XAPP1239 (v1.1) July 16, 2018

Using Encryption to Secure a 7 Series FPGA Bitstream

Author: Kyle Wilkinson

Summary

This application note describes a simple step-by-step process to generate an encrypted bitstream and encryption key using the Xilinx® Vivado® Design Suite. Steps to program that encryption key and encrypted bitstream into a Xilinx 7 series FPGA using the Vivado Design Suite are included.

Introduction

Xilinx 7 series devices have on-chip Advanced Encryption Standard (AES) decryption logic to provide a high degree of design security. Encrypted 7 series FPGA designs cannot be copied or reverse engineered for use on unintended FPGAs. The 7 series FPGA AES system consists of software-based bitstream encryption and on-chip bitstream decryption with dedicated memory for storing the encryption key. Xilinx Vivado tools are optionally used to generate the encryption key and the encrypted bitstream. A user-generated key from a truly random source is recommended. The 7 series devices store the encryption key internally in either dedicated RAM, backed up by a small externally connected battery (BBRAM), or in the eFUSE. The encryption key can only be programmed onto the device through the JTAG port. The 7 series device performs the reverse operation, decrypting the incoming bitstream during configuration. The 7 series FPGA AES encryption logic uses a 256-bit encryption key. The on-chip AES decryption logic cannot be used for any purpose other than bitstream decryption. AES decryption logic is not available to the user design and cannot be used to decrypt data other than the configuration bitstream.

Advanced Encryption Standard (AES) and Authentication

The 7 series FPGA encryption system uses the Advanced Encryption Standard (AES) encryption algorithm. AES is an official standard supported by the National Institute of Standards and Technology (NIST) and the U.S. Department of Commerce (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf). The 7 series FPGA AES encryption system uses a 256-bit encryption key (the alternate key lengths of 128 and 192 bits described by NIST are not implemented) to encrypt or decrypt blocks of 128 bits of data at a time. According to the NIST, there are 1.1 x 10^{77} possible key combinations for a 256-bit key. Symmetric encryption algorithms such as AES use the same key for encryption and decryption. The security of the data is therefore dependent on the secrecy of the key.

The AES supported in 7 series FPGAs is identical to that supported in Xilinx Virtex®-6 devices. (The AES support has been validated, see the *Advanced Encryption Standard Algorithm Validation List at* http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html#2363.) A 256-bit encryption key is loaded into the eFUSE bits or battery-backed RAM by the user. The Xilinx bitstream writer, using AES, encrypts the bitstream. This feature allows you to encrypt your bitstream using 256-bit AES encryption in cipher block chaining (CBC) mode. You can supply a 128-bit Initial Vector and 256-bit key, or let the software choose a random key. Allowing the Vivado Design Suite to *generate* the key is not as secure as generating your own key by means of a a truly random process (see XAPP1084 [Ref 1]). Some security features such as the ability for the FPGA logic to clear the AES key from battery-backed RAM require that the part is configured with an encrypted bitstream in order to function.

7 series devices also have an on-chip bitstream keyed-Hash Message Authentication Code (HMAC) algorithm implemented in hardware to provide additional security beyond that provided by the AES decryption alone. (See FIPS PUB 198-1, *HMAC Federal Information Processing Standards at* http://www.nist.gov/itl/upload/FIPS-198-1_final.pdf.) The additional security provides cryptographically strong authentication of the decrypted bitstream to prove that not even a single bit was modified. Without knowledge of the AES and HMAC keys, the bitstream cannot be loaded, modified, intercepted, or cloned. AES provides the basic design security to protect the design from copying or reverse engineering, while HMAC provides assurance that the bitstream provided for the configuration of the FPGA was the unmodified bitstream allowed to load. Any bitstream tampering, including single bit flips, are detected.

The HMAC algorithm uses a key that is provided to the Xilinx software. Alternately, the software can automatically generate a random key. The HMAC key is separate and different from the AES key. The Xilinx software then utilizes the key and the Secure Hash Algorithm (SHA) to generate a 256-bit result called the Message Authentication Code (MAC). The MAC and HMAC key are transmitted as part of the AES encrypted bitstream, verifying both data integrity and authenticity of the bitstream. Authentication covers the entire bitstream for all types of control and data. When used, the 7 series FPGA security solution always consists of both HMAC and AES.

