## Contents

1. **Intel® Agilex® Device Security Overview** ................................................................. 4  
   1.1. Commitment to Product Security ................................................................. 4  
   1.2. Planned Security Features ........................................................................ 5  
       1.2.1. Physical Anti-Tamper ........................................................................ 5  
       1.2.2. Black Key Provisioning .................................................................... 5  
       1.2.3. Intrinsic ID® Physically Unclonable Function (PUF) ......................... 5  
       1.2.4. Partial Reconfiguration Bitstream Security Verification .................. 5  

2. **Authentication and Authorization** ............................................................................ 6  
   2.1. Creating a Signature Chain ...................................................................... 6  
       2.1.1. Creating Authentication Key Pairs on the Local Filesystem ............ 7  
       2.1.2. Creating Authentication Key Pairs in SoftHSM ............................... 8  
       2.1.3. Creating the Signature Chain Root Entry ....................................... 9  
       2.1.4. Creating a Signature Chain Public Key Entry ................................. 9  
   2.2. Signing a Configuration Bitstream .......................................................... 10  
       2.2.1. Quartus Key File Assignment .......................................................... 10  
       2.2.2. Co-Signing SDM Firmware ............................................................... 11  
       2.2.3. Configuring Bitstream Signing Using the quartus_sign Command .... 12  
       2.2.4. Partial Reconfiguration Multi-Authority Support ............................ 13  
       2.2.5. Verifying Configuration Bitstream Signature Chains ..................... 15  

3. **AES Bitstream Encryption** ....................................................................................... 19  
   3.1. Creating the AES Root Key ....................................................................... 19  
   3.2. Quartus Encryption Settings ...................................................................... 19  
   3.3. Encrypting a Configuration Bitstream ...................................................... 20  
       3.3.1. Configuration Bitstream Encryption Using the Programming File Generator  
             Graphical Interface ........................................................................... 20  
       3.3.2. Configuration Bitstream Encryption Using the Programming File Generator  
             Command Line Interface ................................................................... 22  
       3.3.3. Partially Encrypted Configuration Bitstream Generation Using the  
             Command Line Interface ................................................................... 22  
       3.3.4. Partial Reconfiguration Bitstream Encryption .............................. 23  

4. **Device Provisioning** .............................................................................................. 25  
   4.1. Using SDM Provision Firmware ............................................................... 25  
   4.2. Authentication Root Key Provisioning .................................................. 25  
       4.2.1. Partial Reconfiguration Multi-Authority Root Key Programming .... 26  
   4.3. Programming Key Cancellation ID Fuses ............................................. 26  
   4.4. Canceling Root Keys ............................................................................. 27  
   4.5. Programming Counter Fuses ................................................................. 28  
   4.6. Secure Data Object Service Root Key Provisioning ............................ 28  
   4.7. Security Setting Fuse Provisioning ....................................................... 28  
   4.8. AES Root Key Provisioning .................................................................. 30  
       4.8.1. AES Root Key Compact Certificate .............................................. 30  
   4.9. Converting Owner Root Key, AES Root Key Certificates, and Fuse files to Jam  
        STAPL File Formats ............................................................................. 32  

5. **Advanced Features** ............................................................................................... 34  
   5.1. Secure Debug Authorization ................................................................... 34
1. Intel® Agilex® Device Security Overview


This document follows and assumes knowledge from the Security Methodology User Guide. This document contains instructions intended to help you use Intel Quartus® Prime Pro Edition software to implement security features on Intel Agilex devices.

This document organizes content as follows:

- **Authentication and Authorization**: Provides instructions to create authentication keys and signature chains, apply permissions and revocations, sign objects, and program authentication features on Anti-Tamper Lite Intel FPGA IP devices.
- **AES Bitstream Encryption**: Provides instructions to create an AES root key, encrypt configuration bitstreams, and provision the AES root key to Intel Agilex devices.
- **Device Provisioning**: Provides instructions to use the Intel Quartus Prime Programmer and Secure Device Manager (SDM) provision firmware to program security features on Intel Agilex devices.
- **Advanced Features**: Provides instructions to enable advanced security features, including secure debug authorization, Hard Processor System (HPS) debug, and remote system update.

