS services. See Chapter 10: wolfCrypt Usage Reference in the wolfSSL Manual for additional information on the cryptographic services listed in this section.

wolfCrypt removes the prefix `wc_` and adds the suffix `_fips` to all FIPS mode APIs. For example, `wc_ShaUpdate()` becomes `ShaUpdate_fips()`. The FIPS mode functions can be called directly, but they can also be used through macros.

HAVE_FIPS is defined when using wolfCrypt in FIPS mode and that creates a macro for each function with FIPS support. For the above example, a user with an application calling `wc_ShaUpdate()` can recompile with the FIPS module and automatically get `ShaUpdate_fips()` support without changing their source code. Of course, recompilation is necessary with the correct macros defined.

A new error return code **FIPS_NOT_ALLOWED_E** may be returned from any of these functions used directly or even indirectly.

The error is returned when the Power-On Self-Tests (POST) are not yet complete or they have failed. POST is done automatically as a default entry point when using the library, no user interaction is required to start the tests. To see the current status including any error code at any time call `wolfCrypt_GetStatus_fips()`. For example, if the AES Known Answer Test failed during POS GetStatus may return **AES_KAT_FIPS_E**.

4.1 Authenticated symmetric encrypt/decrypt

The wolfCrypt AES-GCM API allows for both internally and externally generated IVs. When calling the function `AesGcmSetExtIV_fips()`, a 96- or 128-byte IV may be provided. When calling the function `AesGcmSetIntIV_fips()`, a 4-byte fixed value, desired IV size, and `WC_RNG` are passed in; the IV is a combination of the 4-byte value and enough random bytes to fill out the IV length to the size. In either case, calls to `AesGcmEncrypt_fips()` require a pointer to where the IV to be shared with the peer will go so the value can be decrypted. The IV is incremented after every call to `AesGcmEncrypt_fips()`. It shall only be incremented 2^{32}-1 times, except in the case of the 96-byte IV which may be incremented 2^{64}-1 times, at which point an error code is returned.

4.2 Digital Signature

DSA is not supported. Note that the DSA row in CAVP algorithm testing is for FFC key generation, for keys used for Diffie-Hellman as specified by [56A]. FFC key generation is CAVP validated under the auspices of [186].

Use `FreeRSAKey_fips()`, `ecc_free_fips()`, and `FreeDhKey_fips()` for deallocation to free key memory to sanitize allocated keys.

4.3 HMAC

Usage instructions: https://www.wolfssl.com/docs/wolfssl-manual/ch10/ Section 10.2.1

4.4 Message digest

Usage instructions: https://www.wolfssl.com/docs/wolfssl-manual/ch10/ Section 10.1.3.

4.5 Self-Test

This service corresponds to the loading of the application that includes the module. Loading the module results in default entry point invocation of self-test, requiring no operator intervention.

5 Map of API to FIPS 140 module services

API Call	Description
Digital Signature Service	
InitRsaKey_fips	Initializes RSA key object for use with optional heap hint p. Returns 0 on success, < 0 on error.
FreeRsaKey_fips	Releases RSA key resources. Returns 0 on success, < 0 on error.
RsaSSL_Sign_fips	Performs RSA key Signing operation on input in of size inLen, outputting to out of size outLen using rng. Returns 0 on success, < 0 on error.
RsaSSL_VerifyInline_fips	Performs RSA key Verification without allocating temporary memory on input in of size inLen, writes to output out. Returns 0 on success, < 0 on error.
RsaSSL_Verify_fips	Performs RSA key Verification on input in of size inLen, writes to output out of size outLen. Returns 0 on success, < 0 on error.
RsaPSS_Sign_fips	Performs RSA key Signing operation with PSS padding on input in of size inLen, outputting to out of size outLen using rng. It uses the hash algorithm hash with the mask generation function mgf. Returns 0 on success, < 0 on error.
RsaPSS_SignEx_fips	Performs RSA key Signing operation with PSS padding on input in of size inLen, outputting to out of size outLen using rng. It uses the hash algorithm hash with the mask generation function mgf and a salt length of saltLen. Returns 0 on success, < 0 on error.
RsaPSS_VerifyInline_fips	Performs RSA key Verification without allocating temporary memory on input in of size inLen, writes to output out. It uses the hash algorithm hash with the mask generation function mgf. Returns 0 on success, < 0 on error.
RsaPSS_VerifyInlineEx_fips	Performs RSA key Verification on input in of size inLen, writes to output out of size outLen. It uses the hash algorithm hash with the mask generation function mgf and a salt length of saltLen. Returns 0 on success, < 0 on error.