Encrypted Bitstream Implementation Overview

The following is a list of six fundamental steps needed to implement an encrypted design in a Xilinx 7 series FPGA:

1. Choose an AES key storage location: BBRAM or eFUSE; and corresponding security options (see XAPP1084 [Ref 1] for trade-off between BBRAM and eFUSE).
2. Implement the hardware requirements in your board design, based on your AES key storage selection.
3. Using Vivado Design Suite software, generate an AES key or provide your own custom AES and HMAC keys to the software (which is always the most secure approach) and encrypted bitstream.
4. Program the AES key into the FPGA using JTAG interface.

5. Program the encrypted bit file into the FPGA via JTAG or other configuration mode such as SPI or BPI, and ensure that the DONE pin asserts.
6. Perform hardware validation to ensure proper operation.

Hardware Board Requirements

There are a few basic hardware requirements needed to implement an encrypted design flow:

- For programming ability and debugging capability: JTAG connector to FPGA.
- For BBRAM key storage: Battery to V_{CCBATT} (see data sheet for battery voltage requirements)
- For eFUSE key storage: Recommend V_{CCBATT} to V_{CCAUX} to enable the ability to test with BBRAM flow prior to burning the eFUSEs.

Software Requirements

Vivado Design Suite 2014.3.1 or newer is required.

AES Key Storage

There are two options for AES key storage; Battery backed RAM (BBRAM) or eFUSE. When selecting the BBRAM or eFUSE storage options it is highly recommended that you consider the advantages and disadvantages of each option and which option fits your design requirements best. Refer to the following sections for details on each of their respective advantages and disadvantages. Additional information on each of these storage options can be found in the *7 Series FPGAs Configuration User Guide* (UG470) [Ref 2].

BBRAM

When an encryption key is stored in the FPGA's battery-backed RAM, the encryption key memory cells are volatile and must receive continuous power to retain their contents. During normal operation, these memory cells are powered by the auxiliary voltage input (V_{CCAUX}). A separate V_{CCBATT} power input is recommended for retaining the key when V_{CCAUX} is removed. Therefore it is recommended that the AES key be programmed in-system on a board that has the battery back-up. Otherwise, the key is lost when power/battery is removed. BBRAM storage location advantages and disadvantages are identified in Table 1.

Table 1: **BBRAM Storage Location Advantages and Disadvantages**

Advantages	Disadvantages
• Volatile and reprogrammable • Passive and active key clearing (i.e., the evidence can be removed) • Tamper resistant	• Requires an external battery • Many battery vendors do not specify operation at high temperatures and/or long lifetimes

eFUSE

eFUSE is a nonvolatile one-time-programmable technology used for selected configuration settings. The fuse link is programmed (or burned or blown) by flowing a large current for a specific amount of time. User-programmable eFUSEs can be programmed with the Xilinx configuration tools. Again it is important to mention that eFUSE bits are one-time programmable (OTP). After they are programmed, they cannot be unprogrammed. For example, if access to a register is disabled, it cannot be re-enabled. The FPGA logic can access only the FUSE_USER register value. All other eFUSE bits are not accessible from the FPGA logic. eFUSE storage location advantages and disadvantages are identified in Table 2.

Table 2: **eFUSE Storage Location Advantages and Disadvantages**

Advantages	Disadvantages
• No external battery required • If the CFG_AES_Only security eFUSE bit is set, only a bitstream encrypted with the eFUSE key can be loaded into the FPGA	• Permanent: Key can NOT be cleared • Less secure than BBRAM solution

eFUSE Registers

A 7 series FPGA has a total of four eFUSE registers: FUSE_KEY, FUSE_CNTL, FUSE_USER, and FUSE_DNA. For the purpose of this application note we will only focus on the FUSE_KEY, FUSE_CNTL, and FUSE_USER registers. eFUSE registers are described in Table 3.

Table 3: **eFUSE Register Description**

Register Name	Size (Bits)	Contents	Description
FUSE_KEY	256	Bitstream encryption key [0:255] (bit 255 shifted first)	Stores a key for use by AES bitstream decryptor. The eFUSE key can be used instead of the key stored in battery-backed SRAM. The AES key is used by the 7 series FPGA decryption engine to load encrypted bitstreams. Depending on the read/write access bits in the CNTL register, the AES key can be programmed and read through the JTAG port.

