XAPP1098 (v1.2) February 22, 2017

Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs

Author: Ed Peterson

Summary

This application note provides anti-tamper (AT) guidance and practical examples to help protect the intellectual property (IP) and sensitive data that might exist within a system enabled by UltraScale™ and UltraScale+™ FPGAs. This protection (in the form of tamper resistance) needs to be effective before, during, and after the FPGA has been configured by a bitstream. Sensitive data can include the configuration data that sets up the functionality of the FPGA logic, critical data and/or parameters that might be included in the bitstream (e.g., initial block RAM contents and initial state of flip-flops). It also includes external data that is dynamically brought in and out of the FPGA during post-configuration normal operation.

This document summarizes the silicon AT features available in UltraScale and UltraScale+ FPGAs, explains why these features exist, and provides use cases and implementation details for each feature. This document also provides guidance on various other methods that can be used to provide additional tamper resistance.

By following this application note, you can be assured that you are following the best AT practices available with our UltraScale and UltraScale+ FPGAs. These best practices broadly apply whether the goal is to simply prevent cloning/overbuilding of a commercial design, prevent reverse engineering of a military system's valuable critical technology (CT), or anything in-between.

This application note assumes that you are somewhat knowledgeable and proficient in Xilinx FPGA architecture [Ref 1] and design, as well as the Vivado® tools flow methodology [Ref 2] and configuration [Ref 3]. *Solving Today's Design Security Concerns* [Ref 4] and *Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs* [Ref 5] provide a good background on the various security threats and solutions for FPGAs.

Introduction

Xilinx has been at the forefront of providing FPGA AT solutions to their customers for many generations. UltraScale and UltraScale+ FPGAs continue this trend by including asymmetric authentication, side channel attack protection, and other silicon AT features. Additionally, in order to provide a number of tamper protections post configuration, Xilinx offers an IP core known as Security Monitor [Ref 6]. Due to certain restrictions, Security Monitor's availability is limited. Contact your local Xilinx representative for details.

Keeping one step ahead of the adversary is a continuous process that involves understanding the existing vulnerabilities and attacks and then developing new mitigation techniques

(countermeasures) to combat those attacks. Xilinx has a multi-generational commitment to secure FPGA technology in a cost-sensitive manner for the AT-conscious communities which include both commercial and defense markets.

By taking advantage of various Xilinx FPGA AT features, you can choose how much AT to include with the FPGA design based on program and customer requirements. AT can be in the form of enabling individual silicon AT features or a combination of these AT features (perhaps tied together by the developer in the FPGA design and following best practice guidance).

The decision as to how much AT to include primarily depends on three factors:

- Value: The perceived value of the intellectual property and the damage it might cause either financially or to national security if it were to become compromised. Certain AT features can be expensive to implement and that cost must be weighed against the value of the technology or data being protected.
- Adversary: Access to the system and the sophistication level and resources available to carry out the attack. For example, can access to the system be prevented by "guns, gates, and guards" or can it be easily obtained in the open market? Is the adversary a garage-based hacker or a nation-state? The adversary's capabilities could be at these extremes or anywhere in-between.
- Design stage: The point in the system development cycle where the decision is made to enable AT for the FPGA design. Xilinx highly recommends that the decision to utilize FPGA AT features is made very early on (i.e., after CT is defined in a system) to help address both schedule and cost concerns. It is always more costly, more time consuming, and often less effective to insert AT features later on in the development process.

Another factor that needs to be considered is how much of the FPGA's logic resources are consumed by enabling certain active AT features. The overall resource penalty is usually rather small. However, it does depend on how these features are implemented and the size of the FPGA (i.e., the larger FPGAs experience less of an impact).

Xilinx classifies the silicon AT features as either passive or active security. In general, passive security features are those that are either part of the tool flow or built into the FPGA and do not require you to do anything extra in your FPGA logic design. Passive security features are also temporal in nature—they come into effect at different times during the normal life cycle of the FPGA:

- Pre-configuration (e.g., public key authentication of the configuration bitstream)
- During-configuration (e.g., resistance to side-channel attacks via differential power analysis (DPA))
- Post-configuration (e.g., user data protection via disabling of readback)

In contrast, active security features are required to be included in the FPGA logic design. These features only come into effect after the FPGA has been configured via the user bitstream and the design becomes active. Examples are asserting KEYCLEARB to zeroize the battery-backed advanced encryption standard (AES) key or handle a PROGRAM_B intercept.

At a bare minimum, you should always plan on including the appropriate passive security features into your design (e.g., bitstream encryption and authentication). These features do not affect the function of the design. However, they might create logistical challenges (e.g., key management), system challenges (e.g., a battery is needed if using battery-backed RAM (BBRAM) for key storage), and increase the configuration time (e.g., public key authentication might increase configuration time). Otherwise, these features are freely available in terms of impact to the design and can provide a fair amount of tamper protection. For an already fielded system or a design late in the development stage, these AT features are great candidates for enabling because they don't affect the actual FPGA logic design.

Additionally, the AT features and guidance presented in this application note fall into three main AT categories:

- Prevention (e.g., JTAG port blocking)
- Detection (e.g., voltage and temperature monitoring)
- Response (e.g., BBRAM key erasure penalty)

Table 1 summarizes and classifies the built-in silicon AT features of the UltraScale and UltraScale+ FPGAs.

Table 1: **AT Features Classification and Summary**

UltraScale and UltraScale+ FPGAs Silicon AT Features	Type	Category	Life Cycle(1)
Bitstream confidentiality and authentication (symmetric)(2)	Passive	Prevention	Pre and During
Volatile 256-bit BBRAM key storage	Passive	Prevention	Pre
Non-volatile 256-bit eFUSE key storage(3)	Passive	Prevention	Pre
Write-only key load w/ integrity check (BBRAM and eFUSE)(2)	Passive	Prevention	Pre
Obfuscated key loading and storage(2)	Passive	Prevention	Pre
Bitstream authentication (asymmetric)	Passive	Prevention	Pre
Non-volatile 384-bit eFUSE public key hash storage(2)(3)	Passive	Prevention	Pre
DPA protections(2)	Passive	Prevention	During
Hardened readback disabling circuitry	Passive	Prevention	Post
JTAG port permanent disable (eFUSE)(2)(3)	Passive or Active	Prevention or Response	Pre or Post
JTAG port temporary disable	Passive or Active	Prevention	Post
JTAG port monitor	Active	Detection	Post
Configuration memory integrity checking	Active	Detection	Post
Unique identifiers (device DNA and user eFUSE)	Active	Detection	Post
On-chip temperature and voltage monitors/alarms	Active	Detection and Response	Post
Uninterruptible internal clock source	Active	Detection	Post

Table 1: **AT Features Classification and Summary *(Cont'd)***

UltraScale and UltraScale+ FPGAs Silicon AT Features	Type	Category	Life Cycle[1]
External PROGRAM_B intercept	Active	Prevention and Detection	Post
Configuration memory clearing	Active	Response	Post
Key agility (BBRAM only)[2]	Active	Response	Post
BBRAM key zeroize (erase + verify)[2]	Active	Response	Post
Non-volatile (eFUSE) tamper event logging[2]	Active	Response	Post
Bitstream decryptor permanent (eFUSE) disable[2][3]	Active	Prevention or Response	Post
Global 3-state (GTS) enable	Active	Response	Post
Global set-reset (GSR) enable	Active	Response	Post

Notes:

1. Describes when in the FPGA's life cycle this feature is effective (pre-configuration, during configuration, or post-configuration).
2. This feature is new or improved in UltraScale and UltraScale+ FPGAs.
3. Asserting some of these "permanent" tamper penalties (eFUSE-based) is irreversible and might affect whether or not the device can be returned to Xilinx. Refer to the eFUSE Security Register (FUSE_SEC) table in the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3] for details.

The following sections explore the above features in depth, providing detailed explanations on what they are, why they exist, and give specific examples on how to properly use them (either by themselves, in conjunction with other built-in features and user logic, or both). Additionally, specific guidance is given on certain methods and techniques that can be employed to increase the tamper resistance of the FPGA design and overall system.

Any AT features enabled at the FPGA level should always be part of an overall system-level AT solution. The features and techniques outlined in this document provide for a very good AT "umbrella" for the FPGA itself. However, AT is most effective when it is developed with a multi-layer approach with the entire system always in mind.

Passive AT Silicon Features

Bitstream Confidentiality and Authentication (Symmetric)

Storing an encrypted bitstream in external flash (or other means) and then decrypting it during FPGA configuration (via the FPGA's internal decryption engine) provides for a very high level of confidentiality. This ensures that information contained in the bitstream is only accessible to those who share the same (symmetric) secret key. Bitstream encryption and decryption provides confidentiality while the system is at rest and during configuration. It protects the FPGA design contents, including block RAM and flip-flop initialization data. Xilinx highly recommends that the externally stored bitstream always be in encrypted form.

Note: UltraScale and UltraScale+ FPGAs use the National Institute of Standards and Technology (NIST)-approved AES in Galois/Counter Mode (GCM) with a 256-bit key. The NIST CAVP certification for Xilinx is available at http://csrc.nist.gov/groups/STM/cavp/documents/aes/aesval.html as validation number 2800.

To take advantage of this security feature, the configuration bitstream must first be encrypted by the Vivado software using the write_bitstream Tcl command and the appropriate properties that must be defined in the XDC file [Ref 2] [Ref 3] [Ref 7]. The Vivado software uses a key supplied by the user to perform the encryption. If an AES key is not supplied, the Vivado software optionally generates one automatically. However, keys generated by the Vivado software are pseudorandom. Truly random keys are more secure. The key is then loaded into the FPGA via the JTAG port using the Vivado hardware manager.

When bitstream decryption is enabled on UltraScale and UltraScale+ FPGAs (and asymmetric RSA-2048 authentication is not enabled via eFUSE), symmetric authentication is automatically enabled because AES-GCM is an authenticated encryption and decryption algorithm. AES-GCM combines the counter mode for confidentiality with an authentication mechanism that is based on a universal hash (authentication tag) function. Therefore, AES-GCM provides not only confidentiality but integrity and authentication at the same time. This cryptographically strong authentication scheme ensures that any attempt at modifying the bitstream (even just a single bit) drastically alters the bitstream's signature, thus preventing the device from starting up. Basically, symmetric authentication guarantees the source is genuine—only the person that was authorized to configure the device can succeed in doing so.

If the authentication check passes, the device then begins normal operation (i.e., the startup commands take place). Evidence of the authentication step failing is the absence of the DONE output signal asserting High after the bitstream is loaded, the INIT_B signal asserting Low, and the HMAC_ERROR bit in the STATUS configuration register asserting High. An authentication failure might indicate that the bitstream has been tampered with. It could also indicate that the channel used for bitstream loading is noisy and bit corruption(s) are taking place during the configuration process.

Additionally, AES-GCM was designed to facilitate high-throughput hardware implementations. The AES-GCM decryptor in UltraScale and UltraScale+ FPGAs can accept encrypted bitstreams in 32-bit wide format (previous families were limited to 8-bit wide format). This allows for near parity in the device configuration time between unencrypted and encrypted bitstreams.

Volatile and Non-Volatile Key Storage

The 256-bit symmetric AES-GCM key can be loaded into either volatile BBRAM or non-volatile eFUSE one-time programmable (OTP) storage locations within the FPGA. To decide which storage location to use for the key, an understanding of the advantages and disadvantages of BBRAM (Table 2) and eFUSE (Table 3) storage is necessary.

Table 2: **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.(1)	• Requires an external battery. • Many battery vendors do not specify operation at high temperature and/or long lifetimes (although some vendors are now starting to offer betavoltaic type batteries to help address these issues).