1.1. Commitment to Product Security

Intel commits to engineering innovative security features into our products and to supporting and maintaining the security of our products. The security of our products remains an on-going priority. Intel strongly recommends that you become familiar with our product security resources and plan to utilize them throughout the life of your Intel product.

**Related Information**

- Product Security at Intel
- Intel Product Security Center Advisories
1.2. Planned Security Features

Features mentioned in this section are planned for a future release of Intel Quartus Prime Pro Edition software.

Note: The information in this section is preliminary.

1.2.1. Physical Anti-Tamper

Physical anti-tamper features help detect and respond to certain physical attacks on silicon.

The SDM monitors operating conditions such as input clocks, voltage, and temperature to detect device tampering. Changes in these conditions may indicate a tampering event. You can choose an appropriate response to a detected event. Possible responses include but are not limited to the following actions:

- Device lock
- Device lock with configuration data zeroization
- Device lock with configuration data zeroization and BBRAM key erasure

You enable the physical anti-tamper features during the design process. The Intel Agilex device activates the anti-tamper features as a configuration bitstream is loaded.

1.2.2. Black Key Provisioning

Black key provisioning helps to securely program the symmetric bitstream encryption AES key from an instance of the black key provisioning service executing in a trusted environment to an Intel Agilex device that is located in an untrusted environment.

1.2.3. Intrinsic ID® Physically Unclonable Function (PUF)

The Intrinsic ID® PUF utilizes uncontrollable deep-submicron manufacturing process variations to create an unclonable device fingerprint that can be used both for device identity purposes and wrapping the symmetric bitstream encryption AES key.

1.2.4. Partial Reconfiguration Bitstream Security Verification

Partial reconfiguration (PR) bitstream security validation helps provide additional assurance that PR persona bitstreams cannot access or interfere with other PR persona bitstreams.
2. Authentication and Authorization

To enable the authentication features of an Intel Agilex device, you begin by using the Intel Quartus Prime Pro Edition software and associated tools to build a signature chain. A signature chain consists of a root key, one or more signing keys, and applicable authorizations. You apply the signature chain to your Intel Quartus Prime Pro Edition project and compiled programming files. Use the instructions in Device Provisioning on page 25 to program your root key into Intel Agilex devices.

Related Information
Device Provisioning on page 25

2.1. Creating a Signature Chain

You may use the quartus_sign tool or the agilex_sign.py reference implementation to perform signature chain operations. This document provides examples using quartus_sign.

To use the reference implementation, you substitute a call to the Python interpreter included with Intel Quartus Prime and omit the --family=agilex option; all other options are equivalent. For example, the quartus_sign command found later in this section

```
quartus_sign --family=agilex --operation=make_root root_public.pem root.qky
```

can be converted into the equivalent call to the reference implementation as follows

```
pgm_py agilex_sign.py --operation=make_root root_public.pem root.qky
```

Intel Quartus Prime Pro Edition software includes the quartus_sign, pgm_py, and agilex_sign.py tools. You may use the Nios® II command shell tool, which automatically sets appropriate environment variables, to access the tools.

You use the following instructions to bring up a Nios II command shell.

1. Bring up a Nios II command shell.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>On the Start menu, point to Programs ➤ Intel FPGA ➤ Nios II EDS ➤ &lt;version&gt; and click Nios II &lt;version&gt; Command Shell.</td>
</tr>
<tr>
<td>Linux</td>
<td>In a command shell change to the &lt;install_dir&gt;/nios2eds and run the following command: ./nios2_command_shell.sh</td>
</tr>
</tbody>
</table>

The examples in this section assume signature chain and configuration bitstream files are located in the current working directory. If you choose to follow the examples where key files are kept on the filesystem, those examples assume the key files are
located in the current working directory. You may choose which directories to use, and
the tools support relative file paths. If you choose to keep key files on the filesystem,
you must carefully manage access permissions to those files.

Intel recommends the use of a commercially available Hardware Security Module
(HSM) to store cryptographic keys and perform cryptographic operations. The
quartus_sign tool and reference implementation include a Public Key Cryptography
Standard #11 (PKCS #11) Application Programming Interface (API) to interact with an
HSM while performing signature chain operations. The agilex_sign.py reference
implementation includes an interface abstract as well as an example interface to
SoftHSM.