Table 3: **eFUSE Register Description *(Cont'd)***

Register Name	Size (Bits)	Contents	Description
FUSE_CNTL	14	Control Bits CNTL [13:0] (bit 0 shifted first)	Controls key use and read/write access to eFUSE registers. This register can be programmed and read through the JTAG port.
FUSE_USER	32	User Defined [31:0] (bit 0 shifted first)	Stores a 32-bit user-defined code. This register is readable from the FPGA logic using the EFUSE_USR primitive. See the 7 Series Libraries Guide for a description of the EFUSE_USR primitive. Depending on the read/write access bits in the CNTL register, the code can be programmed and read through the JTAG port.

eFUSE Control Register (FUSE_CNTL) Description

This register contains user programmable bits. These bits, described in Table 4, are used to select AES key usage and set the read/write protection for other eFUSE registers.

Table 4: **eFUSE Control Register Bit Description**

Bit Index No.	FUSE_CNTL Bit Name	Description of eFUSE Control Bit	Recommended Setting
0	CFG_AES_Only	• Configure using AES decryptor only. • When programmed to `1`, this bit forces use of AES key stored in eFUSE. • When not programmed (`0`), use of the AES decryptor, or not, is selected by the bitstream's security options. **CAUTION!** *If this bit is programmed to 1, the device cannot be used unless the AES key is known. Return material authorization (RMA) returns cannot be accepted and the Vivado tools Indirect SPI/BPI flash programming flow cannot be used if this bit is programmed.*	No (recommended to keep as `0` pending customer security requirements)
1	AES_Exclusive	• When programmed to `1`, this bit disables partial reconfiguration from external configuration interfaces. • When not programmed (`0`), partial reconfiguration is allowed from external interfaces but the partial bitstream must be encrypted with a matching key. **CAUTION!** *If this bit is programmed to 1, return material authorization (RMA) returns are limited in device analysis and debug. Instead, set the bitstream Security to Level2 which also disables partial configuration from external interfaces.*	No (keep as `0`)

Table 4: **eFUSE Control Register Bit Description *(Cont'd)***

Bit Index No.	FUSE_CNTL Bit Name	Description of eFUSE Control Bit	Recommended Setting
2	W_EN_B_Key_User	• Write enable (active-Low) the key and user-defined eFUSE value. • When programmed to `1`, this bit disables programming of the AES key and user-defined value bits. **RECOMMENDED:** *Program this bit after programming the key to prevent unintended changes/corruption to the eFUSE AES key value.*	Yes (program to `1`)
3	R_EN_B_Key	• Read enable (active-Low) the key. • When programmed to `1`, this bit disables reading of the AES Key and programming of the AES Key and user-defined value bits. **CAUTION!** *This bit must not be left unprogrammed (`0`) after key programming because the eFUSE AES key would be readable via the JTAG interface.*	Yes (program to `1`)
4	R_EN_B_User	• Read enable (active-low) the user-defined eFUSE value. • When programmed to `1`, this bit disables reading of the user-defined value via the JTAG and also has the side-effect of disabling programming of the AES Key and user-defined value bits. ***Note:*** The user-defined value can always be accessed by the FPGA design via the EFUSE_USR primitive.	No (keep as `0`)
5	W_EN_B_Cntl	• Write enable (active-Low) the FUSE_CNTL eFUSE bits. • When programmed to `1`, this bit disables programming of the FUSE_CNTL bits. **RECOMMENDED:** *Program this bit to 1 after programming the FUSE_CNTL register bits to prevent unintended changes to the FUSE_CNTL eFUSE bits.*	Yes (program to `1`)

When FUSE_CNTL[0] is *NOT* programmed:

- Encryption can be enabled or disabled via the bitstream options.
- The AES key stored in eFUSE or battery-backed RAM can be selected via the bitstream options.

CAUTION! *When FUSE_CNTL[0] is programmed, only bitstreams encrypted with the eFUSE key can be used to configure the FPGA through external configuration ports. This precludes device configuration from Xilinx test bitstreams and Xilinx pre-built bitstreams. Thus, Xilinx does not support RMA requests nor Vivado tools indirect SPI/BPI flash programming for devices that have the FUSE_CNTL[0] bit programmed.*

External configuration ports are blocked from accessing the configuration memory after initial configuration if FUSE_CNTL[1] is programmed. The only way to reconfigure the device is to power cycle, issue a JPROGRAM or IPROG command, or pulse the PROGRAM_B pin.

Creating an Encryption Key and Encrypted Bitstream

The bitstream generator (write_bitstream), provided with the Vivado tools, can generate encrypted as well as non-encrypted bitstreams. For AES bitstream encryption, set the write_bitstream property to enable bitstream encryption. You can either specify a 256-bit key as an input to the bitstream generator or you can have the Vivado tool generate a pseudo-random key for you. The bitstream generator in turn generates an encrypted bitstream file (`.BIT`) and an encryption key file (`.NKY`). Table 5 shows the write_bitstream properties available to be defined in the XDC file and their corresponding descriptions.