Notes:

1. There is no physical path to read the key out of BBRAM (write-only access).

Table 3: **eFUSE Storage Location: Advantages and Disadvantages**

Advantages	Disadvantages
• No external battery required. • Makes spoofing difficult (would require the device on the board to be replaced). ◦ Only a bitstream encrypted with the eFUSE key can configure the FPGA. All others are rejected if the cfg_aes_only(1) eFUSE bit is also blown.	• Permanent; the key cannot be cleared or updated(2). • Less secure than BBRAM solution (i.e., the device level evidence remains).

Notes:

When using the cfg_aes_only option (located in the eFUSE control register FUSE_SEC) there are two important points to consider:

1. If using the indirect flash programming method [Ref 8] for the bitstream, ensure that this option (cfg_aes_only) is enabled after the on-board flash has been loaded with the encrypted bitstream. This is because the indirect programming core bitstream from Xilinx is unencrypted. Otherwise, the FPGA attempts to decrypt the indirect programming bitstream using the eFUSE key and fails configuration (and thus the update of the external flash fails as well). This also has implications for subsequent field updates of flash-stored firmware.
2. The permanent AES-GCM decryptor disable feature (via eFUSE) can be used in lieu of erasing the eFUSE-based key which prevents a DPA attack but the eFUSE key still remains in the device and is subject to a more sophisticated physical attack.

Additional details concerning bitstream encryption and key storage can be found in the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3], *Vivado Design Suite User Guide: Programming and Debugging* (UG908) [Ref 7], *Vivado Design Suite Tcl Command Reference Guide* (UG835) [Ref 9], and *Using Encryption to Secure an UltraScale FPGA Bitstream* (XAPP1267) [Ref 10].

Write-only Key Load with Integrity Check

Both the BBRAM and eFUSE 256-bit symmetric keys are loaded via the external JTAG using the Vivado hardware manager (see Key Agility (Response) on how the internal MASTER_JTAG port can be used to update the BBRAM key). For UltraScale and UltraScale+ FPGAs, this key loading path is write-only to the device. There is no physical datapath to read back either key. (In previous families the key was protected by protocol during the loading process where upon entering "key access mode" the existing key and configuration memory were immediately cleared before a new key could be written and then read back to check integrity.) When a key is

written to the device via JTAG, a key integrity check is started by writing the expected CRC32 value via JTAG to the device. An actual CRC32 integrity check is calculated on the stored key by the device (internally) and compared to the expected CRC32 that was just received via the JTAG port. A pass/fail type of result is then written out by the device to the JTAG port instead of the actual key data to signify integrity status. Removing the physical readback path for the key increases the security of the stored key.

Note: For BBRAM-based keys, prior to writing the key, the existing key in BBRAM is zeroized (erased and verified).

Obfuscated Key Loading and Storage

Optionally, key data written and stored into an UltraScale or UltraScale+ FPGA's eFUSE array or BBRAM (via JTAG) can be obfuscated. The key data is encrypted using a fixed obfuscating key that is identical for all UltraScale FPGAs and all UltraScale+ FPGAs, and is known only to Xilinx. (The UltraScale FPGA obfuscating key is different from the UltraScale+ FPGA obfuscating key).This provides for an increased level of security in commercial production situations (e.g., secret red key protection at a contract manufacturer). The internally stored obfuscated key is decrypted at the beginning of an encrypted bitstream load and then used to decrypt the bitstream that follows it. This feature is enabled by a control bit in the FUSE_SEC control register (eFUSE-based key) or by control bits written into the BBRAM (BBRAM-based key). Figure 1 provides a high-level summary of this operation.

Note: The use of obfuscated key storage is *not* compatible with the configuration counting DPA countermeasure for BBRAM key storage (see DPA Protections).

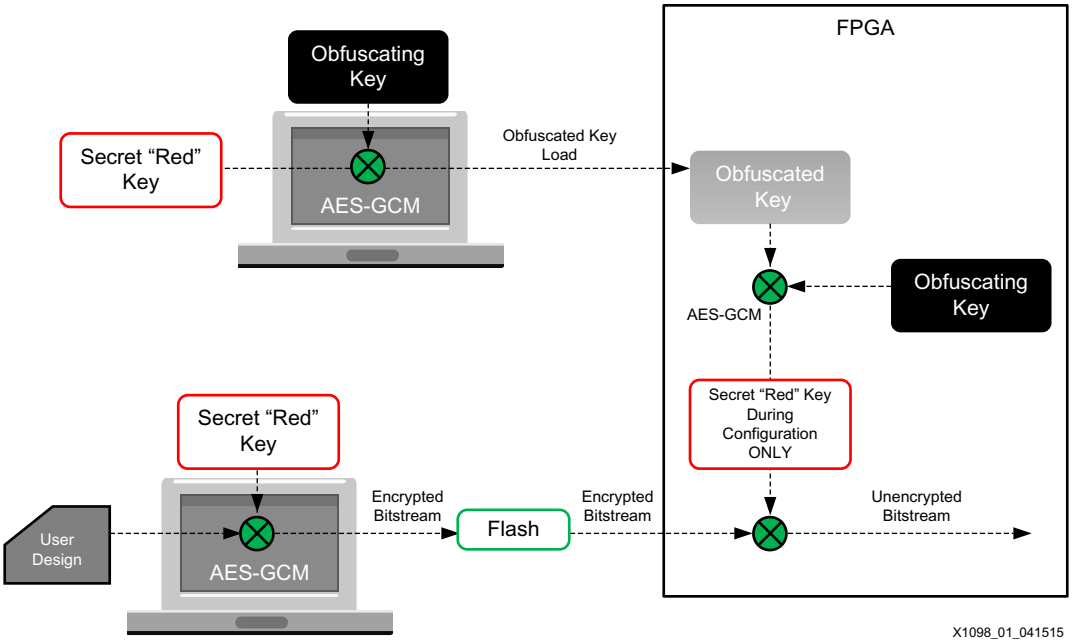

Figure 1: **Summary of Obfuscated Key Loading and Storage**

Bitstream Authentication (Asymmetric)

UltraScale and UltraScale+ FPGAs have the capability of loading the entire encrypted bitstream into the device and then authenticating it before sending it to the on-chip decryption engine (i.e., "authenticate-then-decrypt"). If the bitstream has been modified in any way (including just a single bit change), the device's asymmetric authentication function detects these change(s) and not only disables the decryption engine (if enabled for an encrypted bitstream) but also prevents the startup of the device. In short, if this feature is enabled, only an authorized bitstream can configure the UltraScale or UltraScale+ FPGA. When asymmetric authentication is enabled, the symmetric authentication functions of the AES-GCM algorithm are not performed (i.e., the periodic and complete symmetric authentication checks are not performed).

Because this method uses the RSA-2048 asymmetric digital signature (authentication) algorithm, it does not require the device to contain a "secret" in order to accomplish this authentication task. Instead, the asymmetric authentication function contains user-defined public key information. Due to limited space, the feature uses a 384-bit SHA-3 hash of the 2048-bit public key and is programmed into the eFUSE bits of the UltraScale or UltraScale+ FPGA. It is up to you to define the private and public key pairs for this operation. There are a number of open source and commercial products that can be used to create these key pairs (such as OpenSSL and SafeNet). Because this authentication scheme does not require a secret to operate, adversarial attacks such as side channel analysis do not reveal any information that is useful to an attacker.

There are several reasons to use the RSA asymmetric authentication:

1. Authenticate the entire bitstream before decrypting it. This method is part of a DPA attack countermeasure described in DPA Protections.
2. Prevent unauthorized users from ever running their own (potentially malicious) designs on the UltraScale or UltraScale+ FPGA. When an authorized user programs the public key hash into the eFUSE bits and the RSA_AUTH_ALL_EFUSE eFUSE has been programmed (forcing RSA authentication) only authorized bitstreams can be loaded.
3. Authentication of unencrypted bitstreams. An FPGA design might not contain CT but still have requirements that it be authentic. Some example use cases are:
    a. The design contains a publicly known function (such as the AES encryption algorithm). You don't need the design to be confidential but do need to ensure it hasn't been modified to output red keys or data on external pins, for example.
    b. The FPGA design has different levels of functionality (e.g., basic to advanced features). For instance, only premium-paying end customers have all the features—all others can only access the basic features. The unencrypted bitstream cannot be modified by an adversary to try and "turn on" any of the advanced features without being detected.

Figure 2 illustrates (at a high-level) how the bitstream is constructed using RSA/SHA-3[1] and how the entire bitstream is authenticated internally on-chip. The top portion of the figure is performed by the software tools to create the RSA authenticated bitstream and the bottom

1. RSA-2048 and SHA-3 do not currently follow a NIST standard. Contact Xilinx for details.

portion of the figure is done inside the UltraScale or UltraScale+ FPGA to authenticate the bitstream.

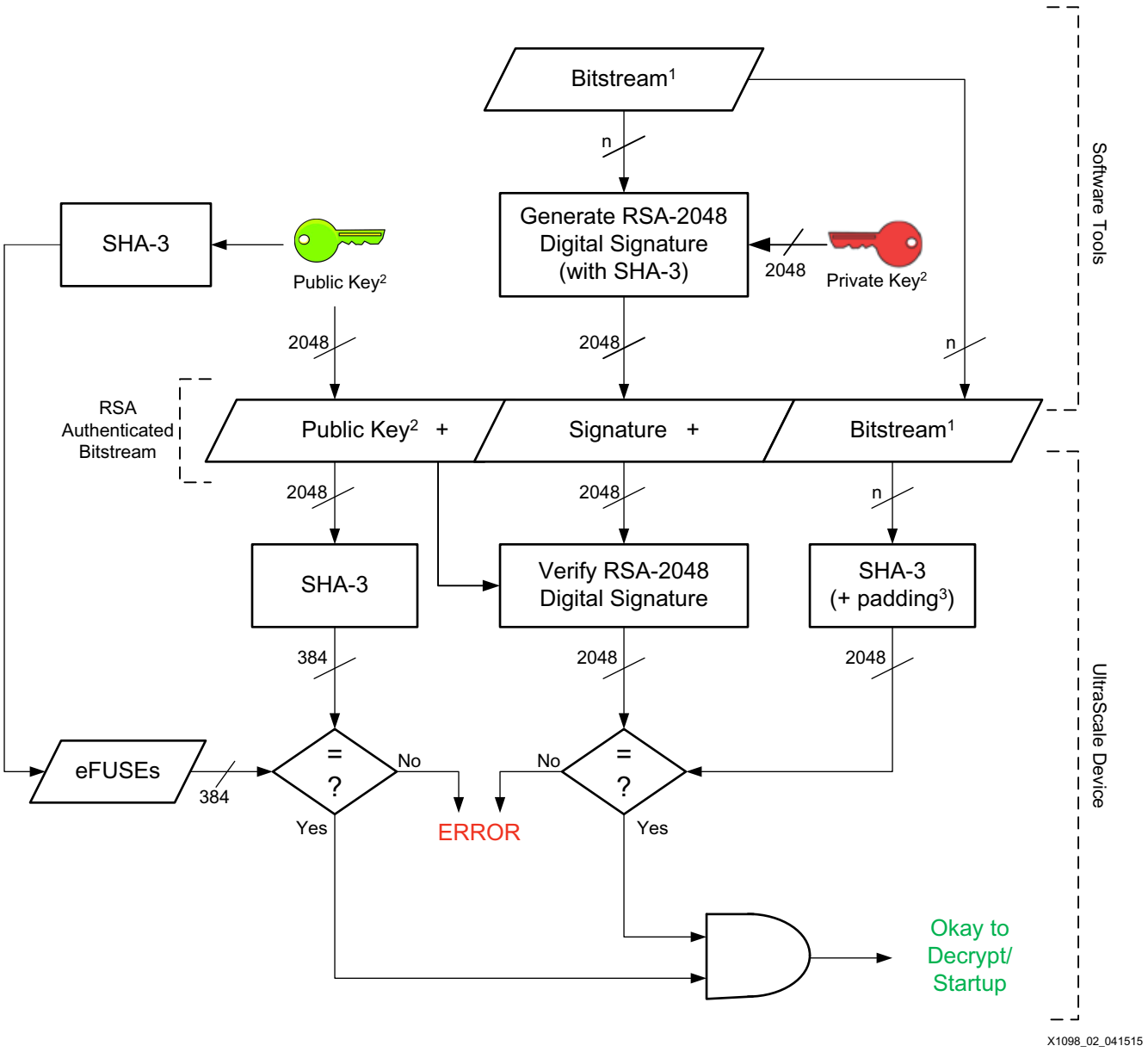

Figure 2: **Constructing the Bitstream with RSA Signature**

Notes related to Figure 2:

1. Bitstream can be encrypted (ciphertext) or unencrypted (plaintext).
2. Private/public key pair generated by the user.
3. PKCS #1 v1.5 padding scheme.

