National Security Agency
Cybersecurity Technical Report

DoD Microelectronics: Field Programmable Gate Array Security Guidance

JAN 2025

U/OO/100236-25
PP-24-4776
Version 1.0

Notices and History

Document Change History

Date	Version	Description
January 2025	1.0	Initial Publication

Disclaimer of warranties and endorsement

The information and opinions contained in this document are provided "as is" and without any warranties or guarantees. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the United States Government, and this guidance shall not be used for advertising or product endorsement purposes.

Publication information

Author(s)

National Security Agency
Cybersecurity Directorate
Joint Federated Assurance Center

Contact information

Joint Federated Assurance Center: JFAC_HWA@radium.ncsc.mil
General Cybersecurity Report Inquiries: CybersecurityReports@nsa.gov
Defense Industrial Base Inquiries and Cybersecurity Services: DIB_Defense@cyber.nsa.gov
Media inquiries / Press Desk: Media Relations, 443-634-0721, MediaRelations@nsa.gov

Purpose

This document was developed in furtherance of NSA's cybersecurity missions. This includes its responsibilities to identify and disseminate threats to National Security Systems and Department of Defense information systems, and to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

Executive summary

This document discusses the security features present in commercial Field Programmable Gate Array (FPGA) devices and provides recommendations to mitigate common security threats. These topics are covered in three sections:

- **Design security concerns in FPGAs** – Discusses the FPGA design concerns being addressed in the document and how it differs from existing hardware assurance documents.
- **FPGA security features overview** – Describes general non-vendor specific security features offered in commercial FPGA devices. Each security feature listed includes a discussion of tradeoffs to consider and NSA recommended guidance when selecting and implementing the security feature.
- **Security subjects and methods overview** – Describes security processes to consider during the lifecycle of the fielded device. It includes a tradeoff discussion when applicable, as well as NSA recommended guidance.

This guidance is intended to inform readers about FPGA security features, discuss tradeoffs, and provide recommendations associated with their use.

Table of Contents

1. Introduction

This generic field programmable gate array (FPGA) security implementation guide offers advice for using FPGA vendor provided security features to mitigate the risk of adversarial compromise of devices. This guidance is brand agnostic and can be applied to any FPGA product. This document describes security concerns, general security features available on FPGAs, and recommended security procedures related to FPGA use.

2. Design security concerns in FPGAs

FPGAs are readily available for adversary study, reprogrammable, and are used in both DoD and non-DoD systems, making them a potential target for adversarial attacks. For this reason, FPGA users should understand the risks and the available best practices to mitigate these risks. Risk mitigation falls under both the categories of security and hardware assurance. While there is some overlap to these categories, they refer to different aspects of protection from attacks. Hardware assurance (HwA) generally refers to protecting the design and manufacturing phases of the hardware from adversarial influence. In this document, security refers to device features used to safeguard sensitive data associated with the FPGA configuration, protect the data it contains, and ensure its correct operation. This document focuses on security, specifically the following four types of security threats:

- An adversary stealing confidential data, including user application configuration data, information about security features in use, and dynamic operational data being processed by the application.
- An adversary modifying or replacing the configuration data for the purpose of sabotage or subversion.
- An adversary modifying the FPGA security settings, such as cryptographic functions, tamper detection, and access denial on pins. These modifications can weaken a device’s security profile.
- An adversary stealing cryptographic keys.

2.1 Scope

NSA is providing this security guidance to assist the reader in making the best use of the security features provided by an FPGA vendor to mitigate security risks. This guidance describes common security features recommended for use with FPGA-based DoD systems. This document does not address hardware assurance concerns in the design or manufacturing stages of FPGAs except where they overlap security related topics. HwA concerns are addressed in other NSA documents which can be found at https://www.nsa.gov/Press-Room/DoD-Microelectronics-Guidance.

This guidance is solely intended to reduce risk; adhering to it does not constitute compliance with DoD or commercial security standards.

3. Security features overview

Each FPGA family has a number of built-in security features and vendors have consistently added new protections with each release of new products. As FPGA complexity grows, so do the functions available to protect them. While there may be some commonality in the types of protections provided by each vendor, the specific combination of offerings and their implementation typically vary. This section discusses the various security features commonly available to protect FPGA devices. The following subsections identify the available security features, explains their operation, the tradeoffs of enabling them, and recommendations on their use.

3.1 FPGA device type selection

When selecting an FPGA device one must consider the underlying technology the FPGA utilizes to store the device configuration bits contained in the configuration bitstream. The underlying technology impacts security and design choices. The three major types of FPGAs utilize either SRAM, flash, or antifuse technology to store the device configuration. In the case of SRAM, the bit storage element consists of a single bit of volatile memory. That is, the SRAM bit can hold the bit value as long as power is applied to the cell. Flash bit cells are ones that can continue to store the bit cell after the power has been removed. Both SRAM and flash bits can also be reprogrammed. Antifuse bit cells are one time programmable cells that, once programmed, can never lose or change their value. Each storage type has its own advantages and disadvantages. SRAM-based FPGA devices require the full configuration data file to be stored externally and locally to the FPGA, usually in a flash device, to support the

programming of the FPGA upon each power up event. Flash-based devices store the configuration data internally and do not require additional external storage elements. An antifuse device is programmed once and also does not require additional local, external storage elements.

3.1.1 Tradeoffs

Because of the relative complexity of the manufacturing process, SRAM-based FPGAs have the greatest logic capacity and the best performance in comparison to the other FPGA device types. When considering selection of an FPGA, the large capacity and performance of the SRAM-based FPGAs make them the most popular choice, but other factors, such as operating environment, boot times, and system complexity, should be considered.

When operating in a high-radiation environment, flash and antifuse devices are very reliable in maintaining their configuration, while an SRAM device configuration is more susceptible to radiation. Choosing an SRAM device requires the user to ensure they understand the device's capabilities to detect and correct configuration cells affected by these events, as well as to plan how to safely handle device operation during the time between error detection and correction.

SRAM-based devices require longer boot times compared to the other FPGA device types because the configuration information is stored off-chip in encrypted form and thus must be decrypted and authenticated prior to being loaded onto the device.

SRAM-based devices are volatile and thus lose configuration when power is removed, while flash and antifuse devices are non-volatile and retain configuration when power is removed. Thus, systems utilizing SRAM FPGAs must also include a separate non-volatile memory containing the configuration file, while flash and antifuse FPGAs do not require an extra component to store the configuration file.

SRAM can be programmed an unlimited number of times. Flash has a high limit on the number of times it can be re-programmed (device-dependent). In contrast, antifuse devices are one time programmable (OTP) and thus the user has only one chance to program the part correctly.

3.1.2 Guidance

From a security perspective, it is recommended to utilize a flash-based device when possible since the configuration data is stored internally and is not accessible to an

adversary. While the antifuse devices have an immutable internal programming storage element, making them resistant to configuration modification and certain forms theft, they cannot be programmed or updated.

3.2 Configuration file encryption

Every FPGA must be configured to perform its intended function. The configuration input data is called the bitstream or configuration file. Because the configuration data used to program the FPGA contains the design, it is considered sensitive and requires special protection. FPGA vendors offer the capability to encrypt this data at rest and to decrypt it as it is being loaded into the end-use device. The encryption process relies on a secret key that is used for both encryption and decryption. The encryption/decryption key can be generated by vendor provided software or provided by the user. Encryption of the configuration data is performed by vendor provided software, but decryption is carried out in the FPGA hardware by a dedicated, on-chip decryption engine as the data is being loaded onto the device.

FPGA configuration data is often stored in separate non-volatile memory (NVM) on the printed circuit board (PCB) (for SRAM-based FPGAs), while flash-based and antifuse-based FPGAs store the configuration data on the FPGA itself. Storing the configuration data in encrypted form prevents an adversary from examining this data for the purposes of stealing secret algorithms, copying the application, maliciously modifying the function of the device, or subverting the application to serve their own purposes as long as the adversary does not have the secret key. The only commercially available encryption algorithm for protecting the bitstream and approved by the current Commercial National Security Algorithm Suite (CNSA 1.0 and CNSA 2.0) is the Advanced Encryption Standard-256 (AES-256). All recent FPGA devices offer AES-256 for configuration file encryption.

This algorithm should be used with one of three acceptable modes:

- Cipher Block Chaining (CBC)
- Counter mode (CTR)
- Galois Counter mode (GCM)

The following subsections provide detail on AES and each of the currently acceptable modes.

Encryption algorithm

The Advanced Encryption Standard (AES) is a symmetric block cipher algorithm that uses the same secret key for both encryption and decryption. FPGA vendors offer this encryption algorithm to protect the configuration file.

The recommended three AES modes have the desirable characteristic that repeated plaintext input blocks do not result in equivalent repeated output blocks. This removes any one-for-one correlation from input to output making code breaking more difficult. These modes and their differences are described here:

- Cipher Block Chaining (CBC) mode – In this mode the cipher-text output of one block (i.e., an encrypted block) feeds into an exclusive-or of the next plain-text block, which is then used for encryption of the next block as a new initialization vector (IV) (as compared to Counter mode). CBC mode's "reproduction" of "different" IVs for each block attempts to resist IV nonce misuse. While it is a highly secure implementation, the entire operation is strictly serial, preventing a parallelized encryption process and resulting in slower completion times.