Table 5: **Write_bitstream Encryption Properties**

Write_bitstream Property	Default Values	Possible Values	Description
BITSTREAM.ENCRYPTION.ENCRYPT	No	No–Yes	Encrypts the bitstream.
BITSTREAM.ENCRYPTION. ENCRYPTKEYSELECT	bbram	BBRAM, eFUSE	Determines the location of the AES encryption key to be used, either from the battery-backed RAM (BBRAM) or the eFUSE register. ***Note:*** This property is only available when the Encrypt option is set to *Yes*.
BITSTREAM.ENCRYPTION.HKEY	Pick	Pick, <hexstring>	HKEY sets the HMAC authentication key for bitstream encryption. 7 series devices have an on-chip bitstream-keyed Hash Message Authentication Code (HMAC) algorithm implemented in hardware to provide additional security beyond AES decryption alone. These devices require both AES and HMAC keys to load, modify, intercept, or clone the bitstream. The pick setting tells the bitstream generator to select a pseudo-random number for the value. To use this option, you must first set Encrypt to *Yes*.
BITSTREAM.ENCRYPTION.KEY0	Pick	Pick, <hexstring>	Key0 sets the AES encryption key for bitstream encryption. The pick setting tells the bitstream generator to select a pseudo-random number for the value. To use this option, you must first set Encrypt to *Yes*.
BITSTREAM.ENCRYPTION.KEYFILE	None	<string>	Specifies the name of the input encryption file (with a `.nky` file extension). To use this option, you must first set Encrypt to *Yes*.
BITSTREAM.ENCRYPTION.STARTCBC	Pick	Pick, <32-bit hexstring>	Sets the starting cipher block chaining (CBC) value. The pick setting enables selection of a pseudo-random number for the value.

The following is an example XDC file showing BBRAM Key storage and a custom user-defined AES key. These encryption properties are also available in the *Edit Device Properties* GUI.

```
#Encryption Settings

set_property BITSTREAM.ENCRYPTION.ENCRYPT YES [current_design]
set_property BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT BBRAM [current_design]
#set_property BITSTREAM.ENCRYPTION.ENCRYPTKEYSELECT eFUSE [current_design]
set_property BITSTREAM.ENCRYPTION.KEY0 256'h12345678ABCDDCBA1234578ABCDDCBA1234578
     ABCDDCBA1234578ABCDDCBA [current_design]

```

The NKY file generation occurs at the same time as bitstream generation. The NKY file takes the same top_level name as the bit file and is placed in the same implementation directory.

The NKY file format is:

```
KEY 0 <hex string> (256 bit AES key)
```

For example: (`top.nky`)

```
Device xc7k325t;
Key 0 12345678ABCDDCBA12345678ABCDDCBA12345678ABCDDCBA12345678ABCDDCBA;
Key StartCBC 7115e9aa80085ea3ed65d26d3a8ab608;
Key HMAC d293d51c6058430262b05521f8f67279c9abce27d5fcafcf839bbe1af46713cc;
```

Loading an Encryption Key and Encrypted Bitstream

The encryption key can only be loaded onto a device through the JTAG interface. The Vivado Device Programmer tool can accept the NKY file as an input and program the device with the key through JTAG, using a supported Xilinx programming cable. To program the key, the device enters a special key-access mode. In this mode, all FPGA memory, including the encryption key and configuration memory, is cleared. After the key is programmed and the key-access mode is exited, the key cannot be read out of the device by any means, and it cannot be reprogrammed without clearing the entire device. The key access mode is transparent to most users. The key can be programmed into the battery-backed RAM (BBRAM), which is powered by V_{CCAUX} or V_{CCBATT}, or into the eFUSE bits.

BBRAM key programming solutions include a Vivado Design Suite and JTAG cable.

> ***Note:*** Any attempted read or write access to the BBRAM via JTAG causes the BBRAM contents to be cleared and the entire configuration of the FPGA to be erased prior to access being enabled (being able to enter *key access mode*).

eFUSE key programming solutions include:

- A Vivado Design Suite and JTAG cable
- Contact an authorized Xilinx distributor for availability of device programming services.

> ***Note:*** For the eFUSE solution, it is also recommended to take the following precautions for in-system programming of the AES key:
>
> - Prevent or clear the FPGA of a configured design to minimize power supply noise within the FPGA.
> - If possible, stop board-level system clocks to also minimize system power supply noise.