Figure 3 provides details on the actual format of an encrypted and RSA authenticated bitstream and further describes which parts are plaintext, authenticated plaintext, and authenticated ciphertext.

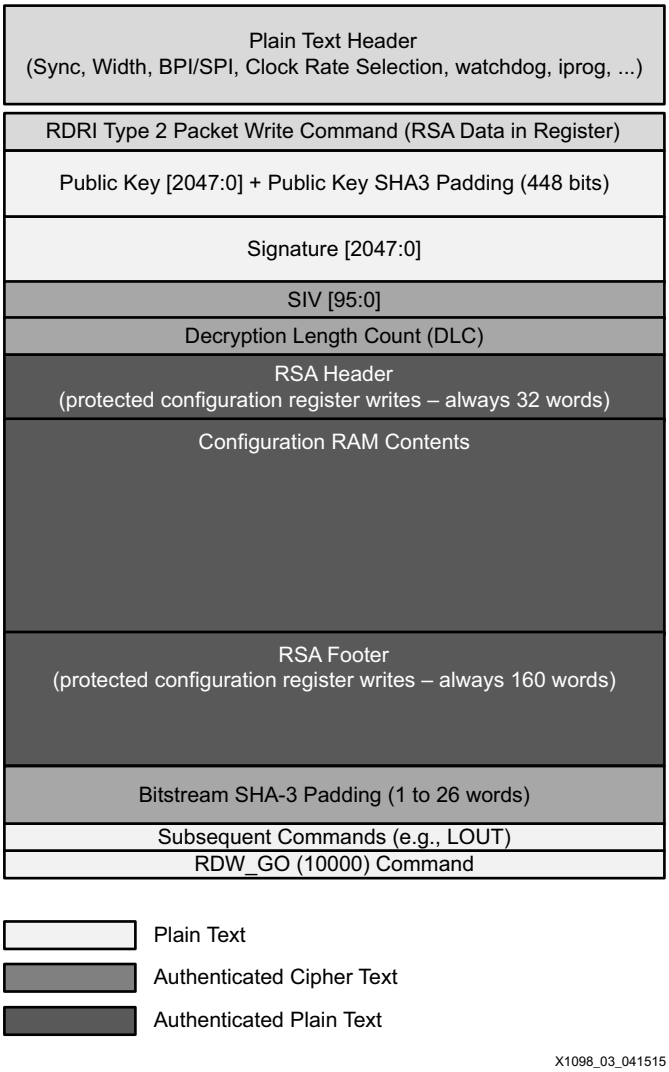

Figure 3: **Bitstream Format Using RSA**

Because the entire bitstream is loaded into the device and then authenticated prior to use, there is a configuration time penalty when using the RSA asymmetric authentication scheme (with an encrypted bitstream). However, depending on the configuration port used, the increase in overall configuration time might not be very significant. This is due to the fact that after the entire bitstream has been loaded and authenticated, the decryption process can take advantage of a full 32-bit wide data bus. Consult the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3] for precise configuration time details.

Due to certain silicon design constraints, there are some limitations when using the RSA asymmetric authentication scheme on UltraScale and UltraScale+ FPGAs:

- RSA authenticated bitstreams cannot be compressed. For details on compression, see the *UltraScale Architecture Configuration User Guide* (UG570).
- Partial reconfiguration (PR) bitstreams cannot be RSA authenticated with the built-in silicon feature. (You can put your own authentication function in the FPGA logic.) For details on PR, see Partial Reconfiguration in the Vivado Design Suite

- Tandem bitstreams cannot be RSA authenticated. Tandem configuration is the Xilinx solution for fast configuration of PCIe® designs to meet enumeration needs within open PCIe systems.
- For UltraScale FPGAs only, there are limitations on bitstream configuration widths when RSA is enabled. Some of the narrow configuration widths are not supported. See *UltraScale FPGA RSA Authentication and Supporting Configuration Modes* (XCN15038) [Ref 11] for additional details.

DPA Protections

Instead of trying to directly attack a security function (e.g., breaking FPGA bitstream AES-256 decryption using a non-feasible brute-force key attack), an attacker will often look for an easier solution such as side-channel analysis. The *side channel* is an unintentional information leakage path that might exist in an electronic device. If observed for a long enough period, it might be possible to extract secret information from it (such as a cryptographic function's red key data).

Differential power analysis (DPA) is a side-channel technique that observes and records samples of the power supply fluctuations due to digital switching (by monitoring voltage across a low resistance in series with the power line(s) or nearby electromagnetic probing) of a functioning electronic device. Signal processing and statistical methods are then applied to the recorded data to extract red key data. The number of data samples required has been consistently reduced over time as the capabilities of the attackers have improved.

Xilinx provides DPA resistance by limiting the amount of side channel data that an adversary can collect on any one key. This protocol-based data limiting technique is used on the UltraScale and UltraScale+ FPGAs to mitigate against DPA attacks of the on-chip bitstream decryptor. This technique offers the most long-term flexibility because the level of protection is programmable and can be increased as the capabilities of the attacker improve.

There are two types of data that must be limited in this regard: invalid/random bitstream data and valid bitstream data. To be effective, there must be countermeasures for both invalid/random and valid bitstream data attacks.

Invalid/Random Bitstream Data

Traditionally, a DPA attacker would simply feed an unlimited amount of random data into the decryptor's ciphertext input port and then collect the side-channel information for analysis. There are two methods by which UltraScale and UltraScale+ FPGAs can detect this random (or invalid) bitstream. To accomplish this, the user selects only one of the following two methods:

1. Configuration counting quickly detects an invalid bitstream in real time during the decryption process. UltraScale and UltraScale+ FPGAs add a 32-bit periodic symmetric authentication check every eight words. Decryption is halted immediately if the periodic authentication fails and the UltraScale or UltraScale+ FPGA marks it as an invalid configuration attempt. This method should only be used with BBRAM-based AES-GCM keys and incurs a tamper penalty if enough invalid configuration attempts are made (BBRAM key zeroize). The number of allowable configuration attempts is programmable by the user.

Note: The use of obfuscated key storage is not compatible with the configuration counting DPA countermeasure for BBRAM key storage.

2. The encrypted bitstream is authenticated-then-decrypted using asymmetric authentication (RSA) before being fed to the decryptor. Side-channel attacks on the signature verification process are useless because there are no secrets to discover (the public key and public key hash can be known by anyone). This method can be used for eFUSE or BBRAM-based AES-GCM keys and does not incur a tamper penalty. When using this method, symmetric authentication checks are not performed.

For the configuration counting solution (solution 1), there is an associated down counter located in the BBRAM that tracks configuration attempts (the maximum initial count value is 255 and is set by the user at the same time the key is loaded). The counter can be configured to count either invalid configuration attempts or all configuration attempts (whether valid or invalid). The counter is decremented prior to every configuration attempt. If configured to count only invalid configurations, the counter increments after successful configurations. If configured to count all configurations, the counter remains decremented at the end of every configuration.

The BBRAM key is zeroized (tamper penalty) after the counter reaches its terminal count of zero. A smaller initial count value corresponds to a higher level of protection because it reduces the amount of side-channel data that can be collected. Additionally, security checks are also performed to verify counter operation and BBRAM integrity.

For solution 1, there is no way to distinguish between an intentionally random (invalid) bitstream and invalid data that is due to signal integrity issues. This eventually causes the BBRAM key to be zeroized if the FPGA is configured enough times. If using this solution, it is critical that the datapath from the memory device to the FPGA's configuration port be robustly designed to guarantee there are no signal integrity problems.

For the random data solution (solution 2), Figure 4 illustrates at a high-level how authenticate-before-decrypt prevents the decryption of random data.

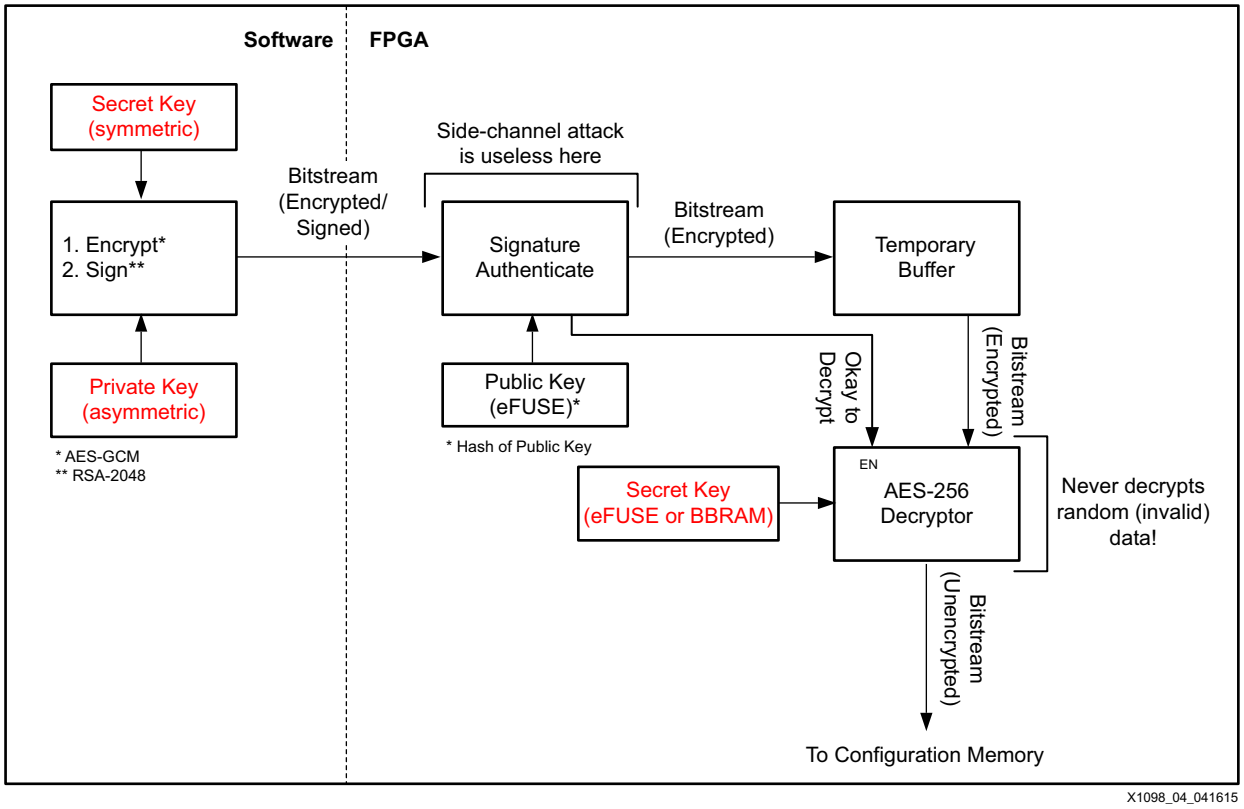

Figure 4: **Authenticate-then-Decrypt to Prevent Random Data Attack**

The temporary buffer shown in Figure 4 is actually the configuration memory normally used to hold the data that defines the user logic. However, in this case it is used to securely hold the encrypted bitstream until after the RSA authentication step is completed. If authentication passes, each frame of encrypted bitstream data is decrypted and then placed back into the same location in configuration memory (this is the read-decrypt-write (RDW) phase). As mentioned in Bitstream Authentication (Asymmetric), this RDW phase operates at the rate of the specified CCLK frequency used for configuration (either the internally specified CCLK frequency or the CCLK frequency that is derived from the EMCCLK).