- Counter (CTR) mode – This AES mode uses a counter to add variability to the output cipher text. Therefore each 128-bit block of input does not rely on the previous block. This allows for the same highly secure encryption method with the benefit of encrypting many blocks of text in parallel. As a result, CTR mode can be significantly faster than CBC.

- Galois/Counter Mode (GCM) – This AES mode is a variant of CTR and is implemented similarly, but with the addition of a hash-based authentication tag to verify the integrity of the data.

It is important to note that during decryption these modes result in a single bit output response to a single bit-flip input. An adversary can potentially exploit this weakness to identify security related bits in the decrypted configuration file if they have knowledge of the formatting of the configuration file. So unless GCM is used, an authentication mechanism is required. With any of these modes, authentication must be performed fully to verify the integrity of the encrypted configuration file prior to beginning decryption.

These modes all require generation of an initialization vector (IV). As with generating keys, the IV should be generated by a random number generator (RNG) compliant with

the NIST SP 800-90 series of documents (SP 800-90A Rev.1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators, SP 800-90B: Recommendation for the Entropy Sources Used for Random Bit Generation, and SP 800-90C (4th Public Draft): Recommendation for Random Bit Generator (RBG) Constructions).

3.2.1 Tradeoffs

Use of AES encryption/decryption algorithms for the configuration file provides confidentiality of the information at the expense of a slower boot process as it takes time for the device to decrypt the incoming configuration file. CBC and CTR modes provide stronger security but may introduce additional overhead and computational complexity that can impact performance.

The use of configuration file encryption comes with the cost of managing secret keys. This management includes the generation of the keys, the program's storage of the keys, secure injection of the keys into the FPGA devices, and security architectures to protect, revoke, and update keys. These are non-trivial tasks.

3.2.2 Guidance

- Use AES-256 for encryption of the configuration file in all cases. Programs should not rely on the native unencrypted complexity of the configuration data to protect their application from disclosure via reverse engineering.
- Use AES with either GCM or CTR mode. In addition, AES encryption should always be paired with an authentication algorithm that completes before beginning decryption.
- Verify that the FPGA vendor algorithm implementation (both the on-chip decryptor and encryption implementation in FPGA vendor software) is validated by the NIST Cryptographic Algorithm Validation Program (CAVP).
- In the event of a decryption failure while loading the encrypted FPGA configuration file, the user should abort the loading process and investigate why the failure occurred. The user should resolve the issue causing the failure and properly decrypt the configuration file. Loading an unencrypted configuration file is not advised.

- Some FPGA families offer a means to encrypt the secret key when loading it in the device and when storing it at rest. This key protection is called “key-wrapping”. Techniques to cryptographically wrap the encryption key must use the same number of bits as the key it is protecting. Wrapping an AES-256 bit key with an AES-128 key wrapping algorithm would reduce the overall security strength of the system to 128 bits. A 128 bit wrap is contrary to security recommendations put forth by NIST SP 800-57 and therefore is not recommended. The encryption key should be wrapped by encrypting it using the same number of bits as the key the wrapping is protecting.

3.3 Configuration file authentication

Authentication of an FPGA configuration file is the act of verifying the integrity of the configuration file and its authorization. It verifies that the configuration file contents have not been modified and that the contents come from an authorized source. Many FPGAs offer the user the ability to authenticate the FPGA configuration file using symmetric or asymmetric algorithms. This ensures the user only loads a design generated by an authorized source and whose contents have not been altered. Authentication can be performed on the entire configuration file at once or block by block. Checking block by block provides a more robust integrity check than a simpler Cyclic Redundancy Check (CRC) over the integrity of the entire file. In addition to providing integrity and provenance verification, authentication prior to decryption prevents the decryption process from being carried out on unverified data. This prevents the use of unauthorized configuration files from being used to expose key data through side channel attack techniques.

Symmetric algorithms

Examples of symmetric algorithms used by FPGA vendors to conduct authentication include a Keyed-Hash Message Authentication Code (HMAC) and the AES-GCM authenticated encryption algorithm. HMAC symmetric cryptographic algorithms generate a unique tag or code based on the contents of the configuration file and a secret (symmetric) key. The tag is included with the configuration file. During the authentication process, the FPGA also calculates what the tag should be using the received configuration file and its copy of the secret key. The FPGA compares the calculated tag with the included tag. If they match, the configuration file is trusted as unmodified and from an authorized source. An HMAC provides authentication and

integrity but does not provide non-repudiation. A third-party could not tell who authored the configuration data if the key was shared by more than one entity.

The AES-GCM algorithm is unique because it combines authentication and decryption, so it only requires one secret key. Using separate algorithms for decryption and authentication requires a key for each algorithm. To achieve the highest performance, some AES-GCM hardware implementations run decryption and authentication in parallel. However, this can expose secret key material to potential side channel techniques before, or in parallel to, authentication. This parallelism weakens authentication as a means to protect the secret key from side channel related replay techniques. Compromising one key compromises both security services. Finally, similar to an HMAC, AES-GCM only provides authentication and integrity but does not provide non-repudiation.

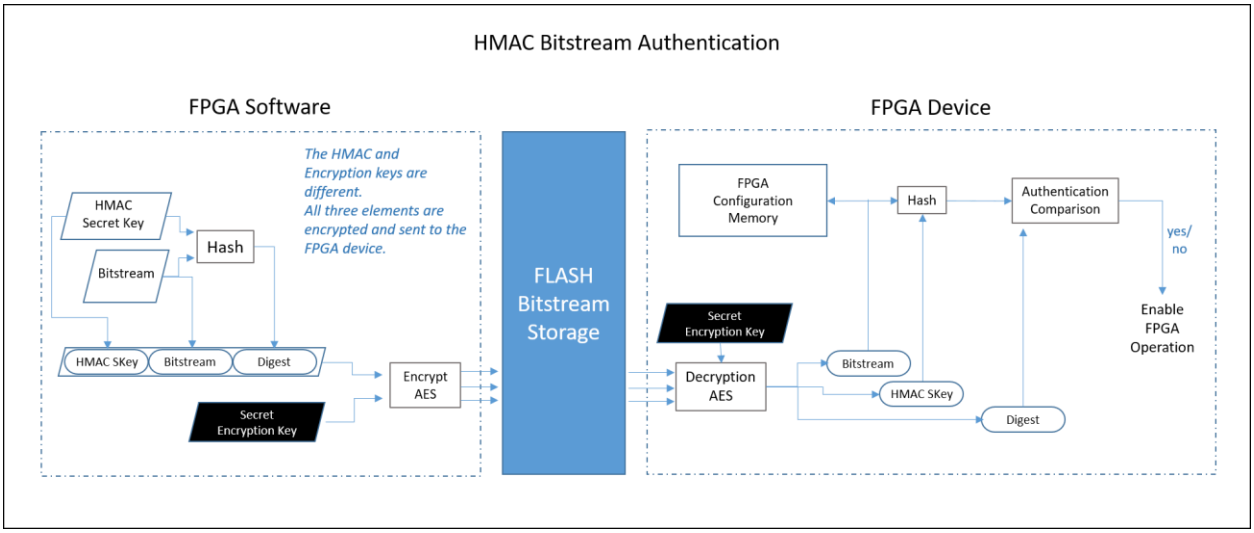

Figure 1: HMAC bitstream authentication

Asymmetric algorithms

Examples of asymmetric algorithms used by FPGA vendors include the Rivest, Shamir, and Adleman (RSA), Elliptic Curve Cryptography (ECC), and Elliptic Curve Digital Signature Algorithm (ECDSA) algorithms.

Like symmetric algorithms, asymmetric algorithms provide a way to verify the integrity and authenticity of a configuration file. This means the FPGA can determine that the configuration file has not been modified from its original form and originates from an authorized source. The FPGA designer signs a hash of the configuration file using a private key, and the FPGA contains the corresponding public key to verify the signature. The FPGA calculates the hash of the configuration file and verifies that it matches the

hash from the signature using the public key. If the signature is valid, the configuration is considered authentic.