RsaPSS_Verify_fips	Performs RSA key Verification on input in of size inLen, writes to output out of size outLen. It uses the hash algorithm hash with the mask generation function mgf. Returns 0 on success, < 0 on error.
RsaPSS_VerifyEx_fips	Performs RSA key Verification on input in of size inLen, writes to output out of size outLen. It uses the hash algorithm hash with the mask generation function mgf and a salt length of saltLen. Returns 0 on success, < 0 on error.
RsaPSS_CheckPadding_fips	Checks the padding after RSA key verification on input in of size inSz with signature sig of size sigSz using hash hashType. Returns 0 on success, < 0 on error.
RsaPSS_CheckPaddingEx_fips	Checks the padding after RSA key verification on input in of size inSz with signature sig of size sigSz using hash hashType and a salt length of saltLen. Returns 0 on success, < 0 on error.
RsaEncryptSize_fips	Retrieves RSA key Output Size. Returns key output size > 0 on success, < 0 on error.
wc_RsaPrivateKeyDecode	Decodes an Rsa Private key from a buffer input starting at index inOutIdx of size inSz. Returns 0 on success, < 0 on error.
wc_RsaPublicKeyDecode	Decodes an Rsa Public key from a buffer input starting at index inOutIdx of size inSz. Returns 0 on success, < 0 on error.
ecc_init_fips	Initializes ECC key object for use. Returns 0 on success, < 0 on error.
ecc_free_fips	Releases ECC key object resources. Returns 0 on success, < 0 on error.
ecc_import_x963_fips	Imports the ECC public key in ANSI X9.63 format from in of size inLen. Returns 0 on success, < 0 on error.
ecc_sign_hash_fips	Performs ECC key Signing operation on in of length inlen and output to out of length outlen using rng. Returns 0 on success, < 0 on error.
ecc_verify_hash_fips	Performs ECC key Verification of sig of size siglen, with hash of length hashlen. The signature verification result is returned in res. Returns 0 on success, < 0 on error.
Generate Key Pair Service	
MakeRsaKey_fips	Generates an RSA key with modulus length size and exponent e using the random number generator rng. Returns 0 on success, < 0 on error.
CheckProbablePrime_fips	For a potential modulus of length nlen, check the candidate numbers pRaw of size pRawSz and qRaw of size qRawSz to see if they are probably prime. They should both have a GCD with the exponent eRaw of size eRawSz of 1. The prime candidates are checked with Miller-Rabin. The result is written to isPrime. Returns 0 on success, < 0 on error.
RsaExportKey_fips	Exports the RSA key as its components e of eSz, n of nSz, d of dSz, p of pSz, q of qSz. The sizes should be the sizes of the buffers, and are updated to the actual length of number. Returns 0 on success, < 0 on error.