Configuration Counting Trade-off

Because there is a fixed amount of overhead associated with each of the individual encrypted blocks, the bitstream size grows larger as the blocks are made increasingly smaller when using the configuration counting random data attack solution. This creates a trade-off that must be considered (configuration storage and time vs. security level). When using key rolling with configuration counting, there is always at least an approximate 14% increase in bitstream size. As the block size decreases (becomes more secure), the bitstream size increases and causes a longer configuration time. Figure 5 illustrates how bitstream size grows (y-axis) as the as the number of blocks per key are made smaller (x-axis).

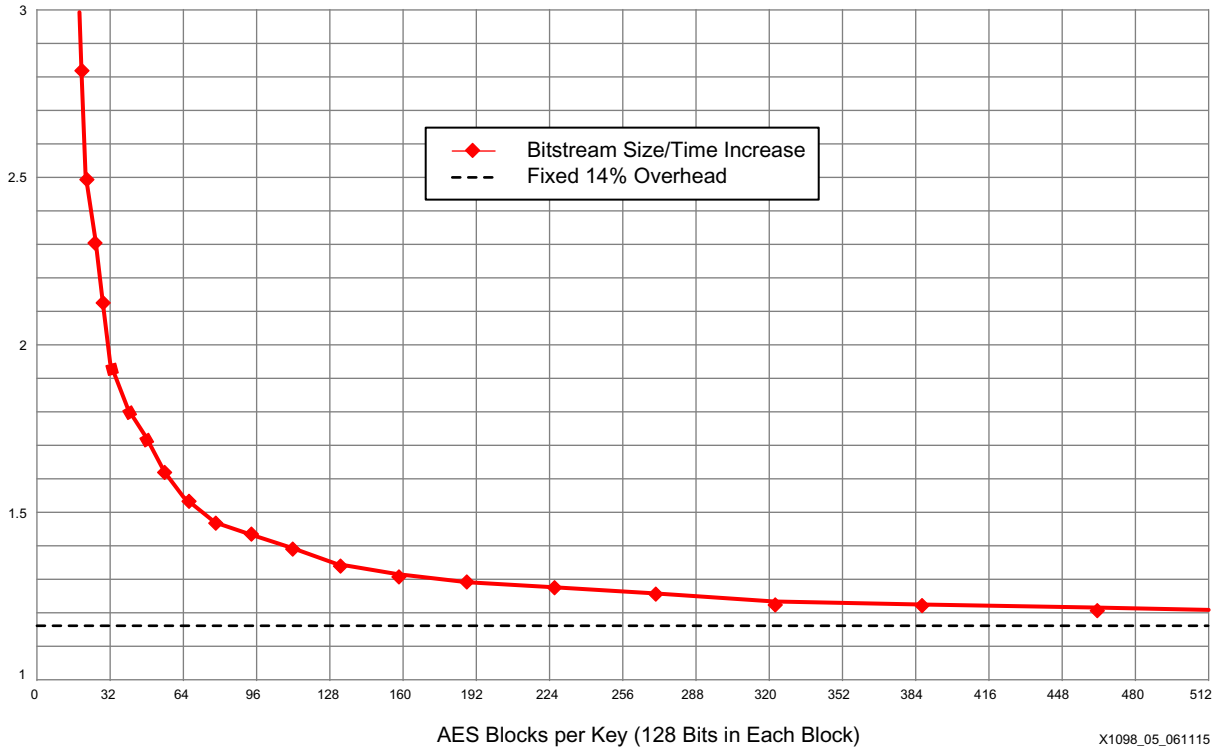

Figure 5: **Key Rolling With Configuration Counting Trade-off**

Authenticate-then-Decrypt Trade-off

Because the entire encrypted bitstream is loaded into the FPGA's configuration memory prior to decryption, the bitstream size is limited by the amount of memory in the device. In this case, what must be considered is the percent impact to overall configuration time. This impact depends on which external port is used for configuration. As mentioned previously, the internal RDW phase is always performed at 32 bits wide and at the rate of the specified clock frequency used for configuration. Therefore, if the external port is slower (i.e., the "bottleneck"), the overall configuration time might not be affected that much. Conversely, if a fast external port is used, there will be a larger overall percent increase in configuration time. Some examples are:

- JTAG and Master SPI (x1):
  Overall configuration time increased by ~8%
  Serial load phase followed by 32-bit parallel RDW phase
- Master SPI Quad (x4):
  Overall configuration time increased by 32%
  Quad SPI load phase followed by 32-bit parallel RDW phase
- Master SPI Dual Quad (x8):
  Overall configuration time increased by 63%
  Dual Quad SPI load phase followed by 32-bit parallel RDW phase
- Master (CCLK) asynchronous BPI (x16):
  Configuration time increased by ~125%
  16-bit parallel load phase followed by a 32-bit parallel RDW phase

- Master (EMCCLK) synchronous BPI (x16):
  Configuration time increased by ~125%
  16-bit parallel load phase followed by a 32-bit parallel RDW phase

Note: All of the above SPI modes (with RSA enabled) are supported in UltraScale+ FPGAs. Consult *UltraScale FPGA RSA Authentication and Supporting Configuration Modes* (XCN15038) [Ref 11] for which SPI modes (with RSA enabled) are supported on UltraScale FPGAs.

IMPORTANT: *Even though the examples towards the bottom of the list above have the greatest overall percent increase in configuration time, they are still a much less overall configuration time than the examples at the top of the list.*

IMPORTANT: *Because the entire encrypted bitstream must fit inside the FPGA's configuration memory prior to decryption, there is a limit as to how small the key rolling blocks are made. In the case of authenticate-then-decrypt, the size of the key rolling block cannot be smaller than 246 AES decryption blocks for UltraScale FPGAs and 186 AES decryption blocks for UltraScale+ FPGAs (each AES decryption block is 128 bits).*

Valid Bitstream Data

Bitstream lengths on modern FPGAs are now large enough that a DPA attack can be attempted on a single valid bitstream load (the valid bitstream data appears random enough with many input changes). To protect against this, bitstreams in UltraScale and UltraScale+ FPGAs can be broken up into multiple smaller blocks, and each block is encrypted using its own unique user-defined key. The size of each block is programmable and depends on your security requirements. Smaller blocks are more secure because less side-channel data can be collected that corresponds to any one key.

To avoid having to store all the decryption keys in on-chip memory (BBRAM or eFUSE), UltraScale and UltraScale+ FPGAs use a *key rolling* technique where only the initial key (key 0) is stored on-chip, while keys for each successive block are encrypted (wrapped) in the previous block. Figure 6 illustrates this concept.

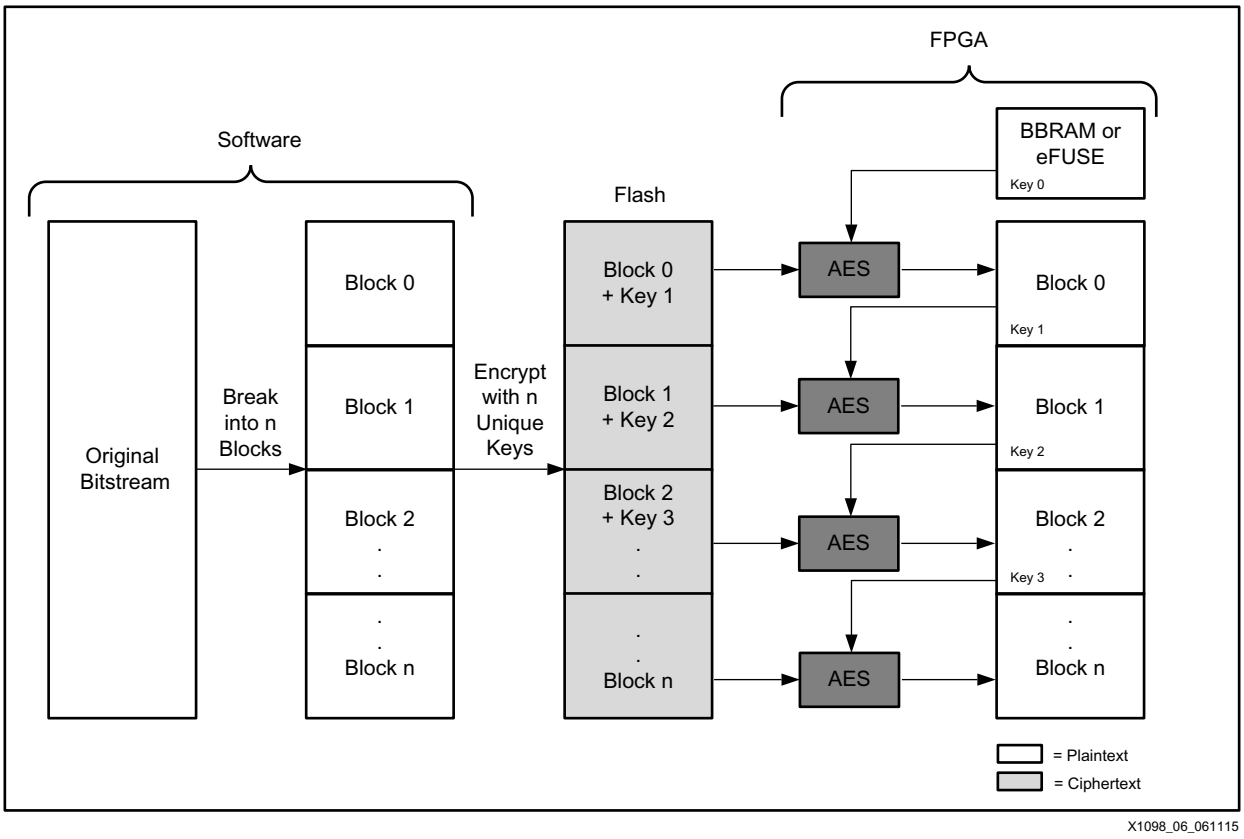

Figure 6: **Key Rolling**

Although it is possible to load a valid bitstream an unlimited number of times, additional configurations do not reveal any new side channel information to the adversary. They can only reduce the signal-to-noise ratio of the side channel data contained in the first configuration. The key rolling method (along with either configuration counting or authenticate-then-decrypt) prevent the adversary from having enough changing values applied into the ciphertext port.

IMPORTANT: *You must use key rolling (valid data attack countermeasure) with either of the random data attack countermeasures (configuration counting or authenticate-then-decrypt).*

There are trade-offs to consider when combining the key rolling techniques with either configuration counting or authenticate-then-decrypt.