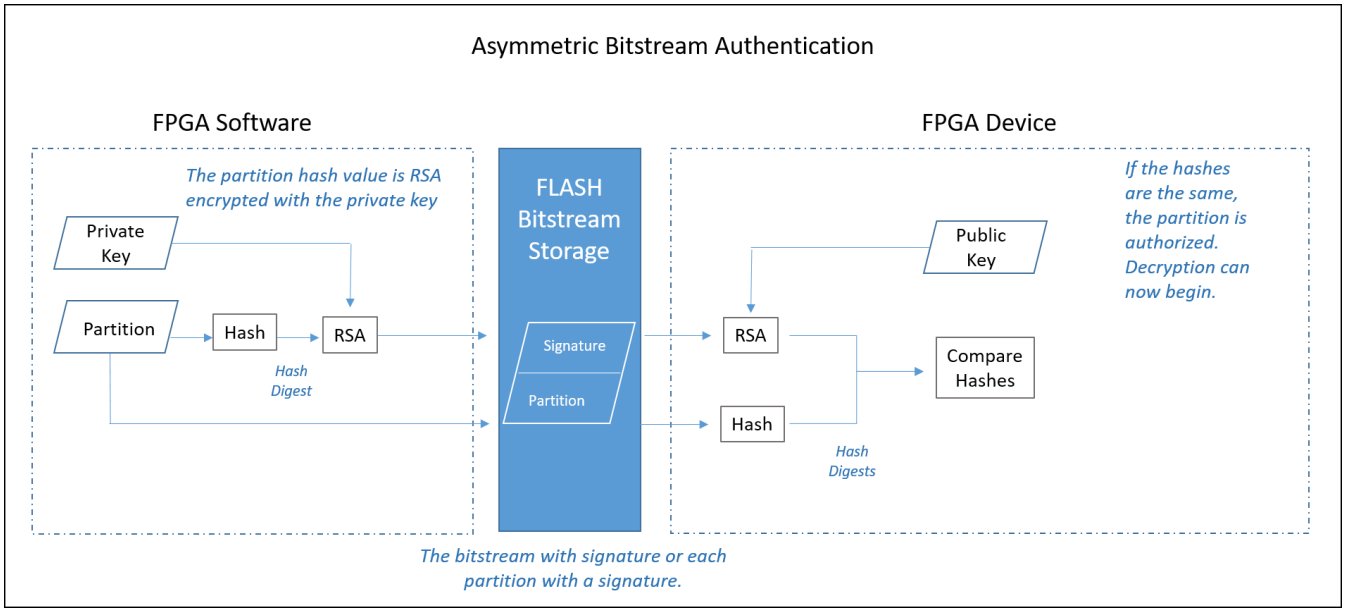

Figure 2: Asymmetric bitstream authentication

The public key can be shared with anyone, but the private key must be protected similar to how the secret key for symmetric algorithms must be protected. The FPGA must have some mechanism to be assured that the public key is authentic. One method to achieve this is by embedding the public key in a certificate signed by a trusted entity. Private and public key pairs should be generated in a controlled or secure environment. Additionally, many FPGA device families have a key revocation feature built in to be able to revoke trust in a key whose confidentiality may have been compromised. This function needs to be fully understood by the user to ensure an adversary cannot revoke the authorized authentication key and replace it with an adversary key.

3.3.1 Tradeoffs

Authenticating the configuration file comes at the expense of a slower boot process.

Asymmetric authentication requires more resources and is slower than using symmetric authentication algorithms, but it provides desirable cryptographic services not possible with symmetric algorithms, such as non-repudiation. Asymmetric authentication algorithms require greater computational resources than symmetric counterparts due to larger key sizes and more complex mathematical operations. However, modern architectures offer improved performance, allowing the authentication to be performed in real time as the configuration file is being loaded. Additionally, there are some cases

where the integrity of the FPGA configuration could require revalidation, adding additional time to the boot process. For example in irradiated environments, radiation can sometimes cause stored data to become corrupted and require revalidation.

Asymmetric algorithms require generation and management of public/private key pairs and distribution and verification of digital certificates, which requires more complex key management. While the use of asymmetric algorithms adds to the boot time, their use can prevent loading of a malicious configuration. It acts as another layer of protection against device subversion.

Symmetric authentication uses the same secret key by both the originator and the recipient for verifying the configuration file, so the secure distribution of the shared key between the designer and the authentication entity is critical. Any compromise or unauthorized access to the secret key must be considered a breach in authentication.

Symmetric authentication provides integrity and authenticity, but does not offer additional features like non-repudiation or secure key exchange, which are possible with asymmetric authentication.

When considering a particular algorithm, larger key lengths provide additional security. Key sizes between different algorithms are not as easily compared since different algorithms may have more equivalent security per bit. For example, elliptic curve algorithms use smaller key sizes than RSA algorithms, but have more equivalent security per key bit and are therefore considered more efficient.

3.3.2 Guidance

- The user should authenticate FPGA configuration files using CNSA asymmetric algorithms. Configuration file authentication is especially critical if the configuration data will leave the control of the program at any point. Combining both symmetric and asymmetric algorithms could provide an additional layer of security.

- The user should not allow the device to start decryption or read key material without proper authentication of the configuration file first. Failure to authenticate the configuration file should halt the configuration process or alert the device that authentication failed and give the user options for how to proceed.

- Asymmetric cryptography methods of authentication should be used prior to symmetric Authenticated Encryption (AE), but AE is not a substitute for asymmetric cryptography authentication methods. This approach serves as an important countermeasure against side channel attack techniques in which an adversary creates and loads random configuration files into the device in an effort to deduce the decryption key. Combining authentication and encryption via an authenticated encryption algorithm, such as AES-GCM, is not recommended since the FPGA configuration file should always be authenticated prior to being decrypted and loaded into FPGA configuration memory. AES-GCM should be used only for confidentiality and integrity.

- If the design team is using an FPGA that does not have a built-in authentication option, they should design this feature into their system or refresh to a device family with this function. This function could be implemented on another device on the PCB or it could be implemented in the FPGA programmable logic via a two stage boot process. If the design team implements its own authentication, they should design in the capability to control the failure response. Extra precautions should be taken to ensure the off-chip authentication cannot be compromised by PCB level tampering. These extra precautions fall outside the scope of this document.

- Additionally, while not recommended, a custom configuration file authentication option that runs in parallel to decryption can speed up the boot process. In a case like this, the parallel authentication function should be able to stop the configuration decryption and loading in the event of an authentication failure.

- The program team should assess the expected lifetime of the system and if their chosen authentication algorithm is projected to be vulnerable to a future cryptanalytically relevant quantum computer (CRQC) based attack techniques as specified in reference (d). Currently most FPGA families do not offer quantum resistant authentication algorithms, but the program team should monitor FPGA technology advancements and should urge vendors to implement a quantum resistant authentication solution and plan to incorporate it into their system when it becomes available.

- NSA recommends using CNSA authentication algorithms with implementations validated by an independent organization, such as the National Institute of Standards and Technology (NIST).
- If the system fails to authenticate the configuration file, the system should be programmed to shut down automatically. The failure should then be investigated and corrected.
- If the device has a key revocation function, the user should understand its implementation and document the key management plan. If the user does not anticipate updating the authentication keys, the revocation feature should be disabled. If the revocation function will be needed, the user needs to ensure that the trigger for this action can only be carried out by an authorized party.

3.4 Device identification

FPGA vendors have worked to address user community concerns about counterfeit FPGAs by assigning each device a unique identifier (ID). There are several methods used by FPGA vendors to implement a unique ID for each individual device. Combining the unique ID with cryptographic methods provides robust protection against counterfeit devices.

The use of unique and immutable chip IDs combined with cryptography is useful for securing certain processes in the operation of the FPGA, specifically:

- Verification of device authenticity during acceptance testing.
- Verification of device authenticity while it is or after it has been out of the control of the program either during the manufacturing process or after being fielded.
- Verification of device authenticity prior to performing a remote update.
- Management of a device's association with its various secret passwords.

Serial number

FPGA serial numbers are typically unique IDs assigned to each individual FPGA during manufacturing. This uniqueness can be leveraged for tracking and authentication purposes. Serial numbers can simplify inventory management by providing a way to track and manage individual FPGAs. However, they are exposed to potential malicious

actors and can be misused, such as by creating counterfeits using legitimate serial numbers, if appropriate cryptographic and secure boot processes are not utilized.

Serial number / on-chip ID

A common method FPGA vendors use to store a unique ID in FPGAs is to program them in fuse cells during manufacturing. The length of this value varies between vendors and FPGA families. The ID typically can be read from a device using Joint Test Action Group (JTAG) ports. Additionally some vendors offer one-time programmable (OTP) eFUSE cells that allow the user to assign a custom ID to control chip identification beyond the vendor ID programmed during FPGA manufacturing.

Various FPGA vendors maintain a record of package markings and the associated device ID as stored on the FPGA die, enabling the user to check if the packaged FPGA has the expected device ID. Some vendors make this information available via barcode or QR Code. Although these features are convenient, they represent an attack surface because this information can be easily replicated.

Cryptographically protected IDs

Some devices use a public/private key pair to enable authentication of the FPGA via decryption of a nonce (a randomly generated number that is used once). In this scenario, the FPGA receives a nonce encrypted with its public key. By successfully decrypting the nonce with its private key, the FPGA proves its possession of the corresponding private key and thus its identity. These devices include a digital certificate that contains the device's unique ID and public key, which is signed by the vendor with the vendor's private key. This signature serves as cryptographic proof that the certificate, ID, and associated public key were issued by the vendor. The FPGA having the corresponding private key for the public key in the certificate and decrypting the nonce serves as cryptographic proof that the ID in the certificate belongs to the FPGA.