ecc_make_key_fips	Performs the ECC Key Generation operation on key of size keysize using rng. Returns 0 on success, < 0 on error.
ecc_make_key_ex_fips	Performs the ECC Key Generation operation on key of size keysize with elliptic curve curve_id using rng. Returns 0 on success, < 0 on error.
ecc_export_x963_fips	Exports the ECC public key in ANSI X9.63 format to out of size outLen. Returns 0 on success, < 0 on error.
InitDhKey_fips	Initializes DH key object for use. Returns 0 on success, < 0 on error.
FreeDhKey_fips	Releases DH key resources. Returns 0 on success, < 0 on error.
DhSetKeyEx_fips	Sets the group parameters for the DH key from the unsigned binary inputs p of size pSz, q of size qSz, and g of size gSz. Returns 0 on success, < 0 on error.
DhGenerateKeyPair_fips	Generates the public part pub of size pubSz, private part priv of size privSz using rng for DH key. Returns 0 on success, < 0 on error.
CheckRsaKey_fips	Performs a pair-wise key validation on key. Returns 0 on success, < 0 on error.
ecc_check_key_fips	Performs a pair-wise key validation on key. Returns 0 on success, < 0 on error.
DhCheckPubKeyEx_fips	Performs a mathematical key validation on the pub value of size pubSz using the domain parameters in key or against the prime value of size primeSz.
DhCheckPrivKeyEx_fips	Performs a mathematical key validation on the priv value of size privSz using the domain parameters in key or against the prime value of size primeSz.
DhCheckKeyPair_fips	Performs a pair-wise key validation between the pub value of size pubSz and the priv value of size privSz using domain parameters key. Returns 0 on success, < 0 on error.
HKDF_fips	Performs HMAC based Key Derivation Function using a hash of type and inKey of size inKeySz, with a salt of length saltSz and info of infoSz. The key is written to out of size outSz. Returns 0 on success, < 0 on error.
Key Agreement Service	
ecc_shared_secret_fips	Performs ECDHE Key Agreement operation with privKey and the peer's pubKey and storing the result in sharedSecret of length sharedSz. Returns 0 on success, < 0 on error.
DhAgree_fips	Creates the agreement agree of size agreeSz using DH key private priv of size privSz and peer's public key otherPub of size pubSz. Returns 0 on success, < 0 on error.
Key Transport Service	
RsaPublicEncrypt_fips	Performs RSA key Public Encryption on input in of size inLen, writes to output out of size outLen using rng. Returns 0 on success, < 0 on error.
RsaPublicEncryptEx_fips	Performs RSA key Public Encryption on input in of size inLen, writes to output out of size outLen using rng. It uses padding of type. If using

	PSS padding, it uses hash and mgf, with label of size labelSz. Returns 0 on success, < 0 on error.
RsaPrivateDecryptInline_fips	Performs RSA key Private Decryption without allocating temporary memory on input in of size inLen, writes to output out. Returns 0 on success, < 0 on error.
RsaPrivateDecryptInlineEx_fips	Performs RSA key Private Decryption without allocating temporary memory on input in of size inLen, writes to output out. It uses padding of type. If using PSS padding, it uses hash and mgf, with label of size labelSz. Returns 0 on success, < 0 on error.
RsaPrivateDecrypt_fips	Performs Rsa key Private Decryption on input in of size inLen, writes to output out of size outLen. Returns 0 on success, < 0 on error.
RsaPrivateDecryptEx_fips	Performs Rsa key Private Decryption on input in of size inLen, writes to output out of size outLen. It uses padding of type. If using PSS padding, it uses hash and mgf, with label of size labelSz. Returns 0 on success, < 0 on error.
Keyed Hash Service	
HmacSetKey_fips	Initializes hmac object with key of size keySz using the hash type. Returns 0 on success, < 0 on error.
HmacUpdate_fips	Performs hmac Update on input data of size len. Returns 0 on success, < 0 on error.
HmacFinal_fips	Performs hmac Final, outputs digest to hash. Returns 0 on success, < 0 on error.
Gmac_fips	Performs GMAC on input authIn of size authInSz and outputs authTag of size authTagSz. Uses key of length keySz and randomly generates an IV of length ivSz stored in iv using random number generator rng. GMAC Returns 0 on success, < 0 on error.
GmacVerify_fips	Verifies GMAC authTag of length authTagSz on input authIn of size authInSz using the key of length keySz and the iv of length ivSz. Returns 0 on success, < 0 on error.
InitCmac_fips	Initializes cmac object with key of size keySz using the hash type. Returns 0 on success, < 0 on error.
CmacUpdate_fips	Performs cmac Update on input in of size inSz. Returns 0 on success, < 0 on error.
CmacFinal_fips	Performs cmac Final, outputs digest to out of size outSz, which is updated with the actual output size. Returns 0 on success, < 0 on error.
Message Digest Service	
InitSha_fips	Initializes sha object for use. Returns 0 on success, < 0 on error.
ShaUpdate_fips	Performs sha Update on input data of size len. Returns 0 on success, < 0 on error.