DPA Solution Comparison

Only one of the two random data countermeasures (configuration counting or authenticate-then-decrypt) can be enabled on an UltraScale or UltraScale+ FPGA at any one time, so you should understand the trade-offs with each method. Key rolling must always be used with either one of these methods to ensure that a complete set of DPA countermeasures are being used. Table 4 summarizes the trade-offs.

Table 4: **Comparison of Trade-offs in Configuration Counting and Authenticate-then-Decrypt**

Metric	Configuration Counting	Authenticate-then-Decrypt
Supported key storage	BBRAM	BBRAM and eFUSE
Minimum data exposed per key(1)	4 blocks	246 blocks (UltraScale FPGAs)/ 186 blocks (UltraScale+ FPGAs)
Increases bitstream size	Yes	No
Increases configuration time	Based on key life	Based on configuration mode
Requires eFUSE programming	No	Yes(2)
Supports all bitstream features (compression, PR, tandem)	Yes	No
Field key maintenance	Yes	No

Notes:

1. Refer to www.dpacontest.org/home for the current state of the art in open-literature DPA attacks.
2. Required for the hash of the RSA public key.

Hardened Readback Disabling Circuitry

Whenever an encrypted or authenticated bitstream has been loaded into the FPGA, readback of the internal configuration memory cannot be performed by any of the external interfaces (including JTAG). All external readback is automatically blocked (disabled) by hardened triple-redundant logic. The only readback access to the configuration memory after an encrypted bitstream load is via the internal configuration access port (ICAPE3). Because the bitstream is authenticated during the loading process, the ICAPE3 is considered a trusted channel. The ICAPE3 is also considered a trusted channel because it can only be used via a direct connection to the design within the FPGA logic. If the design does not instantiate the ICAPE3, it cannot be used at all.

Note: A Vivado tools security option via a particular control bit in the bitstream provides a soft means of enabling and disabling readback. This bit can be changed during configuration. Therefore, readback disable is easy to defeat for devices that are not using an encrypted or authenticated bitstream. Hardened readback disabling has no such weakness and always overrides the security option when using an encrypted or authenticated bitstream. Xilinx recommends using both the hard and soft methods of disabling readback.

JTAG Port Disable (Passive)

There are two passive ways of disabling the external JTAG port: one is temporary and the other is permanent. The temporary method is to add the set_property BITSTREAM.GENERAL.DISABLE_JTAG YES [current_design] option when using the write_bitstream Tcl command in the Vivado tools. In this case, the external JTAG port becomes disabled after the configuration bitstream is loaded. This method also disables the internal MASTER_JTAG port.

Note: If the Vivado tool option is used and an attacker tries to turn off this option later on, this bit-flip attack is detected by the authentication and the device does not start up. This modification is either detected by the RSA (asymmetric) or AES-GCM (symmetric) algorithms, depending on which authentication methods are enabled.

The Vivado hardware manager can program an eFUSE bit (namely, the FUSE_SHAD_SEC[3] bit in the FUSE_SEC register) to permanently disable the external JTAG port. This method effectively disables the external pins and turns them inside to the FPGA logic. This is effective as an external JTAG disable if authentication is required or if the only time the JTAG disable is required is while the user design is loaded. This would most likely be done as one of the last steps in a production facility so that JTAG-based boundary scan could still be used on the circuit board for testing prior to the programming of this eFUSE.

Active AT Silicon Features

As mentioned in Introduction, the active AT features require you to do something in your FPGA logic design to take advantage of the particular features. For example, you must instantiate the STARTUP primitive in your design to drive the KEYCLEARB input in response to some tamper event. Table 5 summarizes each of these active features, their use cases, and how you can implement the feature.

Table 5: **Active Security Features Use Cases Summary**

Feature	Use Case	User How-to
JTAG port permanent disable (eFUSE)(1)	Permanently prevent unauthorized JTAG access in response to a tamper event.	Instantiate the MASTER_JTAG primitive or internally program the DISABLE_JTAG eFUSE (permanent disable). This also requires forcing authentication (symmetric with aes_efuse_only or symmetric with rsa_auth_all).
JTAG port temporary disable	Prevent an unauthorized JTAG access.	Instantiate the MASTER_JTAG primitive or instantiate the BSCANE2 primitive with the DISABLE_JTAG attribute set to TRUE, or use the BITSTREAM.GENERAL.DISABLE_JTAG option.
JTAG port monitor	Detect unauthorized JTAG access.	Instantiate the BSCANE2 primitive and add a monitoring/response function in the FPGA logic. If MASTER_JTAG is used, you will be detecting activity on the internal JTAG port.
Configuration memory integrity checking	In-the-background check of configuration memory integrity (non-interfering run-time check).	Instantiate the soft error mitigation (SEM) IP core [Ref 12].
Unique identifiers (device DNA and user eFUSE)	Prevent the design from operating (or operate in a limited manner) if unique identifier is not recognized.	Develop FPGA logic to be able to read and process the unique identifier(s) and determine if they are valid.
On-chip temperature and voltage monitor/alarms	Ensure device is operating within normal environmental limits.	Instantiate the system monitor (SYSMONE1) primitive and develop FPGA logic to check and respond to environment status.

Table 5: **Active Security Features Use Cases Summary *(Cont'd)***

Feature	Use Case	User How-to
Uninterruptible internal clock source	Ensure active AT functions cannot be disabled by simply removing an external clock source.	Instantiate STARTUPE3 primitive, connect to the CFGMCLK output, and use that as the clock source for user-defined AT functions. Bitstream encryption must also be used to prevent the clock from being turned off.
External PROGRAM_B intercept	Hold off device configuration to allow data elements that cannot be reset to be cleared (e.g., elements that are not automatically cleared by house-cleaning prior to configuration such as transceiver FIFOs).	Instantiate STARTUPE3 primitive and develop FPGA logic to determine the proper conditions for PROG_ACK assertion after receiving a PROG_REQ.
Configuration memory clearing	Erase the configuration memory in response to a tamper event.	Instantiate ICAPE3 primitive and develop FPGA logic to determine the proper conditions for sending an IPROG command.
Key agility (BBRAM only)(1)	Update the BBRAM key securely in the field without having to return the board or module to a secure facility.	Instantiate the MASTER_JTAG primitive along with logic that can perform a secure key exchange in FPGA logic in response to a key management event.
BBRAM key zeroize (erase + verify)(1)	Zeroize the battery-backed key in response to a tamper event.	Instantiate STARTUPE3 primitive and develop FPGA logic to determine the proper conditions for KEYCLEARB assertion and for reading the verification bit in the STATUS register.
Non-volatile (eFUSE) tamper event logging(1)	Securely log a tamper event in non-volatile memory (eFUSE) for later forensic analysis.	Instantiate the MASTER_JTAG primitive and develop FPGA logic function for logging tamper events in eFUSE bits (POST_CRC must be paused when eFUSE bits are programmed).
Bitstream decryptor permanent disable (eFUSE)(1)	Permanently prevent side-channel analysis of the bitstream decryptor (either as a preventive measure or in response to a tamper event).	Instantiate the MASTER_JTAG primitive and develop FPGA logic to program the decryptor disable eFUSE.
GTS	Shut off outputs in response to a tamper event to prevent any information leakage out of the device.	Instantiate STARTUPE3 primitive and develop FPGA logic to determine the proper conditions for GTS assertion.
GSR	Restore user flip-flop states to initial conditions in response to a tamper event, effectively clearing possible CT from within the device.	Instantiate STARTUPE3 primitive and develop FPGA logic to determine the proper conditions for GSR assertion.

Notes:

1. This feature is new or improved in UltraScale and UltraScale+ FPGAs.

JTAG Port Disable (Active)

An attacker often starts at the external JTAG port when trying to break into a system. With UltraScale and UltraScale+ FPGAs, there are a number of active methods to block the JTAG port that can be done both temporarily and permanently. Like in previous FPGA families, the first method uses a single BSCANE2 primitive that is instantiated with the DISABLE_JTAG attribute set to TRUE. This breaks the JTAG chain (both the external JTAG port and the internal MASTER_JTAG port) and any other devices in the same chain as the FPGA device, either upstream or downstream. Here is an example VHDL instantiation [Ref 13] for an UltraScale or UltraScale+ FPGA design:

```
BSCAN_U0 : BSCANE2
 generic map (
   DISABLE_JTAG => TRUE,
   JTAG_CHAIN   => 1 -- can be 1, 2, 3, or 4 depending on chain location
 )
 port map (
   CAPTURE => open,
   DRCK    => open,
   RESET   => open,
   RUNTEST => open,
   SEL     => open,
   SHIFT   => open,
   TCK     => tck_signal,
   TDI     => tdi_signal, TMS     => tms_signal, UPDATE  => open,
   TDO     => '1'
 );
```

Note: The TCK, TDI, and TMS signals are the only ports connected above because they can be used to monitor for any JTAG activity (this is explained in JTAG Monitoring (Detection)).

The second method to block the external JTAG port is to instantiate a new primitive for the UltraScale and UltraScale+ FPGAs named MASTER_JTAG. This primitive can be used to override the external JTAG pins of the FPGA allowing full access to the JTAG port from within the device. This method effectively disables the external pins and turn them inside to the FPGA logic. This is effective as an external JTAG disable if authentication is required or if the only time the JTAG disable is required is while the user design is loaded. Using this method, the user design now has full JTAG control but any activity on the external JTAG pins cannot be monitored. Because MASTER_JTAG is accessible by the user design, in response to a tamper event it could be used to program an eFUSE bit that permanently disables the external JTAG port. Contact your local Xilinx FAE for additional details on how to program eFUSE bits from within the device via MASTER_JTAG.

If there are any JTAG-based debugging tools in your FPGA logic design (which are connected to the dedicated external JTAG port), breaking the JTAG chain does not allow them to function. During the FPGA debug phase, the JTAG chain can be left intact and then broken later on in the development cycle when the JTAG-based debugger tool is no longer required.

JTAG configuration of the device cannot be used when the option to break the JTAG chain is enabled. You must choose one of the other configuration interfaces such as serial, serial peripheral interface (SPI), byte parallel interface (BPI), and SelectMAP [Ref 3]. JTAG boundary-scan board-level tests operate normally as long as the actual configuration of the FPGA device is delayed.

JTAG Monitoring (Detection)

To detect any JTAG activity from within the device, you need to monitor any combination of the JTAG TCK, TDI, or TMS line(s) on a BSCANE2 primitive. Because any external JTAG command requires these lines to toggle, activity detectors on any or all of these lines can catch JTAG activity. For example, to monitor any rising edges on the TDI line, the circuit in Figure 7 could be used.

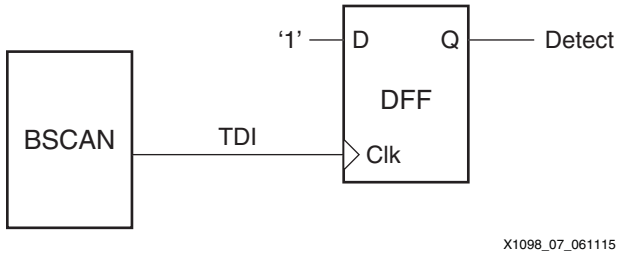

Figure 7: **Example JTAG Activity Detector**

Any rising edge on the TDI line gets latched, and the detect output of the DFF remains at 1 until the part is reconfigured (i.e., until the PROGRAM_B input is asserted). This method can be extended to monitor rising or falling edges on any of the TCK, TDI, or TMS signal lines. When any of the outputs of the JTAG detector DFFs are set, they can be used to initiate a tamper penalty.