Hard PUF

Newer FPGAs include Physical Unclonable Function (PUF) technology for unique device identification and anti-counterfeiting purposes. The PUF function derives a unique value for each FPGA by making use of manufacturing variations in the semiconductor manufacturing process. Vendors use various techniques such as SRAM-based and ring oscillator-based layouts to produce PUFs. PUFs may be leveraged to derive cryptographic key material without the need to store such key material in non-

volatile memory. This ID is stronger when the PUF supports multiple challenge/response values that could not be easily counterfeited by an adversary. When the PUF is implemented by the vendor in dedicated hardware logic, it is known as a hard PUF.

3.4.1 Tradeoffs

The tradeoffs in selecting chip IDs centers around the type of implementation, the ease of accessing the ID, and the strength of the security.

Package markings can be read with a scanner and do not require any electrical interaction with the part. While easier to read, package markings are easily forged or modified.

Any IDs that are stored internal to the device require a means to power up the part and conduct the necessary steps to read the ID using a PCB/socket and a computer. A fuse-based ID can be counterfeited by an adversary with manufacturing capabilities.

3.4.2 Guidance

- Verify the FPGA package serial number against the vendor bill of materials before powering up the device.
- The unique chip ID alone does not provide strong authentication of an FPGA. Binding the unique ID to a cryptographic technique provides strong authentication at the expense of having to store and manage private/public key pairs.
- Ideally, the FPGA should have a certificate signed with a private key from the vendor. The certificate should be stored on the FPGA, and the certificate should include the unique device ID number. In every case, a unique ID should be validated by a cryptographic technique. Verify the certificate based ID against the expected value.
- If the selected FPGA lacks a method for storing a strong unique ID, using a soft PUF can overcome this shortcoming. In this approach the user implements a PUF in the programmable fabric (known as a soft PUF) and reads out the unique ID to be associated with device. Once this is done, the soft PUF can be deleted from the fabric. The ID can be recreated at any time in the future by reloading the soft PUF and verifying the output. Note that the implementation of the soft PUF in

the fabric must be repeatable and exact. There are existing commercial solutions available in the marketplace to do this.

3.5 Device secret key storage

The program should plan to protect the secret key throughout its lifetime. This includes ensuring that the key is encrypted while not in use and storing the keys on the device in a secure manner. All available FPGA key storage mediums are either volatile or nonvolatile memory. Each of these mediums has several advantages and disadvantages that should be weighed against the program requirements and the overall threat environment.

a. Volatile key storage memory is reprogrammable and erasable. When power is removed the key disappears unless the key storage power plane is maintained with a battery. This is often referred to as a battery backup.

b. Non-volatile memory (NVM) stores the keys in a medium that does not lose its content when the power is turned off. These types of memory include flash-based cells and fuses. Fuses come in two types; poly fuse and eFUSE. Both of these can only be programmed once, unlike flash which can be reprogrammed many times. Fuses are the most commonly used NVM used for key storage.

3.5.1 Tradeoffs

Volatile memory is reprogrammable, allowing the keys to be revoked and updated, while most NVM keys are immutable. Volatile memory is also quickly erasable. When power is removed the key disappears, unless there is battery backup. This can enhance security as keys are not retained if the FPGA loses power or is reset. For example this allows for a speedy reaction to detected tamper events. The tamper response can pull the power to the key storage elements and quickly clear the key values, while the immutability of OTP NVM prevents the keys from being cleared or zeroed.

Immutable NVM keys cannot be erased or reprogrammed, but some device families offer enough non-programmed bits to allow a limited number of key revocations and updates. While NVM bits are hidden under layers of metal, the NVM keys are susceptible to physical reverse engineering techniques to recover the values. Secret keys may be wrapped by encrypting them with a different secret key to mitigate certain physical reverse engineering techniques.

Volatile memory is susceptible to single event upsets (SEUs), while NVM is resistant to SEUs and other conditions that can cause key corruption.

Using volatile memory also adds the additional cost of battery backup, increases the PCB size to house the battery, adds the burden of ensuring the battery is always powered, and possibly adds regulatory and disposal costs. Using NVM memory eliminates the cost and need for a backup battery.

3.5.2 Guidance

As with many security decisions, the program should consider all design requirements when choosing secret key storage. In general, the following guidelines should be considered for selecting a secret key storage medium:

- Flash memory presents a good option for key storage in that it does not require a battery backed power plane. Additionally, it can be updated with new keys and overwritten in a tamper event.
- SRAM-based volatile memory can also be updated with new keys as needed and can be cleared exceptionally quickly by removing power. However, it requires the use of a battery backed plane.
- Neither flash nor SRAM should be used in the case where a readback capability exists for the key. There are key theft techniques that are enabled by the availability of key readback.
- When the application contains very sensitive information and requires a speedy response to a tamper event, volatile memory based key storage should be considered to allow key clearing by pulling the power.
- For systems that cannot support the presence of a backup battery, non-volatile memory key storage almost always has be used.
- For FPGAs that will be fielded in an adversary accessible location and that are not expected to be recovered by the program for reuse or updating, eFUSEs should be used for the storage of security settings and keys.
- Only encrypted keys — keys that are encrypted by a different secret key, also known as wrapped keys — should be stored in eFUSEs.

- Unencrypted keys in eFUSEs should not be used in cases where the bitstream contents should be protected from disclosure since the keys will not be cleared in a tamper event.
- In cases where very limited key changes are expected, some devices provide additional eFUSE bits/words to support the storage of multiple keys, allowing limited key revocation or rotation. However, this allows an adversary to have potential access to future keys through reverse engineering techniques. It is not advisable to store multiple keys in OTP unless the keys are encrypted at rest.

3.6 Readback prevention

Readback is the process of retrieving configuration data or secret keys from an already programmed FPGA. This is useful for verifying that the configuration data and keys have been correctly programmed into the FPGA or for debugging.

In a fielded product, having the readback capability available or enabled can allow an adversary to capture critical data. It is important that readback is disabled in the operational environment. By disabling readback functions, the access to the configuration data is restricted. Disabling readback is critical in applications where the FPGA contains sensitive algorithms, sensitive cryptographic keys, or other valuable information that needs to be kept secure.

3.6.1 Tradeoffs

Leaving readback paths enabled is a security vulnerability by which an adversary can steal the configuration data or keys. There is no penalty to disabling this access.

3.6.2 Guidance

Readback capabilities for configuration data and keys should be disabled in all production/operational devices. Additionally, it is recommended to disable these features using fuses when available. Some device families provide the ability to read out a hash value of the keys or configuration data as a means to verify the programmed data. This solution provides the ability to perform verification while avoiding the security concerns of a full readback capability.

3.7 Tamper detection and response

Tamper detection

Tamper detection is a mechanism used to detect any unauthorized access (also called a tamper event) or compromise of the FPGA's function. It is used to protect the cryptographic keys and intellectual property (IP) of the FPGA from being stolen or reverse-engineered. Many FPGAs provide various anti-tamper physical sensor components built into the devices to detect:

- Out of specification voltage levels or swings in voltage levels, which can be indicative of a compromise attempt on the security bits of a configuration file.
- Glitches on the external input clock signals, including an abnormal duty cycle, jitter, and other glitches appearing on the signal, which are indicative of attack attempts on the security bits of the configuration file during load. Once the security bits are compromised, the adversary can then further compromise the device or steal information using standard read commands and without worry of a tamper response.
- Activity on the JTAG port, which may be indicative of an intrusion attempt into the device. JTAG tamper monitoring is used to detect unauthorized access to the JTAG port or specific JTAG instructions. This generally means unexpected activity on the ports. If the JTAG ports have been deactivated in the production devices, any activity most likely represents a tamper event.
- Extreme temperatures or detection of rapid temperature changes, or changes outside of a specified range.
- Opening of the FPGA device package lid. Additionally, some devices can detect if security meshes have been breached by lab probing.
- Intrusion into the device via factory testing ports.

Tamper response

Tamper responses come in two categories; built in and custom. In the case of built in tamper responses, the FPGA vendor has provided responses that can be triggered automatically by the sensor. These include many of the options listed below. The custom response is one where the user application is monitoring the tamper sensors

and triggers a response designed by the user and carried out by the application. Not all FPGA devices support custom responses. Different FPGA families offer different tamper response options. The following are types of built in tamper response options found in various device families:

- Release a notification – in this case, the FPGA device raises an error flag readable by the chip and the system it is in, indicating that a tamper event has occurred. This is often used when the system should continue to operate once the tamper event has been detected.
- Restart – the FPGA is made to perform a restart by power cycling.
- Reset – the FPGA is made to perform a restart and reload with a known bitstream.
- Clear data – The FPGA device clears the sensitive information in the device, such as keys, and verifies that the keys have been cleared by reading back the values and ensuring they have been zeroized.
- Lock I/O – The FPGA locks the device I/O, which immediately prevents data exfiltration or input of malicious data.
- Device wipe and lock - also known as security lock down of the FPGA. All user security bits behave as though they are locked. The user-defined state no longer matters. All sensitive data is erased. Any passcodes that were unlocked become immediately locked.