ShaFinal_fips	Performs sha Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha224_fips	Initializes sha224 object for use. Returns 0 on success, < 0 on error.
Sha224Update_fips	Performs sha224 Update on input data of size len. Returns 0 on success, < 0 on error.
Sha224Final_fips	Performs sha224 Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha256_fips	Initializes sha256 object for use. Returns 0 on success, < 0 on error.
Sha256Update_fips	Performs sha256 Update on input data of size len. Returns 0 on success, < 0 on error.
Sha256Final_fips	Performs sha256 Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha384_fips	Initializes sha384 object for use. Returns 0 on success, < 0 on error.
Sha384Update_fips	Performs sha384 Update on input data of size len. Returns 0 on success, < 0 on error.
Sha384Final_fips	Performs sha384 Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha512_fips	Initializes sha512 object for use. Returns 0 on success, < 0 on error.
Sha512Update_fips	Performs sha512 Update on input data of size len. Returns 0 on success, < 0 on error.
Sha512Final_fips	Performs sha512 Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha3_224_fips	Initializes sha3 (224-bit) object for use. Returns 0 on success, < 0 on error.
Sha3_224_Update_fips	Performs sha3 (224-bit) Update on input data of size len. Returns 0 on success, < 0 on error.
Sha3_224_Final_fips	Performs sha3 (224-bit) Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha3_256_fips	Initializes sha3 (256-bit) object for use. Returns 0 on success, < 0 on error.
Sha3_256_Update_fips	Performs sha3 (256-bit) Update on input data of size len. Returns 0 on success, < 0 on error.
Sha3_256_Final_fips	Performs sha3 (256-bit) Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha3_384_fips	Initializes sha3 (384-bit) object for use. Returns 0 on success, < 0 on error.
Sha3_384_Update_fips	Performs sha3 (384-bit) Update on input data of size len. Returns 0 on success, < 0 on error.

Sha3_384_Final_fips	Performs sha3 (384-bit) Final, outputs digest to hash. Returns 0 on success, < 0 on error.
InitSha3_512_fips	Initializes sha3 (512-bit) object for use. Returns 0 on success, < 0 on error.
Sha3_512_Update_fips	Performs sha3 (512-bit) Update on input data of size len. Returns 0 on success, < 0 on error.
Sha3_512_Final_fips	Performs sha3 (512-bit) Final, outputs digest to hash. Returns 0 on success, < 0 on error.
Random (Number Generation) Service	
InitRng_fips	Initializes RNG object for use. Returns 0 on success, < 0 on error.
InitRngNonce_fips	Initializes RNG object for use with a nonce of size nonceSz. Returns 0 on success, < 0 on error.
FreeRng_fips	Releases RNG resources and zeros out state. Returns 0 on success, < 0 on error. Also part of Zeroize Service.
RNG_GenerateBlock_fips	Retrieves block of RNG output for user into buf of size in bytes bufSz. Returns 0 on success, < 0 on error.
RNG_HealthTest_fips	When reseed is 0, tests the output of a temporary instance of an RNG against the expected output of size in bytes outputSz using the seed buffer entropyA of size in bytes entropyASz, where entropyB and entropyBSz are ignored. When reseed is 1, the test also reseeds the temporary instance of the RNG with the seed buffer entropyB of size in bytes entropyBSz and then tests the RNG against the expected output of size in bytes outputSz. Returns 0 on success, < 0 on error.
Show Status Service	
wolfCrypt_GetStatus_fips	Returns the current status of the module. A return code of 0 means the module is in a state without errors. Any other return code is the specific error state of the module.
wolfCrypt_GetVersion_fips	Returns a pointer to the null-terminated char string of the wolfCrypt library version.
wolfCrypt_GetCoreHash_fips	Returns a pointer to the null-terminated char string of the core hash in hex.
Symmetric Cipher Service	
AesSetKey_fips	Initializes aes object with userKey of length keylen, dir indicates the direction while iv is optional. Returns 0 on success, < 0 on error.
AesSetIV_fips	Initializes aes object with user iv. Returns 0 on success, < 0 on error.
AesCbcEncrypt_fips	Performs aes CBC Encryption on input in to output out of size sz. Returns 0 on success, < 0 on error.
AesCbcDecrypt_fips	Performs aes CBC Decryption on input in to output out of size sz. Returns 0 on success, < 0 on error.