Configuration Memory Integrity Checking (Detection)

Corruption of any of the internal configuration memory cells (which are configured by the decrypted bitstream) could cause the FPGA to operate in an unknown or undesired manner. The corruption could occur by an intentional post configuration tamper attack or by an unintentional event such as a single-event upset (SEU). By using the SEM IP core [Ref 12], continuous readback of configuration data in the background of a design is performed to detect any bit flips. The SEM IP core can also perform SEU corrections.

Unique Identifiers (Detection)

Two types of unique identifiers (UIs) are available for use: device DNA and user eFUSE. These UIs can be used as anti-cloning security measures (i.e., someone steals the user's bitstream and uses it to program their own devices) or for enabling or disabling certain features (upgrade or downgrade) depending on the value of the UI.

Device DNA consists of a 96-bit device-specific serial number and is set by Xilinx in one-time programmable (OTP) eFUSE bits on the FPGA during the manufacturing flow (FPGA logic read access to the value is via the DNA_PORTE2 primitive, or it can be read externally via JTAG). User eFUSE provides 32 bits of user read/write OTP area and is set by the user via JTAG (FPGA logic read access to the value is via the EFUSE_USR primitive). Both of these UIs can be used separately or in conjunction for security purposes.

Note: Use of device DNA or user eFUSE provides for unique IDs, but does not provide cryptographically strong confidentiality or authentication (such as AES-GCM). AES-GCM encryption is the preferred

method of providing anti-cloning protection. However, by taking advantage of these UIs, you can add another layer to the overall AT scheme.

These UIs can be used to link the bitstream to one particular device (in the case of device DNA, or multiple devices in the case of user eFUSE). The UI comparison is put into the FPGA design by the user and the results of this comparison can be used to gate FPGA activity. For example, if the UI comparison fails, the design can refuse to function or function with limited capability. An example use case of the UIs is as follows:

1. Setup: Read the UI value(s) from the FPGA via JTAG, generate a hash from the UI value(s), and store in a flash device accessible to the FPGA using a robust one-way function (a keyed type is the most secure) as shown in Figure 8.

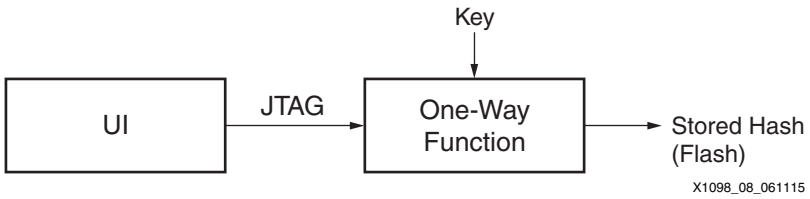

Figure 8: **Encrypt the DNA Value with a Confidential Key**

   The key for the one-way function shown in Figure 8 could be stored within the encrypted bitstream. If bitstream encryption is not used, this approach relies on the complexity of the bitstream to keep the key confidential.

2. Configure the FPGA.
3. Compare: The FPGA design reads the UI value from either or both the DNA_PORT and EFUSE_USR primitive(s) and then calculates the checksum using the same algorithm. The design then compares the calculated hash with the hash read from flash. If the hash passes, the design is allowed to become active.

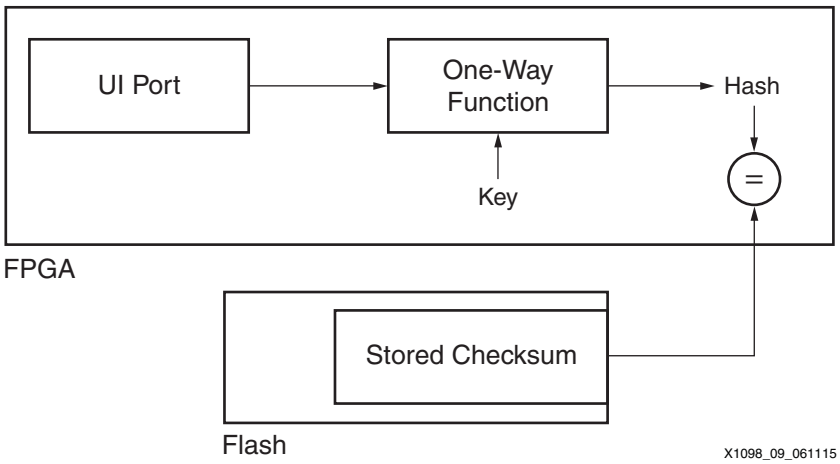

Figure 9: **Hash Comparison**

For additional information on these UIs, consult the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3]. *Security Solutions Using Spartan-3 Generation FPGAs* (WP266) [Ref 14] also talks about device DNA operation for the Spartan-3 generation FPGAs.

On-Chip Temperature and Voltage Monitors/Alarms (Detection/Response)

By modifying the normal operating voltages and/or temperature of an FPGA, an attacker might attempt to cause the device to behave in an unintended way, such as to extract data from it or cause it to bypass certain security features. For example, the Federal Information Processing Standards Publication (FIPS) 140-2 *Security Requirements For Cryptographic Modules* [Ref 15] states: "In particular, the cryptographic module shall monitor and correctly respond to fluctuations in the operating temperature and voltage outside of the specified normal operating ranges."

To help meet this type of requirement, on-chip dedicated IP blocks can be used, namely, the UltraScale and UltraScale+ FPGA system monitor (SYSMONE1) [Ref 16]. The SYSMONE1 is a multi-channel ADC within the FPGA and can be used to monitor on-chip power supply voltages, a number of I/O bank voltages, on-chip die temperature, and a number of user analog voltages fed into the FPGA's external pins. The SYSMONE1 can be easily instantiated in your FPGA logic design. This type of on-chip monitoring is more secure than off-chip monitoring because it is harder to tamper with. You also have the option of using an external or internal voltage reference (V_{REF}) for SYSMONE1. Although the external reference is more accurate, the internal reference is more secure because it is much more difficult for an adversary to tamper with. The selection between the internal and external reference is controlled by an external pin. You should thus read the appropriate SYSMONE2 status register to confirm that the internal V_{REF} is being used [Ref 16].

Upper and lower alarm limits can be programmed directly into SYSMONE1 for the on-chip parameters. Additional FPGA logic can be used to create alarm limits for external voltage inputs (e.g., an external analog voltage tamper loop or the output of a pressure sensor). The status of the alarm signals can be used by your design or system to determine the appropriate course of action in case they become active (i.e., determine the appropriate tamper penalty). The analog inputs are bandwidth limited (see *UltraScale Architecture System Monitor User Guide* (UG580) [Ref 16] for the maximum input frequencies).

If detection of very fast changes in temperature or voltages is required, an off-chip solution might be required. It is up to you to define what the required detection bandwidth needs to be. *The Sorcerer's Apprentice Guide to Fault Attacks* [Ref 17] describes a number of methods to mount an attack on a chip, one of them being variations in the power supply voltage.

Uninterruptible Internal Clock Source (Detection)

When using bitstream encryption, you can take advantage of an uninterruptible clock source named CFGMCLK (configuration internal oscillator clock output) located as an output on the STARTUPE3 block primitive (refer to the example VHDL instantiation of the STARTUPE3 block in PROGRAM_B intercept). Because this clock is always active, it can be used as the basis for a user clock (or other critical user signal) monitoring function. Even though CFGMCLK can vary quite a bit from its nominal value of 50 MHz ± 15% [Ref 18] [Ref 19], it can still be quite useful in a monitoring function to make sure a critical user clock or signal is still "alive" and toggling between a lower and upper frequency range (which takes into account the CFGMCLK variation). If the critical user clock or signal falls out of this range, it can indicate either that the design has malfunctioned or is being tampered with, and the appropriate penalty could be asserted.

CFGMCLK can be also used as the clock source for any other user-defined AT functions in FPGA logic. It is important that AT functions cannot be halted by simply removing an external clock source.

External PROGRAM_B Intercept (Prevention and Detection)

Not all memory elements in the FPGA are cleared upon configuration. For example, there might be GTH and GTY transceivers with FIFOs in use that retain state even after the external PROGRAM_B pin is asserted (assertion of PROGRAM_B causes the FPGA to reset and become reconfigured via the bitstream). An attacker could potentially assert the PROGRAM_B pin and replace the user bitstream with their own bitstream, one that is designed to dump out the contents of the uncleared memory elements after the FPGA is configured. By using the PROGRAM_B intercept feature such as the PROGRAM_B request/acknowledge pair on the STARTUP block PREQ/PACK, you can indefinitely delay the reconfiguration of the FPGA so that these memory elements can first be cleared by your design or any other housekeeping tasks can be completed before allowing a PROGRAM_B to happen.

Another use case for PROGRAM_B intercept could be a system (when fielded) that should never see PROGRAM_B assertion while active—its mere presence signifies a tamper event. One of the first things an attacker might do is to assert the PROGRAM_B to observe the start-up behavior of the FPGA. If PREQ goes active in your design, it could invoke a penalty and then allow the PROGRAM_B to occur by asserting PACK.

Whenever an encrypted bitstream with the PROGRAM_B intercept security feature enabled has been loaded and the PROGRAM_B pin is asserted externally (or an internal IPROG command is sent to the ICAPE3), the PREQ output is asserted High on the STARTUPE3 block. Configuration is held off indefinitely until you drive the PACK input High (rising-edge sensitive) or until the device is power cycled.

The following is an example VHDL instantiation of the STARTUPE3 block with the correct security generic set and the PREQ/PACK signal connections:

```
STARTUP_U0 : STARTUPE3
  generic map (
    PROG_USR => TRUE ) -- turn on PROGRAM_B intercept security feature
  port map (
    CFGCLK    => open,
    CFGMCLK   => cfgmclk_signal,
    DI        => open,
    EOS       => open,
    PREQ      => preq_signal, -- PROGRAM request to FPGA logic output
    DO        => (others => '0'),
    DTS       => (others => '0'),
    FCSBO     => '0',
    FCSBTS    => '0',
    GSR       => gsr_signal,
    GTS       => gts_signal,
    KEYCLEARB => keyclearb_signal,
    PACK      => pack_signal, -- PROGRAM acknowledge input (rising edge)
    USRCCLKO  => '0',
    USRCCLKTS => '0',
    USRDONEO  => '0',
    USRDONETS => '0'
);
```

Note: The PROGRAM_B intercept intercepts *all* program attempts, not just external. It intercepts IPROG and JPROGRAM from MASTER_JTAG as well.

Configuration Memory Clearing (Response/Penalty)

IPROG is an internal command sent through the ICAPE3 interface that clears the FPGA configuration memory, all flip-flop contents, and key expansion memory, but does not clear the key itself. IPROG is equivalent to the assertion of the external PROGRAM_B pin. This command effectively clears configuration memory (configuration data, block RAMs, and flip-flop states).

After both of the KEYCLEARB and IPROG penalties are invoked, the FPGA becomes inoperable because the existing bitstream can no longer be decrypted by the FPGA. The fact that device configuration is no longer possible with the encrypted bitstream is an indication that a tamper event has occurred. At this point, your design might choose to load in an unencrypted bitstream after the KEYCLEARB and IPROG penalties so that there is some basic functionality without exposing any of the CT. Of course, an unencrypted bitstream cannot be loaded if using an eFUSE-based key and the cfg_aes_only eFUSE is also programmed.

To send an IPROG command to the configuration engine, the ICAPE3 primitive must be instantiated in your design and the appropriate sequence of commands must be written to the ICAPE3. For more information, refer to the IPROG Reconfiguration sections in the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3].

Key Agility (Response)

Key agility refers to the ability to update or change the AES bitstream decryption key in BBRAM via the FPGA logic. This does not apply to eFUSE-based keys.

In Figure 10, an UltraScale/UltraScale+ FPGA is shown with its initial key already loaded into BBRAM. You can then load in an external encrypted bitstream, decrypt it, and then have the initial FPGA logic design up and running.