3.7.1 Tradeoffs

Having tamper detection may require including additional logic blocks in the design to enable the operation and monitoring of the sensor. The additional logic requires minimal resources from the programmable fabric, but adds to the power and size budget of the application. These already exist and do not require further development on the part of the user.

Custom tamper responses require development and testing work on the part of the user and are not always supported by FPGA devices. However, without active tamper sensors, a device has no ability to sense and address a compromise attempt.

3.7.2 Guidance

The program should use tamper detection sensors when available and use an appropriate tamper response when triggered. Typically, each of the tamper detection monitors can be enabled individually. When tamper detection against the security bits has been triggered, it should be assumed that the targeted security bits have been compromised and that any sensitive information loaded on the device is vulnerable. It is not enough to simply restart the configuration process. At this point, all sensitive data should be overwritten or erased before proceeding.

3.8 Internal configuration clock

In an attempt to thwart clock glitching attack techniques, some FPGAs provide the option to configure the part using an FPGA internally generated configuration clock signal. This signal drives the internal logic that performs the configuration process and is visible on an output pin to synchronize the reading of the bitstream from the flash. Being internally generated, the clock is protected from being manipulated by an adversary with access to the physical PCB or FPGA device.

3.8.1 Tradeoffs

This internal configuration clock signal can be slower than one generated from the PCB and has a larger margin with respect to the quality of the duty cycle.

3.8.2 Guidance

It is recommended that the user take advantage of the internal configuration clock when available, especially when it is expected that an adversary could have physical access to the device at some point.

3.9 SEU/error detection

In some application areas, an SRAM-based FPGA device’s configuration data can become susceptible to corruption due to environmental conditions, such as radiation exposure. This kind of corruption can impede the correct operation of the application. Having the ability to validate the integrity of the data and then correct corruption during operation can be vital for certain systems. In a high radiation environment, this kind of data corruption is called single event upsets or multi-event upset (SEU/MEU). SEU/MEU monitors detect errors caused by electrical noise, radiation, component failures, and other factors. Error correction reduces the risk of data loss and ultimately helps to eliminate downtime.

Error detection/correction requires additional resources and energy, and can introduce latency. Additionally, some errors cannot be detected or corrected. Designers should factor in additional verification time, as error correction implementation can be time consuming. Error detection and correction should be applied in the following ways:

- The ability to detect and correct errors within the boot-up controller memory segment, allowing recovery from transient faults, such as single-bit flips in memory or registers, without disrupting normal operation.
- The ability to detect and correct errors within the FPGA fabric, which refers to the programmable logic and interconnections within an FPGA.
- The ability to detect and correct errors within the memory segment storing device configuration and security settings.

3.9.1 Tradeoffs

SEU detection requires the FPGA device to read through all the configuration data and create a hash value to compare against an expected value. Due to the amount of data, this cycle can take a significant time from a microelectronic view point. In many cases, chip operation must be paused to perform this check, causing a loss or delay in processing.

3.9.2 Guidance

SEU detection and correction is recommended in two cases:

- There is a risk of data corruption due to a known environmental factor.
- There are sufficient downtimes during normal operation that would allow for an SEU check.

Outside of these conditions, there are usually no compelling reasons to absorb the cost to perform these checks as the device is expected to behave according to specifications otherwise.

When the system should perform SEU checks, first determine if the targeted FPGA device family can perform the SEU checks in the background and in parallel to normal operation. If not, determine if the system can afford the cost of stopping operation to perform SEU checking. If not, seek out a device that can perform SEU checking in the background.

3.10 Side-channel attack protection

Side channel attacks (SCA) involve collecting various emanations from the FPGA device, such as power draw or electromagnetic radiation, during the decryption stage of the configuration load, applying statistical analysis to the data captured, and unmasking the secret keys. Because SCA techniques are well-documented, mature, and effective, users should select FPGAs with effective side-channel resistance. These resistance techniques generally focus on hiding FPGA emanations or obfuscating them with additional noise in the collection space. Additional approaches include limiting the usefulness of information that an adversary can collect using SCA by using multiple secret keys with each key only used for small portions of data (known as key rolling) and using authentication to prevent the adversary from providing their own bitstream. As authentication was covered earlier, the remaining techniques are reviewed below.

3.10.1 SCA resistant cryptographic functions

In many FPGA families, vendors have developed or licensed cryptographic functions that are resistant to SCA. The details of achieving this resistance are proprietary and tightly guarded. Academic efforts have shown that side channel analysis attack techniques are well understood and effective at stealing keys of any length given sufficient time and data against an unprotected hardware based cryptographic engine. The strength of a decryption engine is measured in “traces.” A “trace” refers to a single round of decryption during which side channel emanations are collected. The higher the number of traces required to extract a secret key, the stronger the protection. While it is not clear at this time how effective current commercial SCA protections are, they are documented with a trace metric and are a good starting point.

3.10.1.1 Tradeoffs

The use of strong SCA protected cryptographic functions can increase the power, time, and area requirements of the device.

3.10.1.2 Guidance

If an FPGA-based system’s emanations could be monitored by an adversary, the system should be using FPGA families whose cryptographic functions have the strongest possible SCA protections built in. Multiple protections may be better than single mitigations. For situations requiring high assurance or high security, programs should engage a government lab or Joint Federated Assurance Center (JFAC) to ascertain the strength of the cryptographic engines’ SCA resistance.

3.10.2 Key Rolling

In addition to protecting the decryption process from analysis, some FPGA families provide a key rolling feature. Key rolling uses multiple secret encryption keys to protect different sections of the configuration file. The configuration file is broken up into blocks, each encrypted by a different key and containing the key for the following block. In these cases, the user can choose how many blocks to encrypt with a given key. The goal of this methodology is to limit the amount of information an adversary can analyze before the secret key is changed.

3.10.2.1 Tradeoffs

Key rolling requires additional time to decrypt a new key and then store it as the current one. While an increasing number of key rolls directly correlates with increased resistance to SCA, the time to perform the configuration process also increases. Increased configuration time is the tradeoff for higher security.

Additionally, key rolling complicates the management of keys for the program since revoking and updating the keys become much more difficult.

3.10.2.2 Guidance

When key rolling is available, it can be a strong protection against SCA techniques and the user is encouraged to use it. The number of key rolls can only be determined by understanding the amount of data needed to perform SCA against a specific cryptographic engine. Thus, the specific number of keys that should be used is device dependent. However, it is recommended to generally use more key rolls to increase the increase the security of the secret keys.

3.11 System on a chip security features

System on a Chip (SOC) FPGAs have both the configurable fabric of a standard FPGA and the processor based system of an application specific integrated circuit (ASIC). The SOC portion of these devices typically is comprised of a processor, memory, a bus system, general purpose I/O (GPIO), high speed protocol support, cryptographic functions, and a host for peripherals embedded on the FPGA as hardened IP functions. Some of the SOC resources are accessible by applications programmed into the FPGA side. Unlike the FPGA fabric, the SOC portion of the design cannot be modified or reconfigured. Together, these two different systems provide extraordinary capability with very little development risk to a program. However, in addition to the standard FPGA, the SOC addition brings its own set of attack surfaces that must be protected. FPGA

vendors have built in a number of protections that programs should leverage when designing a SOC FPGA.

3.11.1 Secure boot

Initial boot code is used to initialize the FPGA at power up. This code is developed by the vendor and run by the onboard central processing unit (CPU) upon release from reset following power up. It performs “device ready” tests and then hands off control of the boot process to the first stage boot loader. This code is generally immutable and stored in a memory device programmed during manufacturing that cannot be altered. The memory device is typically a read only memory (ROM) and is not readable or writable by the user.

3.11.1.1 Tradeoffs

Having an immutable boot ROM protects the startup sequence of the FPGA from tampering. If the startup boot code is reprogrammable, the power up process becomes vulnerable to potential modifications and would need to be monitored and protected.

3.11.1.2 Guidance

When SOC FPGAs are required for a program, using FPGAs with immutable boot code storage, such as a ROM, is recommended.

3.11.2 First-stage boot loader encryption

First-stage boot loader (FSBL) code is executed by the processor portion of the SOC after initial power up is complete. Once the initial boot code completes a successful power up, it transitions control to the FSBL instructions. These instructions are binary executable code contained in an off-chip, non-volatile memory (NVM), such as a flash, and are executed by the SOC CPU. The FSBL configures the FPGA fabric or loads the processor operating system. Secure start-up is essential to the operation of the device. Encryption of this code is important to protect it from disclosure or modification.

3.11.2.1 Tradeoffs

As with other forms of device configuration, protecting this data comes with tradeoffs. Encrypting the FSBL code prevents an adversary from examining the details of the startup procedures for possible vulnerabilities. However, decrypting the FSBL code adds time to the startup procedure and provides additional volume of data against which to perform SCA, unless key rolling is used.