After connection to a valid hardware target using the Vivado tools *HW_Manager*, right-click on the 7 series FPGA and select either *Program BBR Key...*(to use BBRAM storage) or *Program eFUSE Registers...*(to use eFUSE storage), depending on which storage option you have previously chosen (see Figure 1).

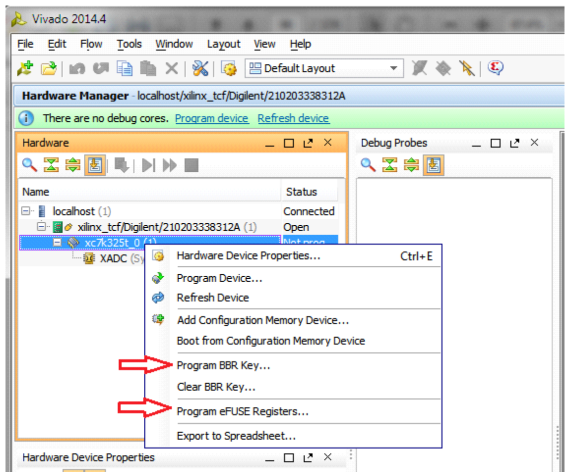

Figure 1: **Vivado Tools HW Manager Key Programming Selection**

BBRAM Key

When the *Program BBR Key* is selected you have the ability to browse to the recently generated NKY file in the project directory. After you add the .NKY file, the key value appears in the AES key field as shown in Figure 2. This allows you to check the key value and verify that this is the correct key you intend to program into the device.

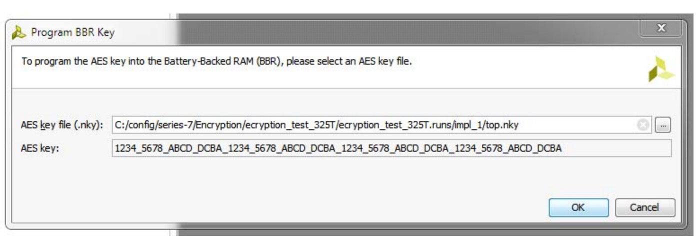

Figure 2: **BBRAM Programming GUI**

After successfully programming the NKY file into the FPGA via JTAG, the TCL console reports the following:

```
set_property ENCRYPTION.FILE
{C:/config/series-7/Encryption/ecryption_test_325T.runs/impl_1/top.nky} [get_property
PROGRAM.HW_BITSTREAM [lindex [get_hw_devices] 0]]
program_hw_devices -key {bbr} [lindex [get_hw_devices] 0]
INFO: [Labtools 27-3088] BBR Key programmed:
12345678ABCDDCBA12345678ABCDDCBA12345678ABCDDCBA12345678ABCDDCBA
INFO: [Labtools 27-3087] Key programming succeeded
INFO: [Labtools 27-3087] Key programming succeeded
```

Program eFUSE Registers

When *Program eFUSE Registers* is selected, a Wizard appears and guides you through the process of selecting the NKY file and the eFUSE registers you want to program. The eFUSE Programming GUI/AES Key Setup is shown in Figure 3.

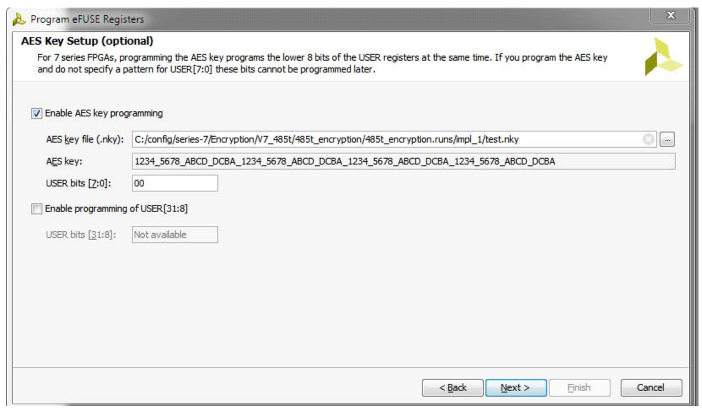

Figure 3: **eFUSE Programming GUI (AES Key Setup)**

IMPORTANT: *For 7 series FPGAs, programming the AES key and the lower 8 bits [7:0] of the FUSE_USER register occurs at the same time. Therefore if you program the AES key and do not specify a pattern for the FUSE_USER [7:0] bits, they cannot be programmed at a later time. Similarly, if you program the lower FUSE_USER bits and not the AES key then you cannot program the key at a later time.*