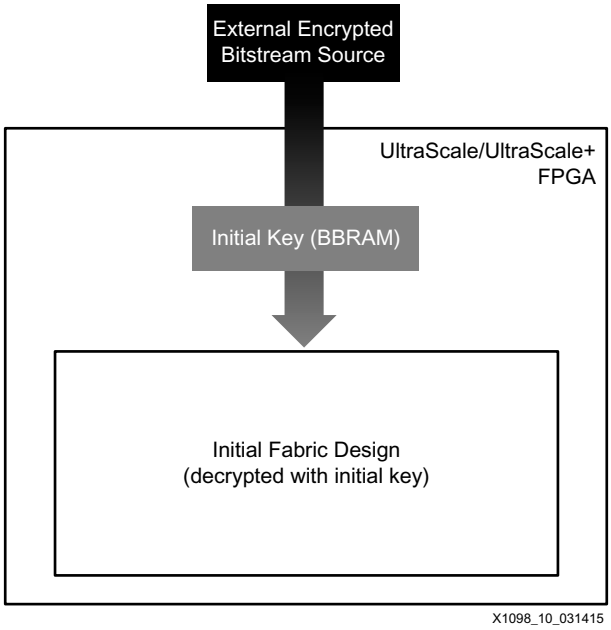

Figure 10: **UltraScale/UltraScale+ FPGA with Key Loaded into BBRAM**

At some time in the future, a key management event occurs, which means that the key requires to be changed or updated due to a security breach, good cryptographic practice, or new design. By using some sort of secure key exchange algorithm (which would be an IP core running in FPGA logic, such as the Diffie-Hellman protocol) you can bring in an external encrypted key, perhaps even via the Internet. Then, you can decrypt this key with the IP core, load it into BBRAM, and perform the integrity check internally via the MASTER_JTAG primitive (Figure 11).

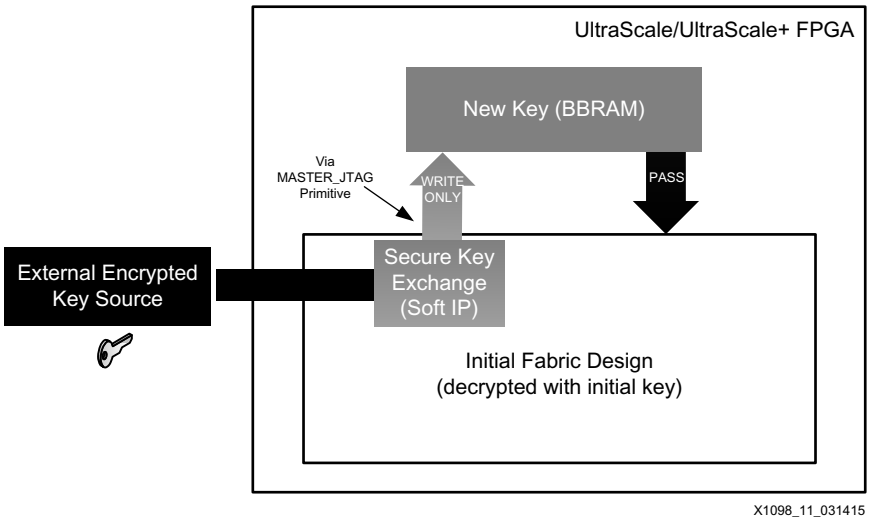

Figure 11: **Key Management Event**

You can then load in an external encrypted bitstream (encrypted on the new key), decrypt it, and have a new fabric design that was decrypted with this new key as shown in Figure 12.

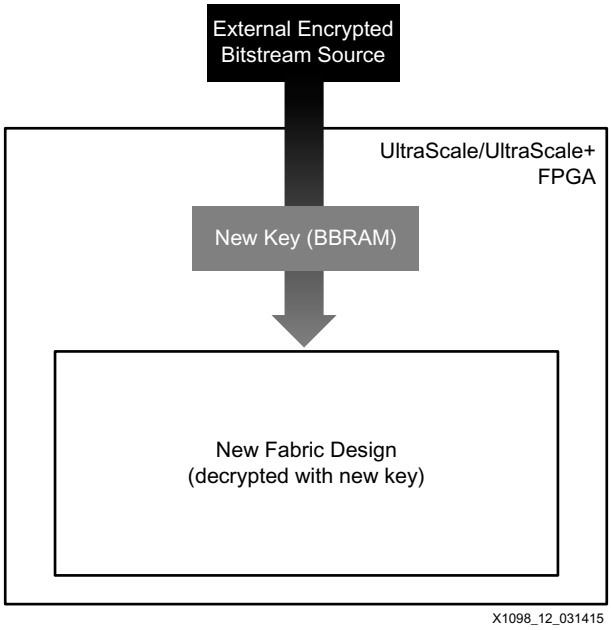

Figure 12: **New Fabric Design**

Using the above scheme, it is possible to update the key in the field without having to bring the board or the system all the way back to a secure facility.

BBRAM Key Zeroize (Response/Penalty)

The crown jewel of the FPGA is the AES-GCM key used to decrypt the bitstream. After an attacker has access to the key, the contents of the original bitstream can be easily determined. By connecting your design to the KEYCLEARB(1) input (located on the STARTUP block primitive in External PROGRAM_B Intercept (Prevention and Detection)), you can choose to assert this signal as a penalty in response to internal (or external) tamper events. By zeroizing the 256-bit key in BBRAM, the registered content of the eFUSE key, and the 1920-bit expanded key in the decryption block, the encrypted bitstream stored in off-chip non-volatile memory becomes useless to the attacker. For UltraScale and UltraScale+ FPGAs, a configuration status bit(2) is also available that indicates whether the key data erasure has been verified (proves zeroization).

The decision for the KEYCLEARB assertion (as well as all other tamper responses) does not necessarily need to originate from within the FPGA itself. It could be due to a tamper event somewhere else in the system (e.g., a breach of a system-level or module-level tamper boundary).

After the key is zeroized, the FPGA device is useless until reprogrammed with the same key or reconfigured using IPROG [Ref 3] to an unencrypted bitstream (perhaps with reduced functionality). Ensure that the KEYCLEARB input is only asserted under the proper conditions. In many cases, equipment must be taken out of the field and sent back to a central depot or manufacturing facility to be re-enabled with a key load operation.

1. The KEYCLEARB signal has no effect on the non-volatile eFUSE key.
2. This is bit 21 of the ICAP accessible status register

KEYCLEARB can also be combined with the IPROG command (described in Configuration Memory Clearing (Response/Penalty)) in response to tampering to also erase the configuration memory. The KEYCLEARB assertion must occur before an IPROG command is sent (i.e., the IPROG command can be sent to the ICAPE3 immediately following the assertion of KEYCLEARB).

Non-Volatile (eFUSE) Tamper Event Logging (Response)

The UltraScale and UltraScale+ FPGAs include a new 128-bit USER eFUSE register. This register can be used to flexibly meet your needs for a non-volatile area. The eFUSE register bits can only be programmed and read back using JTAG instructions (see the *UltraScale Architecture Configuration User Guide* (UG570) [Ref 3] for more information). These eFUSE bits can be programmed via the external JTAG port or the internal MASTER_JTAG. Contact your local Xilinx FAE for additional details on how to program eFUSE bits from within the device via MASTER_JTAG.

IMPORTANT: *After these eFUSE bits are programmed, it is possible to program another eFUSE (a write disable) that prevents any more programming of the eFUSE register bits (i.e., to "lock the door"). Thus, an adversary cannot attempt to overwrite the tamper log information.*

Bitstream Decryptor Permanent (eFUSE) Disable (Prevention/Response/Penalty)

To prevent an attacker from collecting any side-channel information, an eFUSE can be programmed that permanently disables the AES-GCM decryptor. This eFUSE can be programmed via the external JTAG port or the internal MASTER_JTAG. Contact your local Xilinx FAE for additional details on how to program eFUSE bits from within the device via MASTER_JTAG. This feature can be used as a tamper response or to simply create a one-time encrypted configurable device. If the cfg_aes_only eFUSE bit is also programmed, it prevents the device from getting configured again (i.e., "brick" the device).

Global 3-State (Response/Penalty)

Depending on the system design, critical (red) information might flow out of the external FPGA pins (e.g., a cryptographic module). Asserting the GTS input on the STARTUPE3 block (see the sample code in External PROGRAM_B Intercept (Prevention and Detection)) in response to a tamper event causes all FPGA outputs to immediately enter a high-Z state and prevent any more data from flowing outside the FPGA. This could be an immediate step to take just prior to an IPROG or KEYCLEARB to make sure red data flow is halted as soon as possible.

Global Reset (Response/Penalty)

Critical data or sensitive parameters can be stored in FPGA logic registers such as a user key (not the AES BBRAM bitstream decryption key). Asserting the GSR input on the STARTUP block (see External PROGRAM_B Intercept (Prevention and Detection)) in response to a tamper event causes all FPGA registers (i.e., flip-flops) to be restored to their default state. This could be an immediate step to take along with KEYCLEARB to make sure all sensitive data in the FPGA is

erased as soon as possible. The GSR does not impact the shift register look-up table (SRL) or block RAM contents. These must be cleared by your design or by an IPROG command.

Tamper Resistance Guidance

This section provides guidance and technical tips that can be used in conjunction with the previously addressed built-in silicon AT features to create tamper-resistant designs using UltraScale and UltraScale+ FPGAs.

Load CT Only When Needed (Prevention)

If the design can be partitioned into sections that contain non-critical and critical technology blocks, it might be possible to only have the non-CT portion of the design resident at all times and use partial reconfiguration (PR) features [Ref 20] of the FPGA to allow the CT to be loaded only when needed. The CT can then be erased by loading in a black box version of the PR region when it has completed its tasks. The partial bitstream for the CT can be decrypted by your algorithm of choice in the FPGA logic. In response to a tamper event, both the PR region and the key for the CT (perhaps stored in block RAM or FPGA logic registers) could be erased.

For example, Figure 13 and Figure 14 illustrate a general system with an FPGA, CPU, and external memory devices (for FPGA configuration, PR, CPU code, and CPU boot code). In Figure 13, the PR region (named user CT logic region) is empty.

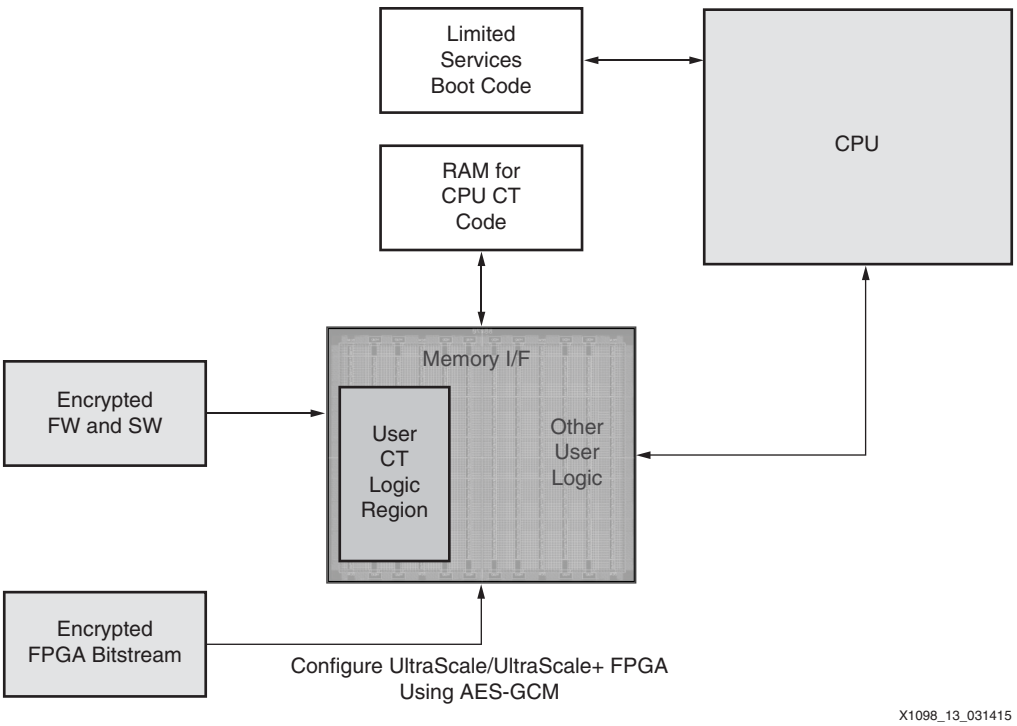

Figure 13: **Use Model: Protecting System CT**

In Figure 14, the PR region on the FPGA has been dynamically loaded with CT logic via PR (named user CT logic). When the CT function is completed or a tamper event occurs, the PR

region can be returned to the state shown in Figure 13 by loading in a black box version of the PR module.