3.11.2.2 Guidance

The FSBL should always be encrypted, and key rolling should be used if available. Key lengths should be in accordance with a CNSA approved algorithm and key length for NSS systems, or a NIST-approved algorithm and key length for non-NSS systems.

3.11.3 First-stage boot loader authentication

FSBL authentication is the act of verifying that the contents of the FSBL have not been modified and that they are authorized for use. Authentication verifies the authenticity and integrity of the FSBL. Typically, authentication is carried out by the use of cryptographic functions such as RSA or ECDSA. Performing authentication first before decrypting the FSBL prevents an adversary from conducting SCA attack techniques in the interest of stealing the FSBL encryption private key.

Note that some FPGA families offer replay prevention of older versions of the FSBL by allowing the user to assign a version number to each FSBL version in the authentication data and specify which version or versions are permitted to be loaded onto the device.

3.11.3.1 Tradeoffs

FSBL authentication prevents an unauthorized person or group from providing a new and malicious FSBL to the processor, since any unauthorized modifications are detected and rejected. FSBL authentication adds time to the startup process and can only be used in a device that supports FSBL authentication accompanied by the ability of the device to respond to a failure. This requires additional development effort.

3.11.3.2 Guidance

Always authenticate the entire FSBL utilizing a CNSA approved algorithm and key length for NSS systems or a NIST approved algorithm and key length for non-NSS systems. Full authentication should take place prior to decryption. Authentication of the entire FSBL is a priority over encryption and should be done in all cases.

3.11.4 Secure memory

Secure storage consists of non-readable on-chip memory for security and boot settings in a secure zone. This memory location is protected from direct access by boot code or fabric logic. This storage is often protected from updates or modification with a long password to prevent malicious alteration.

3.11.4.1 Tradeoffs

Secure storage has only positive value and no operational downside. Secure storage is a good option for storing sensitive data, such as keys, passwords, and boot loader code. This helps to prevent malicious modification of these values by the fabric logic or other application features. This type of memory is generally only writable by the user during provisioning and with the use of passwords or keys. Only specific features that require this information are allowed to access it.

3.11.4.2 Guidance

Users of SOC-based FPGAs should use devices with secure storage. The settings and keys that control the boot process should be protected from access and modification by an adversary. They should be kept in a special “Trusted” or “Secure” area of the device as provided by the vendor. Decryption keys, public keys for authentication, security settings, passwords, and other sensitive data should be stored in secure storage.

3.11.5 Boot order

Some SOC-based FPGAs allow the user to choose whether to configure the SOC or the FPGA fabric first. In some cases, there is no choice as the order is built into the device.

3.11.5.1 Tradeoffs

In the case of SOC-based FPGAs, the specifics of the two implementations should determine which portion should be programmed first. Whichever one is booted first has the opportunity to monitor the second. Therefore, whichever side is booted more securely should be booted first and then should monitor the booting of the second.

3.11.5.2 Guidance

Minimally, any loading process should entail authentication and decryption in that order. The boot order should be dependent upon the available security features of either side. The side with the most security features and that offers the more secure loading process should be loaded first to provide a root of trust for loading the second side. The first side should be tailored to provide any needed security monitoring for the second.

3.12 Partial reconfiguration

Some FPGAs offer a partial reconfiguration (PR) capability that allows a predefined region of the programmable logic (PL) to be reconfigured while the other PL regions remain operational. PR provides the user with the flexibility to swap different functionality in and out of the PR region without rebooting the entire FPGA and

triggering a full reconfiguration. One use case for PR is when an application has to support multiple user selected algorithms for the same function. Without PR, the application would have to contain and implement all the algorithms along with multiplexing circuitry to allow the user to select the desired algorithm. Using PR results in the PL only having to support and contain the currently desired algorithm. This results in significant PL area and power savings.

3.12.1 Tradeoffs

Early planning is required to use PR, as the design has to be partitioned to separate the PL configurable regions and the static regions. Each PR design should be simulated with the entire design and have the same logical and physical footprint as the other PR designs. The design team should follow the vendor recommended PR process for their device. With PR, the user has to create and store the individual PR bitstreams. The user must investigate options for encrypting and authenticating the partial reconfiguration bitstream for their specific FPGA, as there may be limitations. In addition to saving PL resources, partial reconfiguration is much faster than loading an entirely new configuration because configuration time is proportional to the size of the configured region.

3.12.2 Guidance

When using partial reconfiguration, the following recommendations should be followed:

- As with the main configuration file, the partial configuration files should be authenticated prior to decrypting and loading. The authentication methodology should be of the same strength as that used to authenticate the main configuration file.
- As with the main configuration file, the PR file should be encrypted when stored in off-chip NVM. The decryption methodology and keys should be of the same strength as that used to decrypt the main configuration file.
- Tamper sensors should remain active during the PR load and should maintain their ability to sense compromise attempts and trigger appropriate responses.
- The primary application should include monitoring functions to detect compromise attempts during the PR load process.

- Off-chip access to the PR regions should be closed off during the PR configuration.
- The primary application should ensure the PR process completes and has been loaded properly.
- The primary application should have a means to recover from a failed PR load attempt.
- If PR is an available function in the chosen device family, but not used by the application, it should be disabled and prevented from being made active. PR can be a powerful access tool for an adversary. Disable PR and close off all access points to it when not used. When used, ensure only authorized PR configurations can be loaded and only at the appropriate time.

3.13 Physical isolation flow

Some FPGA device families offer a design flow in which the user can physically isolate user specified logic from the remaining logic within the FPGA fabric. The process should also include an independent flow to verify the isolation. Without this capability, designers can guarantee isolation only by placing the logic to be isolated on separate FPGAs. One example of an application that requires this capability is an encryption module requiring red (plaintext) and black (cipher text) separation. The designer needs to be assured that there is complete isolation between the red and black signals so there is no chance of coupling between nearby nets, causing information to leak from red to black logic. If logic separation is required, the design team should allocate extra time to incorporate these techniques into their design flow. Note that most FPGA tools offer the ability for the designer to place logic in specific regions, but logic placement does not necessarily isolate the routing from the different regions.

3.13.1 Tradeoffs

If logic isolation is required, an isolation design flow will save on system cost and complexity by reducing the number of devices in the system at the expense of more design implementation time for the FPGA using the isolation flow.

3.13.2 Guidance

If physical isolation of logic is required, the user should either select an FPGA device with the capability to isolate and verify logic isolation or implement the logic in separate

devices. If using the isolation flow capability, the user must use the vendor capability to verify the isolation.

4. Security subjects and methods overview

This section discusses security considerations that are best practices, but may not be directly implemented or monitored on the device itself. These best practices are crucial for ensuring the overall security of the system, even if they are not directly related to the functionality of the FPGA hardware.

4.1 Transition to production

In preparation for fielding the FPGA-based system, the program will provision the keys, set security bits, and load configuration files in the NVM blocks. It is essential for the program to review their production-ready application and settings to ensure there are no unintentional entry points or development features that could provide an adversary unauthorized access.

4.1.1 Guidance

The following items should be reviewed prior to production and fielding:

- Ensure the JTAG access has been completely disabled or limited to boundary scan chain testing.
- Ensure all debugging modules and code monitors for development have been removed.
- Ensure all security bits have been set correctly.
- Ensure decryption and authentications settings have been set correctly.
- Ensure any unnecessary I/O access has been removed.

4.2 Authentication failure recovery

Authentication of the configuration file during power up is essential. It is equally important to be able to recover securely from an authentication failure. An authentication failure is one in which the FPGA does not recognize the loading configuration file as authorized or having data integrity. When this occurs, the configuration process should be halted and the device should remain unprogrammed. However, keys and security bits that were provisioned previously remain intact. When

this failure occurs, the intended function performed by the device is now unavailable. The user has a number of possible responses:

- The device can attempt a number of additional configuration attempts, up to a set maximum number.
- The system can attempt a hard reset and power cycle before attempting a configuration load again.
- The system can load the device with a safe mode configuration file with minimum functionality to conduct a remote update or inform “home” that there has been a configuration failure.
- If sensitive data exists on the FPGA, the system can clear the sensitive data, tristate all the I/O, and block any additional configuration attempts. Depending on the FPGA, these steps can be permanent or unlocked with a passcode.

The responses are generally dictated by how sensitive the configuration file is and whether or not an adversary can reasonably be expected to have physical access to the device.

4.2.1 Guidance

For authentication failure, the following responses are recommended:

1. Allow a limited number of configuration retries if the device supports a limitation with a tamper response.

   *If the device is assumed not to be in the physical possession of the adversary proceed to step 2 and 3. Otherwise implement step 3.