RECOMMENDED: *Program all 32 bits of the FUSE_USER register when you program the AES key. Refer to Table 4, page 5 for a description of the FUSE_CNTL register bits. The eFUSE Programming GUI/Control Register Setup is shown in Figure 4).*

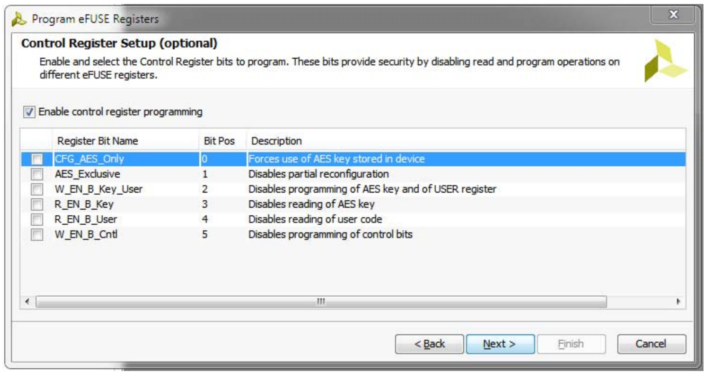

Figure 4: **eFUSE Programming GUI (Control Register Setup)**

The Tcl commands for programming the eFUSE registers are as follows:

- AES Key and entire 32 bits of FUSE_USER:

```
program_hw_devices -key {efuse} -user_efuse {xxxxxxxx} [lindex [get_hw_devices] 0]
```

- FUSE_CNTL bits:

```
program_hw_devices -control_efuse {xxxxxx} [lindex [get_hw_devices] 0]
```

After the eFUSE registers have been successfully programmed you can see the values of the FUSE_CNTL and FUSE_USER registers in the *Hardware Device Properties/EFUSE* register dropdown menu (see Figure 5), or by typing the following Tcl commands into the Tcl console:

- For the FUSE_CNTL register:

```
report_property [lindex [get_hw_device] 0] REGISTER.EFUSE.FUSE_CNTL
```

- For the FUSE_USER register:

```
report_property [lindex [get_hw_device] 0] REGISTER.EFUSE.FUSE_USER
```

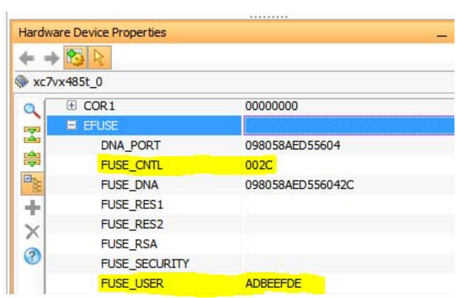

Figure 5: **Hardware Device Properties/EFUSE Register Dropdown Menu**

Loading the Encrypted Bitstream

After the device has been programmed with the correct encryption key, the device can be configured with an encrypted bitstream. After the configuration, it is not possible to read the configuration memory through JTAG or SelectMAP readback, regardless of the bitstream security setting. While the device holds an encryption key, a non-encrypted bitstream can be used to configure the device (only if the CFG_AES_ONLY bit is not programmed) and only after POR or PROGRAM_B is asserted, thus clearing out the configuration memory. In this case the key is ignored. After configuring with a non-encrypted bitstream, readback is possible (if allowed by the bitstream security setting). The encryption key still cannot be read out of the device, preventing the use of Trojan Horse bitstreams to defeat the 7 series FPGA encryption scheme.

Most methods of configuration are not affected by encryption. The 7 series FPGAs allow for bitstreams to be created with both compression and encryption. An encrypted bitstream can be delivered through any configuration interface: JTAG, serial, SPI, BPI, SelectMAP, and ICAPE2. However, an encrypted bitstream has a few limitations or timing differences for some of the configuration methods. The Slave SelectMAP and ICAPE2 interfaces accept encrypted bitstreams only through the x8 bus (x16 and x32 Slave SelectMAP are not allowed). The Master SelectMAP and Master BPI interfaces accept encrypted bitstreams through either the x8 or x16 data bus, but for the x16 bus width, the master CCLK frequency is slowed to half of the ConfigRate, or half of the EMCCLK rate when ExtMasterCCLK_en is used. The slower CCLK begins early in the bitstream when the DEC (AES encryptor enable) bit is read, before the CCLK is updated based on the ConfigRate frequency or the external EMCCLK frequency.