AesEcbEncrypt_fips	Performs aes ECB Encrypt on input in to output out of size sz. Returns 0 on success, < 0 on error.
AesEcbDecrypt_fips	Performs aes ECB Encryption on input in to output out of size sz. Returns 0 on success, < 0 on error.
AesCtrEncrypt_fips	Performs aes CTR Encryption on input in to output out of size sz. Returns 0 on success, < 0 on error. This API also performs CTR Decryption.
AesGcmSetKey_fips	Initializes aes object with key of length len. Returns 0 on success, < 0 on error.
AesGcmSetExtIV_fips	Initializes aes object with an externally generated iv of length ivSz. Returns 0 on success, < 0 on error.
AesGcmSetIV_fips	Initializes aes object with an internally generated IV of length ivSz using ivFixed as the first ivFixedSz bytes and the remainder being random bytes from rng. Returns 0 on success, < 0 on error.
AesGcmEncrypt_fips	Performs aes GCM Encryption on input in to output out of size sz. The current IV is stored in buffer ivOut of length ivOutSz. The authentication tag is stored in buffer authTag of size authTagSz. authInSz bytes from authIn are calculated into the authentication tag. Returns 0 on success, < 0 on error.
AesGcmDecrypt_fips	Performs aes GCM Decryption on input in to output out of size sz using iv of size ivSz. The authTag of size authTagSz is checked using the input and the authInSz bytes of authIn. Returns 0 on success, < 0 on error.
AesCcmSetKey_fips	Initializes aes object with key of length keySz. Returns 0 on success, < 0 on error.
AesCcmSetNonce_fips	Initializes aes object with an externally generated nonce of length nonceSz. Returns 0 on success, < 0 on error.
AesCcmEncrypt_fips	Performs aes CCM Encryption on input in to output out of size inSz. The current IV is stored in buffer nonce of length nonceSz. The authentication tag is stored in buffer authTag of size authTagSz. authInSz bytes from authIn are calculated into the authentication tag. Returns 0 on success, < 0 on error.
AesCcmDecrypt_fips	Performs aes CCM Decryption on input in to output out of size inSz using nonce of size nonceSz. The authTag of size authTagSz is checked using the input and the authInSz bytes of authIn. Returns 0 on success, < 0 on error.
Des3_SetKey_fips	Initializes des3 object with key, dir indicates the direction while iv is optional. Returns 0 on success, < 0 on error.
Des3_SetIV_fips	Initializes des3 object with User iv. Returns 0 on success, < 0 on error.
Des3_CbcEncrypt_fips	Performs des3 Cbc Encryption on input in to output out of size sz. Returns 0 on success, < 0 on error.

Des3_CbcDecrypt_fips	Performs des3 Cbc Decryption on input in to output out of size sz. Returns 0 on success, < 0 on error.
Zeroize Service	
FreeRng_fips	Destroys RNG CSPs. All other services automatically overwrite memory bound CSPs. Returns 0 on success, < 0 on error. Cleanup of the stack is the duty of the application. Restarting the general-purpose computer clears all CSPs in RAM.