2. Load a safe mode configuration file that:

   a. Does not include the application functionality

   b. Performs a check of the configuration data integrity

   c. Performs a remote update to replace the bad application configuration file

   d. Can call home and report a problem

3. If there is a concern regarding theft of sensitive data, use FPGA built in tamper response options to:
    a. Tristate all I/O
    b. Delete all keys
    c. Prevent additional configuration attempts
    d. Sanitize flash contents

4.3 Configuration file decryption failure recovery

A successful decryption of the configuration file is necessary to program an FPGA. It is recommended that the encrypted configuration file always be authenticated prior to decryption. Besides being good security practice, authentication also eliminates configuration file corruption as a possible reason for decryption failure. In the event of a decryption and load failure, the user must be prepared to respond to the event in a safe and secure manner. In the case of a failure, the cause can be:

- The configuration file has been modified or corrupted.
- There is a hardware problem on the board.
- The wrong key was used to encrypt the configuration file.
- The secret key on the FPGA is the incorrect one or has been corrupted.
- An attack on the voltage or configuration clock has resulted in a bad load.

There are a number of responses that can be chosen by the user when the decryption fails:

- The device can repeat the load and decryption process, up to a set number of times.
- The system can attempt a hard reset and power cycle before attempting a configuration load again.
- If the function is available in the given device family, the system can ping the FPGA for a hash of the decryption key to check it for integrity.

- If the function is available, the system can ping the FPGA for a tamper event on the clock or voltage.
- The system can load the device with a safe mode configuration file with minimum functionality to conduct a remote update or inform “home” that there has been a configuration failure.
- If sensitive data exists on the FPGA, the system can clear the sensitive data, tristate all the I/O, and block any additional configuration attempts. Depending on the FPGA, these steps can be permanent or unlocked with a passcode.

4.3.1 Guidance

For a configuration decryption failure, there are a number of different steps that could be taken depending on the security profile of the application. Several of the steps are recommended in all cases and are noted as such below. The others are listed as possible actions that could be helpful.

Recommended in all cases:

- Verify that the configuration file contents were authenticated and verified. This will confirm that the configuration data was not corrupted.
- Check if a tamper event was detected during the attempted load. Confirmation of a tamper event dictates immediate action to protect sensitive data.
    - Clear and zeroize all keys
    - Clear and/or zeroize the configuration storage device. This is generally a flash device
    - Tristate the FPGA I/O
    - Block/prevent additional configuration attempts.
- Depending on the device family, this can be done permanently or unlocked via a secret password.
- Repeat the configuration file load several times, but limit the number of attempts, as each iteration exposes key information to a potential side channel attack. If it is believed the device is in possession of the adversary, do not attempt to reload the configuration file.

Additional Options:

- Verify the validity of the secret key by reading out its hash to verify its integrity. However, the device should never read out the secret keys in the clear.
- Reconfigure the device with a second “safe mode” application that can verify the system state, initiate a remote update, or call “home” to report a problem.

4.4 Remote update

Remote update refers to the ability and requirements associated with updating any of the programmable/configurable portions of the FPGA from a remote location. This means the device is updated while in the field and not in the physical possession of the Program. Software, configuration data, keys, and macro settings of hard functions, such as PCIe or USB, can be updated remotely.

Remote updating provides the user with valuable flexibility and benefits, including:

- Closing vulnerabilities or bugs in the application without returning the device from the field.
- Updating the application with new and improved functionality.
- Revoking, updating, and replacing secret keys that have been compromised without returning the device from the field.
- Responding to a tamper event by returning the device to a safe state.
- Authenticating devices in the field.

4.4.1 Tradeoffs

The benefits of remote updating are accompanied by additional security considerations. Remote updates suffer from the same types of attack techniques as the regular configuration process. However, during a remote update, the device is located in the field and now risks exposure to adversary-in-the-middle techniques. Additionally, in the event the update is not successful, the device could be left broken or vulnerable to compromise.

4.4.2 Guidance

When using a remote update capability, the following are recommended:

- Use a fully tested golden configuration file as a base image.
- Ensure the entire NVM memory space has been zeroed prior to loading the initial configuration file.
- Remotely authenticate each FPGA device before sending an update.
    - Have a failure response plan in the event the device cannot be authenticated.
- The FPGA should authenticate the update file upon receipt and prior to storing in NVM.
    - Have a failure response if the file does not pass authentication, such as:
        - Retry download up to a set number of times
        - Ping owner status
        - Block update command after a set number of failures
            - Allow unblock via password
- Store remote update files in the NVM PCB storage device after authentication.
- Retarget boot pointer to the memory location of the update and reboot.
- Authenticate and decrypt updated configuration file prior to loading
    - Have a load failure response, such as:
        - Retry loading up to a set number of times
        - Fall back to golden boot configuration and ping “home”
- Use of vendor features to support remote update functionality should support update file encryption and authentication with the ability to create recovery options in the case of a failed update.
- Store multiple keys at initial provisioning to remove the need to transmit keys.
    - Wrap keys (encrypt the keys using a Key Encryption Key) before storing them.

- If new keys have to be sent remotely, send them only with authentication and encryption.

4.5 Key management

Key management is the process of generating, storing, distributing, revoking, updating and using cryptographic keys to protect the FPGA configuration and user data. Although many FPGAs offer key management functions, the ultimate responsibility belongs with the program.

4.5.1 Key generation

FPGAs may require generating multiple unique keys for various security services, as well as initialization vectors depending on the algorithm mode. A key should never be used for more than a single purpose.

In general, key generation can be done by using a random number generator (RNG), a FIPS 140-3 compliant hardware security module (HSM), or FPGA vendor-provided key generation tools. For the security of DoD systems, only keys generated via a random number generator meeting the requirements outlined in NIST SP 800-133 Rev. 2 are acceptable.

FPGA vendors provide various options for the generating encryption keys. These options include allowing a user-generated key, having the vendor software generate the key based on a bitstring or strings entered by the user, or allowing the vendor software to create the key with no user input.

4.5.1.1 Guidance

Programs should maintain control of the key generation process using NIST-approved standards. These standards have been approved to meet necessary security requirements. Programs should avoid using software techniques for key generation and use hardware techniques instead.

4.5.2 Key copy storage

If system requirements necessitate that copies of secret and private/public key pairs be stored by the program, that storage plan should attempt to protect the keys throughout lifecycle. This includes protecting the storage of all copies of the keys and ensuring the copies are always encrypted when at rest. Keys that need to be protected include any non-fielded key information held by the program for administration purposes.

4.5.2.1 Guidance

Available storage options are:

- FIPS 140-3 compliant HSMs are considered the most secure option for key storage and cryptographic operations. They provide a high level of physical and logical security and are designed to resist various attack techniques.
- A secure key vault or key storage system within the program's infrastructure uses dedicated hardware or specialized software solutions for key management and protection. This is the best option if an HSM cannot be used.
- For highly sensitive keys that are not needed for regular operations, consider using secure offline storage options. This might include physical hardware devices stored in a secure, controlled environment.
- The program can choose to store keys in an isolated environment on the network. The isolated environment can be protected by access controls and network segmentation to limit exposure. However, this storage mechanism can be susceptible to insider threats.
- Virtual machines to isolate key management processes can help protect keys from unauthorized access or other attack techniques.
- Certain secure software is designed to provide a secure environment for key management within software applications. However, they are not as secure as HSMs and are more susceptible to certain types of attack techniques.
- Third-party service providers that offer key management services allowing the program to securely generate, store, and manage cryptographic keys in a cloud-based environment. The keys themselves should never be managed by a third-party provider.

4.5.3 Key distribution

Key distribution refers to transport of secret and private/public key pairs from the program's secure storage to the facility/tool that will inject them in the target FPGA devices. NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management has guidance for approved algorithms and key distribution protocols.

4.5.3.1 Guidance

When distributing keys to FPGAs, use secure transport mechanisms to protect the keys during transmission. Encryption and secure channels should be used to prevent interception. This can include electronic transmission or physical media transport. The criteria for performing this key transport is outside the scope of this document.

4.5.4 Key revocation

Revoking cryptographic keys is the process of invalidating cryptographic keys to maintain security in cases of compromise, expiration, or changes in security policy.

In the planning stage, the program should create a key management plan that identifies when a revocation is necessary, as well as the implementation procedures.

4.5.4.1 Guidance

The key management plan should contain the following procedures for key revocation. Before current keys are revoked, the new keys should be generated using a compliant RNG or HSM. Once the new keys have been generated, the device and associated systems need to be updated. The new keys need to be distributed using secure channels and encryption. Verify the key integrity during distribution. Then, the revoked keys should be replaced with new keys on devices and systems and the devices and systems should be tested for proper functionality using the new keys. After the new keys have been distributed and tested, revoke (disable or invalidate) the old keys to prevent further use. The method of revocation will depend on the specific cryptographic system and key management infrastructure in use. Although the revoked keys are no longer in use, they still require secure storage. Ensure that access controls and policies are updated to reflect the revocation of the keys. Log and monitor key related events, allowing for the tracking and detection of unauthorized attempts to use revoked keys. Inform all relevant parties, including system administrators and users, about the key revocation and ensure that they are aware of the change and that it is properly implemented in their systems.

4.5.5 Key updating

Key updating, also called key rotation, is the process of replacing existing cryptographic keys with new ones. This is done to maintain security and confidentiality, especially when keys have been in use for an extended period or when there is a need to change encryption algorithms or key lengths. The program should document how often key rotation should occur and the process for implementing key updates. Regularly

changing keys is a security best practice to limit exposure in case of a key compromise. It also serves to exercise the key update process, in case a key update is needed as part of key revocation.