Advanced configuration solutions such as Tandem Configuration and Partial Reconfiguration are supported with encrypted bitstreams. Partial bitstreams can be delivered unencrypted to the ICAP, or encrypted (with the same AES key) to any configuration port, so long as the latter has not been explicitly forbidden by the designer. Setting Security Level2 (via set_property BITSTREAM.READBACK.SECURITY Level2 [current_design]) or programming the eFUSE_CNTL[1] AES_Excusive bit to a '1' prevents partial reconfiguration over external configuration ports. Fallback reconfiguration and IPROG reconfiguration are enabled in 7 series FPGAs after encryption is turned on. Readback is available through the ICAPE2 primitive. None of these events reset the key if V_{CCBATT} or V_{CCAUX} is maintained. A mismatch between the key in the encrypted bitstream and the key stored in the device causes configuration to fail with the INIT_B pin pulsing Low and then back High if fallback is enabled, and the DONE pin remaining Low. The HMAC_ERROR bit in the Config_Status register also flags if an error occurs.

To confirm in hardware that the encrypted design loaded successfully, check that the DONE pin is High or verify using other visual indicators that your design is functioning (LEDs, UARTs, etc.). To confirm in software that the encrypted design loaded successfully you can refer to the Config_Status register included in the Hardware Device Properties list. Bits 1 (DECRYPTOR_ENABLE), 4 (EOS), and 14 (DONE_PIN) are the main indicators for confirmation (see Figure 6).

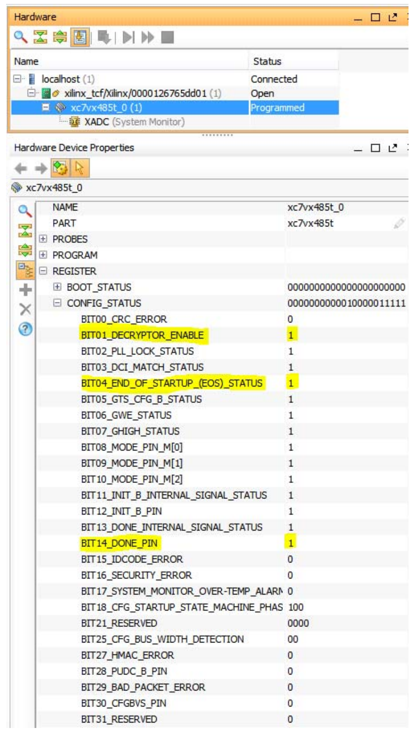

Figure 6: **Device Status Register**

Hardware Verification

You will most likely want to verify that the AES key was properly programmed into either the BBRAM or eFUSE bits properly. The following is a check list of verification steps:

1. Generate bitstreams using Vivado Design Suite 2014.3.1 or later: Unencrypted bitstream, encrypted bitstream with your personalized key, encrypted bitstream with an all-ones key, finally an encrypted bitstream with an all-zeros key
2. Review the generated bitstreams to validate encryption took place. (See Figure 7, page 15 for an example of encrypted and unencrypted bit files.)
3. On an FPGA that has not yet had its eFUSE programmed:
   a. Check hardware: Use Vivado Device Programmer to connect to the FPGA, download the unencrypted `BIT` file via JTAG. Does the design function as expected?
   b. Test FPGA decryptor: Download the encrypted `BIT` file with the all-zeros key (for eFUSE).
   c. Test encrypted bitstream security: Download the encrypted `BIT` file with your personalized key. A configuration failure is expected.
4. Program the eFUSE key and options:
   a. Power-cycle the board to assure any errors from the above tests have been cleared from the FPGA and that the FPGA is not configured.
   b. Program the AES key via JTAG. (If using eFUSE, first do steps 3b and 3c with the BBRAM key as a validation check. Then program the eFUSE for a final test.)
   c. Check key cannot be read: Use the Vivado tool to check the Hardware Device>Property>Registers>eFUSE>FUSE_CNTL and that bit 3 is programmed to `1`. Also, check that the other FUSE_CNTL bits are programmed as selected during the programming operation.
5. On the FPGA with the programmed eFUSE key and options:
   a. Test key: Download the encrypted `BIT` file with your personalized key.
   b. Test key: Download encrypted `BIT` file associated with the all-zeros key. A configuration failure is expected.
   c. Test key settings: Download the unencrypted `BIT` file. Results vary depending on security settings.