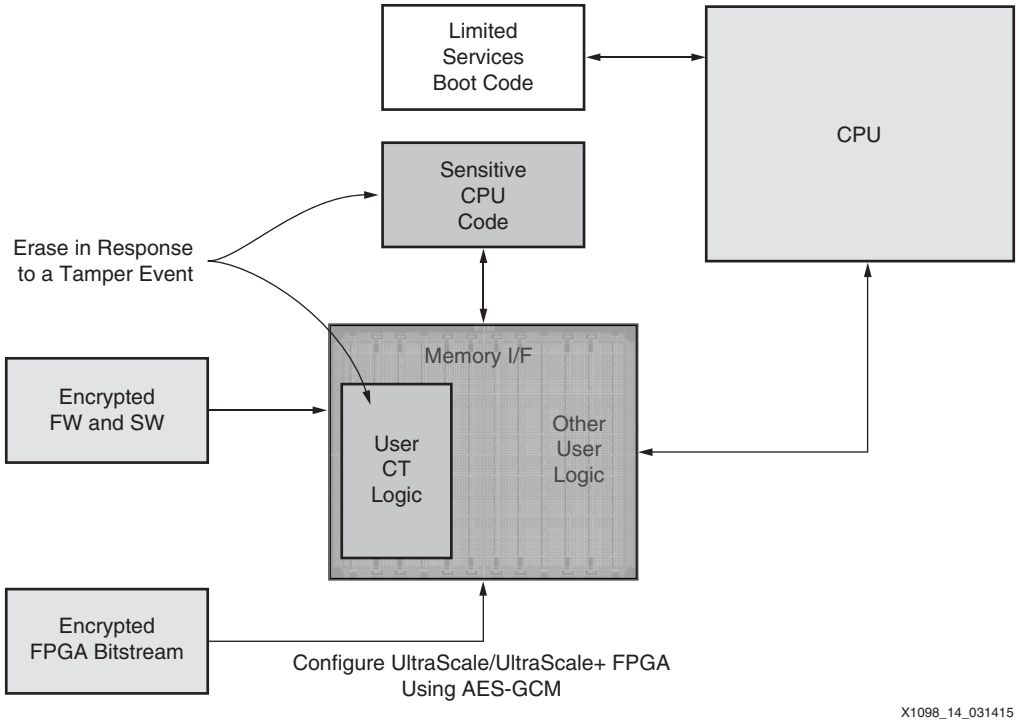

Figure 14: **Use Model: Protecting System CT–Tamper Response**

Note: If a tamper event occurs, the external sensitive CPU code memory could be erased so that only encrypted, cleared, or non-sensitive external memory contents remain.

In these examples, the ICAPE3 is used to perform the PR. Because the ICAPE3 is a trusted channel, it allows either encrypted or unencrypted PR bitstreams (even if an eFUSE key is being used with the cfg_aes_only eFUSE bit blown). An encrypted and authenticated PR bitstream is always recommended where the decryption and authentication take place with your decryptor authentication engine of choice in the FPGA logic. A useful reference on secure PR is *PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration* (XAPP887) [Ref 20].

Key Erase via External Shunt

Another method to erase the BBRAM key is via an external shunt to ground on the V_{BATT} line. This method can be used to erase the key when main power (V_{CCINT} and V_{CCAUX}) to the FPGA is not applied because active features, such as KEYCLEARB, can be used only after the FPGA is powered up and configured. For instance, if a system-level tamper event is detected prior to the FPGA having its main power applied (perhaps a tamper switch becomes activated), the external battery power line to the FPGA's V_{BATT} pin can be opened and the V_{BATT} pin driven to ground with some sort of transistor shunt. Care must be taken that the circuit is designed to open the battery connection before shunting the V_{BATT} pin to ground. Another option is to connect the battery to the V_{BATT} pin via a resistor. (The V_{BATT} pin maximum input current is 150 nA.) By choosing the appropriate resistance value, the V_{BATT} pin can be shunted to ground directly without causing excessive current flow out of the battery.

If an FPGA is not powered up (no V_{CCINT}, V_{CCAUX}, or other voltages except for V_{BATT}), it would take a worst-case maximum time of 50 ms for the AES key stored in BBRAM to become erased if a user was to properly shunt the V_{BATT} pin to ground at –55°C.

IMPORTANT: *For improved security and to keep leakage as small as possible, use a battery whose voltage is as low as possible. For more information on battery voltage levels, refer to the UltraScale and UltraScale+ architecture data sheets* [Ref 18] [Ref 19] [Ref 21] [Ref 22].

Preemptive BBRAM Key Zeroize

Another use case for key zeroize could be a preemptive measure. After loading and decrypting the bitstream, the BBRAM key can be purposely zeroized before sending the system out into the field. Of course, this would only work for a system that is not intended to be power cycled after it becomes deployed (for example, a missile after being launched). This could also be used for an eFUSE key by writing over it with all ones via MASTER_JTAG before deployment as long as eFUSE key reading and writing has not yet been disabled.

Avoiding Weak or Duplicate Keys

All zeros, all ones, or repetitive patterns should never be used in user keys. Keys should not be reused if at all possible. Personnel access to the key values should be tightly controlled (i.e., only those with a need to know should have access to key data). Ideally, a random source should be used to create the keys. Avoid using weak keys. (For example, an all zeros random key is theoretically possible.) The Vivado software can automatically generate the AES-GCM keys. However, it uses a pseudorandom process seeded with the current date and time. The most secure keying material comes from a truly random process.

Key management is a very important element (and probably the most complex) in any cryptographic system. For additional help on this subject, the NIST Key Management Guideline [Ref 23] is a useful reference.

Sending Tamper Status Outputs to System

Upon a tamper event (in addition to asserting a penalty) your design could send tamper status information back to the system (instead of or in addition to logging it locally in user eFUSE space). The system could then store this information away for future auditing purposes. It would have to be designed to transmit the data before an IPROG command (tamper penalty) is given.

Restricting Access to FPGA Probe Points

Making it difficult for an attacker to get near the FPGA is a good example of a layered approach. A robust tamper boundary (perhaps with tamper detect switches) can be used around any device(s) that might contain CT. For example, an activated tamper switch could cause a shunt to go active on the FPGA's V_{BATT} line. Buried vias and routing can be used on the printed circuit board for FPGA signals, power supply routing can be kept within buried layers (and difficult to get to), and adequate decoupling can be used (buried capacitance technology should be used,

if possible). JTAG boundary-scan techniques can be relied on for board-level factory testing, and test points should be removed from production boards. For more information, refer to the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 24] and the *UltraFast Design Methodology Checklist* (XTP301) [Ref 25].

Conclusion

This application note summarizes the AT features currently available in UltraScale and UltraScale+ FPGAs and gives practical examples of how to use them effectively. By taking advantage of these features and following the AT guidance early on in the design cycle, a tamper-resistant FPGA-enabled system design can be realized.

No single AT feature or technique is 100% effective all of the time or can meet all the AT needs for the entire system. However, making the adversary's job as difficult and expensive as possible and following a multi-layered approach almost always yields very good results.

The tools and technologies for the development and testing of new integrated circuits including FPGAs are always evolving and improving. In parallel, the tools used by the adversary also evolve and improve, so it is important to be aware of what AT features and techniques are available for use. Additionally, Xilinx is committed to staying abreast of these developments to enhance and develop new features to protect customer IP now and into the future.

References

1. Xilinx Getting Started
   www.xilinx.com/company/gettingstarted/index.htm
2. Xilinx Vivado Design Suite
   www.xilinx.com/products/design-tools/vivado.html
3. *UltraScale Architecture Configuration User Guide* (UG570)
4. *Solving Today's Design Security Concerns* (WP365)
5. *Developing Tamper Resistant Designs with Xilinx Virtex-6 and 7 Series FPGAs* (XAPP1084)
6. Security Monitor Product Brief
   www.xilinx.com/publications/prod_mktg/CS1140_AD_SecMonIP_ProdBrf_Update_June_2012.pdf
7. *Vivado Design Suite User Guide: Programming and Debugging* (UG908)
8. Introduction to Indirect Programming – SPI or BPI Flash Memory
   www.xilinx.com/support/documentation/sw_manuals/xilinx11/pim_c_introduction_indirect_programming.htm
9. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
10. *Using Encryption to Secure an UltraScale FPGA Bitstream* (XAPP1267)
11. *UltraScale FPGA RSA Authentication and Supporting Configuration Modes* (XCN15038)

12. Soft Error Mitigation (SEM) Core
    www.xilinx.com/products/intellectual-property/sem.html
13. *UltraScale Architecture Libraries Guide* (UG974)
14. *Security Solutions Using Spartan-3 Generation FPGAs* (WP266)
15. *Security Requirements for Cryptographic Modules*, FIPS PUB 140-2
    www.nist.gov/itl/upload/fips1402.pdf
16. *UltraScale Architecture System Monitor User Guide* (UG580)
17. Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. *The Sorcerer's Apprentice Guide to Fault Attacks*.
    http://eprint.iacr.org/2004/100.pdf
18. *Kintex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS892)
19. *Virtex UltraScale FPGAs Data Sheet: DC and AC Switching Characteristics* (DS893)
20. *PRC/EPRC: Data Integrity and Security Controller for Partial Reconfiguration* (XAPP887)
21. *Kintex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics* (DS922)
22. *Virtex UltraScale+ FPGAs Data Sheet: DC and AC Switching Characteristics* (DS923)
23. NIST Key Management Guideline
    csrc.nist.gov/groups/ST/toolkit/key_management.html
24. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
25. *UltraFast Design Methodology Checklist* (XTP301)
26. Partial Reconfiguration in the Vivado Design Suite
    www.xilinx.com/products/design-tools/vivado/implementation/partial-reconfiguration.html
27. Xilinx ChipScope Pro and the Serial I/O Toolkit
    www.xilinx.com/tools/cspro.htm
28. *Secure Hash Standard*, FIPS PUB 180-2
    csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
29. Xilinx Partial Reconfiguration
    www.xilinx.com/tools/partial-reconfiguration.htm
30. *In-System Programming Using an Embedded Microcontroller* (XAPP058)
31. *Embedded JTAG ACE Player* (XAPP424)
32. Google differential+power+analysis search results
    scholar.google.com/scholar?q=Differential+power+analysis&hl=en&as_sdt=0&as_vis=1&oi=scholart

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
02/22/2017	1.2	Initial Xilinx release.

Please Read: Important Legal Notices