4.5.5.1 Guidance

The program should ensure that keys are rotated regularly to limit exposure. The procedures for updating keys should be documented in the program's key management plan. The procedures are similar to key revocation, but without actually revoking the previous key.

4.5.6 Key management

The program has the responsibility to protect cryptographic keys throughout the key's lifecycle. The program should document the various aspects of key management in a key management plan, including details for each of the elements of key management.

4.5.6.1 Guidance

The key management plan should include the following:

- Ideally, cryptographic operations involving keys should be processed within a FIPS 140-3 compliant hardware security modules (HSMs) or secure cryptographic modules using an RNG to ensure security.
- Keys should never be stored in plaintext format. Instead, they should be encrypted or protected using strong encryption methods. The encryption method should be equivalent to or higher than what the keys are used for.
- All keys should be stored in a cryptographic vault, such as an HSM or an isolated cryptographic service. These specialized devices provide a secure environment for key management and cryptographic operations.
- Key generation and cryptographic operations should be performed in a secure location. This typically involves using secure hardware, such as an HSM, and ensuring that access to the keys is tightly controlled and monitored.
- Keys that have been compromised should be revoked and keys that have reached their end-of-life should be updated in accordance with the sections above on key revocation and key updating.
- Keys should be changed regularly to limit exposure.

4.6 Key management tools

Common tools to manage cryptographic keys include HSMs, key management software libraries, and secure element integrated circuits. These tools are used for generating, storing, distributing, and handling cryptographic keys at the program site.

An HSM is a physical device with the objective of protecting cryptographic keys. HSMs perform all major cryptographic operations, including encryption, decryption, authentication, and other key management functions, so that the keys can be used without needing to leave the protection of the HSM.

Key management software libraries allow customization of key management processes based on the specific needs of the application. Many key management libraries are open source.

Secure element integrated circuits (IC) are specifically designed for secure key management, offering strong protection from attack by providing physical isolation for sensitive operations, making them highly resistant to tampering. These often include cryptographic accelerators, improving performance for cryptographic operations. Secure element ICs can be incorporated into the program's systems for key management, but can be costly, are only offered by specific vendors, and may require additional hardware design.

4.6.1 Tradeoffs

HSMs provide a high level of security. An HSM consists of hardware dedicated to centralized and controlled key management, offers significant protection against attack techniques, such as physical tampering and key extraction, and includes built-in random number generators for secure key generation. HSMs can be expensive to acquire and maintain, especially for small-scale applications. They require additional expertise to implement and can cause some latency in performance.

Key management software libraries allow customization of key management processes based on the specific needs of the application. The libraries can be integrated into the FPGA design, often making them more cost-effective than dedicated hardware solutions like HSMs. However, this method means that the security of the keys rely on the security of the software environment, making them susceptible to software-based attack techniques. Software libraries lack the physical protection against tampering and direct

attacks. Designers must pay careful attention to security practices to resist such attack techniques.

Secure element ICs can be costly and complex and may require specialized knowledge. Like other methods, using a secure IC can introduce latency due to the additional communication and computation. Secure ICs are only offered by specific vendors and may require additional hardware design.

4.6.2 Guidance

The use of a FIPS 140-3 compliant HSM is recommended. An HSM is designed to securely generate, store, and manage cryptographic keys, providing a dedicated, tamper-resistant environment for key management and protecting keys from unauthorized access or theft. Whether the program selects the use of an HSM or any of the other mechanisms discussed, the program should ensure they are following key management guidance found in NIST SP 800-57. This guidance provides guidelines for cryptographic key management, including key generation, distribution, storage, and retirement.

4.7 Program or individual keys

In a system with multiple FPGAs, the program must decide whether to use the same decryption key in all devices or unique keys per device. If the adversary is seeking to steal design or algorithm information, theft of a single key may be sufficient to attain their goal. If the adversary is seeking to subvert systems, they will need to steal/compromise keys for each device if unique keys are used.

4.7.1 Tradeoffs

Using the same secret key across multiple FPGAs simplifies key management practices, but can result in additional security concerns. For example, if a malicious actor can capture the key, they will then have access to all of the devices that use that same key. Using multiple keys increases key management complexities, but increases security. In the event that an attacker can gain access to one device, they may still need considerable work to attain keys for additional devices.

4.7.2 Guidance

If there are multiple FPGA types or FPGAs with varying functions, then the program should consider using different keys for each device or function. Possible exceptions to

this guidance include when the FPGAs are being used for non-sensitive or non-critical functions only.

4.8 Development tools and source code security

Every FPGA application development effort requires the use of specialized engineering tools. At a minimum, the following will be required:

- A text editor
- Revision control
- Implementation tools
- Synthesis
- Place and route
- Timing analysis
- Simulation tool
- Configuration generation tool
- Key generation tool
- Provisioning software

These tools can all be sourced from the FPGA vendor or as a collection of individual third-party software packages. In all cases, these tools represent an attack vector an adversary could use to compromise the program. When using these or any other tools, programs should ensure they are following the guidance outlined in the NSA document "CTR: DoD Microelectronics: Field Programmable Gate Array Level of Assurance 1 Best Practices."

4.9 Product end-of-life

Every product has a time at which it will become obsolete or end-of-life. In these cases, it is important to retire the FPGA-based system hardware in a manner that does not allow the leakage of sensitive information. Leaving an FPGA system in the field provides opportunity for an adversary to obtain physical access to devices for examination and study. This kind of access allows for the application of reverse

engineering techniques to extract secret keys, perform full design recovery of the FPGA and the government application, steal sensitive algorithms, and learn program security and assurance techniques. Ideally, a Program should take possession of end-of-life FPGA systems for proper and secure data erasure and hardware destruction. However, this is not always possible.

4.9.1 Guidance

The program should consider the following guidance for executing proper end-of-life steps to ensure the safety of their data:

- It is best to physically obtain the FPGA devices and PCBs, and
    - Delete all keys and configuration file data from all memory locations on the FPGA devices according to CNSS or NIST policy.
    - Once the data has been wiped, the devices should be destroyed in compliance with applicable DoD standards.
- In the event the devices cannot be obtained the program should:
    - Remotely erase all the keys and configuration data.
    - Utilize a built in function to wipe the data from the system.

5. References

- NIST SP 800-57 Part 1 Rev. 5: Recommendation for Key Management: Part 1 – General – Section 5.6.2. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf
- NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf
- NIST FIPS 140-3: Security Requirements for Cryptographic Modules. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.140-3.pdf
- NSA Announcing the Commercial National Security Algorithm Suite 2.0. https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF
- IETF RFC 9206 Commercial National Security Algorithm (CNSA) Suite Cryptography for Internet Protocol Security (IPsec). https://datatracker.ietf.org/doc/rfc9206/

- IETF RFC 9212 Commercial National Security Algorithm (CNSA) Suite Cryptography for Secure Shell (SSH). https://datatracker.ietf.org/doc/rfc9212/
- NSA CTR: DoD Microelectronics: Field Programmable Gate Array Level of Assurance 1 Best Practices. Available at https://www.nsa.gov/Press-Room/DoD Microelectronics Guidance/
- NIST SP 800-133 Rev. 2: Recommendation for Cryptographic Key Generation. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-133r2.pdf

6. Acronyms and abbreviations

AES	Advanced Encryption Standard
ASIC	Application specific integrated circuit
CBC	Cipher block chaining
CNSA	Commercial National Security Algorithm
CNSS	Committee for National Security Systems
CPU	Central processing unit
CRC	Cyclic Redundancy Check
CRQC	Crypt analytically relevant quantum computer
CTR	Counter mode
DFT	Design for test
ECC	Elliptic curve cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
eFUSE	Electronic FUSE
FIPS	Federal Information Processing Standards
FPGA	Field programmable gate array
FSBL	First-stage boot loader
GCM	Galois counter mode
GPIO	General purpose input/output
HMAC	Hash-based message authentication code
HSM	Hardware security module
HwA	Hardware assurance
I/O	Input/output
IC	Integrated circuit
ID	Identifier / identification
IP	Intellectual property
IV	Initialization vector
JTAG	Joint Test Action Group
KEK	Key encryption key
MAC	Message authentication code
MTP	Multiple time programmable
NIST	National Institute of Standards and Technology
NSS	National security system
NVM	Non-volatile memory

OTP	One-time programmable
PCB	Printed circuit board
PCI	Peripheral Component Interface
PL	Programmable logic
PR	Partial reconfiguration
PUF	Physical Unclonable function
QR Code	Quick response code
RNG	Random number generator
ROM	Read only memory
RSA	Rivest, Shamir, and Adleman - An asynchronous public/private key algorithm
SCA	Side channel attack
SEU	Single-event-upset
SOC	System on chip
SRAM	Static random access memory
USB	Universal serial bus
TPM	Trusted platform module