Table 3: Map of FIPS 140-2 Services to API Entry Points

6 Seeding the Random Service and Key Generation

Random values generated by the wolfCrypt library for use as cryptographic keys must use the correctly instantiated wolfCrypt *Random (Number Generation) Service* for compliance with FIPS 140-2. This section describes the expected sequence of API calls, as well as best practice and FIPS compliance considerations.

The general sequence of Random service API calls is:

- Obtain sufficient entropy if a source other the Intel HWRNG is used (see below)
- Call InitRNGNonce_fips to instantiate the random generator
    - InitRNG_fips includes the required RNG Instantiate health test
- Call RNG_GenerateBlock_fips to generate a block of random bits
    - InitRNG_fips includes the required RNG health test
- Call RNG_HealthTest_fips to run DRBG health tests (as needed)
- Call FreeRNG_fips to release the resource (destroying the associated secrets)

If wolfCrypt is compiled to run on an Intel architecture that supports RDSEED (Broadwell and later) with the wolfCrypt *intelasm* compiler option enabled, the wolfCrypt Random Number Generation Service obtains entropy input from the RDSEED instruction to seed the Hash_DRBG via an implementation of hash_df. Assurance of the DRBG security strength and hence keys that are produced using the DRBG is possible only if the RDSEED function is used.

If wolfCrypt is built with *intelasm* enabled and run on a processor that does not support RDSEED mechanism, the call to InitRng_fips() will fail with the error **RNG_FAILURE_E**.

If the conditions above for use of RDSEED are not met, to meet the intent of FIPS 140-2, the developer must provide their own seeding function as a callback from wolfCrypt. The FIPS 140-2 validation process applies an assurance caveat to this situation:

When entropy is externally loaded, no assurance of the minimum strength of generated keys.

The responsible developer should obtain entropy input from a strong entropy source to assure cryptographic strength of solutions built with wolfCrypt under these conditions. To provide seed material from an external source, in your copy of the user settings file, add a prototype for your seed generation function, the static inline function, and preprocessor macro it so the library can find it during the build:

```
static inline int myGenerateSeed(unsigned char* output,
                                 unsigned size);
static inline int myGenerateSeed(unsigned char* output,
                                 unsigned size)
{ /* your code */ return 0; }
#define CUSTOM_RAND_GENERATE_SEED myGenerateSeed
```

Entropy considerations

If the target environment is not Intel-based, the processor may have its own hardware RNG, with the means to obtain random values from the device. Use a significant engineering margin amount to determine the amount of entropy needed: for parts with an entropy analysis available, use at least four times that amount. For example, the Intel entropy source has a publicly available analysis to support a 0.5 min_entropy claim; the default wolfCrypt design uses 2048 bits (hence, effectively 1024 bits of entropy) as entropy input (with an additional 1024 bits for the nonce, if no nonce input is provided). The wolfCrypt DRBG design uses a SP 800-90A rev 1 compliant SHA-256 Hash DRBG, which requires that the entropy input provided by myGenerateSeed contain at least 256 bits of entropy for full security strength.

The nonce value provided to InitRNGNonce_fips is not required by FIPS 140-2 to have provable entropy. However, the nonce should vary from one instance to the next – for example, a monotonically increasing counter or a second entropy source, like the Linux random subsystem, can be used to form a nonce.

To use the wolfCrypt library in a FIPS 140-2 validation with an external entropy source, you will need appropriate assurance documentation for the entropy source. A complete discussion of entropy analysis methods and best practice is out of scope for this guide; however, the following documents provide useful guidance and analysis:

- SP 800-90B *Recommendation for the Entropy Sources Used for Random Bit Generation*.
- FIPS 140-2 Implementation Guidance, article 7.15 *Entropy Assessment*
- Documentation and Analysis of the Linux Random Number Generator

Note that the characteristics of the Linux random subsystem vary widely across versions – the latter document provides a thorough discussion of entropy for current Linux variants.

7 Initialize and Free

For most algorithms, a call to an initialization function for the data structure associated with it. When completing use of the algorithm the paired free function must be performed. The free function will ensure any secure parameters in the structure are properly zeroized.