Conclusion

This application note describes AES encryption and authentication standards and identifies the advantages and disadvantages of the different key storage options available. Most importantly, it functions as an easy *how to* guide to create an AES encryption key and an encrypted bit file and to program these files into a 7 series FPGA using Vivado Design Suite software.

Appendix A: Encrypted and Unencrypted Bitstreams

The difference between an encrypted bit file and unencrypted bit file is shown in Figure 7.

Encrypted bit file

```
00000070  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000080  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000090  00 00 00 BB 11 22 00 44 FF FF FF FF FF FF FF FF
000000A0  AA 99 55 66 20 00 00 00 30 00 C0 01 00 00 00 40
000000B0  30 00 A0 01 00 00 00 40 30 01 C0 01 00 00 00 00
000000C0  20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
000000D0  20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
000000E0  20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
000000F0  20 00 00 00 20 00 00 00 30 01 60 04 71 15 E9 AA
00000100  80 08 5E A3 ED 65 D2 6D 3A 8A B6 08 30 03 40 01
00000110  00 2B A6 58 B7 76 12 F1 90 AB 70 C8 12 9E 9B 08
00000120  34 CF 59 9A 9A D9 2F 83 81 DE D7 1B 01 19 65 10
00000130  9D DA 66 07 E7 3C 07 A5 AC A2 45 93 F4 7A F9 6E
00000140  B3 92 3A BD C7 F3 7A CF 0B A2 E7 E7 51 19 DA 8E
00000150  1E 63 E9 9B FC 6C 6E A3 A8 BE 27 85 8B B8 6E DB
00000160  7E 35 A2 DF D4 5A FA A7 57 C7 F9 54 7F C7 34 C3
00000170  03 CB 16 14 E8 0C C9 D5 AF 4B 72 9F BD 48 57 76
00000180  26 01 65 4D A3 CB 85 D1 FA 2A 40 8C 51 53 64 1F
00000190  BE C0 B5 A6 84 9B 9F 13 14 AF 16 DE BD 41 B3 36
000001A0  B3 B0 19 04 37 3F 1C 15 56 E2 E9 82 5C A6 00 41
000001B0  DD 6F 5E 3A 88 1E 2D 6A 3B 51 5C 88 D9 B0 46 33
000001C0  D2 CE B3 B5 0D B8 C2 E9 41 13 4F 6E 60 43 C4 A8
000001D0  85 AF 0A 5D 64 B5 01 85 21 CF C7 37 92 CC 23 F5
000001E0  E4 C1 15 60 F1 C6 49 F9 4A FA E9 74 2D 91 B8 3C
000001F0  EB BD 8F EC 2A C2 BF 11 0E 3C 73 0B 67 8B A7 EA
00000200  EE 73 46 97 2A 56 4C C7 18 B4 F2 29 E1 1E 38 38
```

Unencrypted bit file

```
00000070  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000080  FF FF FF FF 00 00 00 BB 11 22 00 44 FF FF FF FF
00000090  FF FF FF FF AA 99 55 66 20 00 00 00 30 02 20 01
000000A0  00 00 00 00 30 02 00 01 00 00 00 00 30 00 80 01
000000B0  00 00 00 00 20 00 00 00 30 00 80 01 00 00 00 07
000000C0  20 00 00 00 20 00 00 00 30 02 60 01 00 00 00 00
000000D0  30 01 20 01 02 20 3F E5 30 01 C0 01 00 00 00 00
000000E0  30 01 80 01 03 65 10 93 30 00 80 01 00 00 00 09
000000F0  20 00 00 00 30 00 C0 01 00 00 00 01 30 00 A0 01
00000100  00 00 01 01 30 00 C0 01 00 00 00 00 30 03 00 01
00000110  00 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
00000120  20 00 00 00 20 00 00 00 20 00 00 00 20 00 00 00
00000130  20 00 00 00 30 00 20 01 00 00 00 00 30 00 80 01
00000140  00 00 00 01 20 00 00 00 30 00 40 00 50 2B A5 20
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 7: **Encrypted and Unencrypted Bit Files**

References

1. *Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs* (XAPP1084)
2. *7 Series FPGAs Configuration User Guide* (UG470)
3. *Using Advanced Encryption Standard Keys with the Battery-Backed (BBRAM* Tutorial http://www.xilinx.com/training/vivado/using-encryption-keys-with-bbram.htm

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/16/2018	1.1	Changed “configuration” to “reconfiguration” in Table 4. Clarified third paragraph under Loading the Encrypted Bitstream.
04/15/2015	1.0	Xilinx initial release.

Please Read: Important Legal Notices