You may use these example interfaces to implement an interface to your HSM. Refer
to the documentation from your HSM vendor for more information about implementing
an interface to and operating your HSM.

SoftHSM is a software implementation of a generic cryptographic device with a PKCS
#11 interface that is made available by the OpenDNSSEC® project. You may find more
information, including instructions on how to download, build, and install OpenHSM, at
the OpenDNSSEC project. The examples in this section utilize SoftHSM version 2.6.1.
The examples in this section additionally use the pkcs11-tool utility from OpenSC to
perform additional PKCS #11 operations with a SoftHSM token. You may find more
information, including instructions on how to download, build, and install
pkcs11-tool from OpenSC.

Related Information
- The OpenDNSSEC project
  Policy-based zone signer for automating the process of DNSSEC keys tracking.
- SoftHSM
  Information about the implementation of a cryptographic store accessible
  through a PKCS #11 interface.
- OpenSC
  Provides set of libraries and utilities able to work with smart cards.

2.1.1. Creating Authentication Key Pairs on the Local Filesystem

You use the quartus_sign tool to create authentication key pairs on the local
filesystem using the make_private_pem and make_public_pem tool operations.
You first use the make_private_pem operation to generate a private key. You specify
the elliptic curve to use, the private key filename, and optionally whether to protect
the private key with a passphrase. Intel recommends the use of the secp384r1 curve
and following industry best practices to create a strong, random passphrase on all
private key files. Intel also recommends restricting the file system permissions on the
private key .pem files to read by owner only. You use the make_public_pem
operation to derive the public key from the private key. It is helpful to name the
key .pem files descriptively. This document generally uses the convention
$keyuse<$cancelID>_<keytype>.pem in the following examples.

1. In the Nios II command shell, run the following command to create a private key.
The private key, shown below, is used as the root key in later examples that
create a signature chain. Intel Agilex devices support multiple root keys, so you
repeat this step to create your required number of root keys. Examples in this document all refer to the first root key, though you may build signature chains in a similar fashion with any root key.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| With passphrase | `quartus_sign --family=agilex --operation=make_private_pem \
|                 |  --curve=secp384r1 root0_private.pem` Enter the passphrase when prompted to do so. |
| Without passphrase | `quartus_sign --family=agilex --operation=make_private_pem \
|                   |  --curve=secp384r1 --no_passphrase root0_private.pem`                  |

2. Run the following command to create a public key using the private key generated in the previous step. You do not need to protect the confidentiality of a public key.

`quartus_sign --family=agilex --operation=make_public_pem \
root0_private.pem root0_public.pem`

3. Run the commands again to create a key pair used as the design signing key in the signature chain.

`quartus_sign --family=agilex --operation=make_private_pem \
--curve=secp384r1 design0_sign_private.pem`

`quartus_sign --family=agilex --operation=make_public_pem \
design0_sign_private.pem design0_sign_public.pem`

### 2.1.2. Creating Authentication Key Pairs in SoftHSM

The SoftHSM examples in this chapter are self-consistent. Certain parameters depend on your SoftHSM installation and a token initialization within SoftHSM.

The `quartus_sign` tool depends on the PKCS #11 API library from your HSM. The examples in this section assume that the SoftHSM library is installed to `/usr/local/lib/softhsm2.so` on Linux or `C:\SoftHSM2\lib\softhsm2.dll` on 32-bit version of Windows or `C:\SoftHSM2\lib\softhsm2-x64.dll` on 64-bit version of Windows.

You initialize a token within SoftHSM using the `softhsm2-util` tool.

`softhsm2-util --init-token --label agilex-token --pin agilex-token-pin \
--so-pin agilex-so-pin --free`

The option parameters, particularly the token `label` and token `pin` are examples used throughout this chapter. Intel recommends that you follow instructions from your HSM vendor to create and manage tokens and keys.

You create authentication key pairs using the `pkcs11-tool` utility to interact with the token in SoftHSM. Instead of explicitly referring to the private and public key `.pem` files in the filesystem examples, you refer to the key pair by its label and the tool selects the appropriate key automatically.
Run the following commands to create a key pair used as the root key in later examples as well as a key pair used as a design signing key in the signature chain.

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label agilex-token --login --pin agilex-token-pin --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp384r1 --usage-sign \
--label root0 --id 0
```

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label agilex-token --login --pin agilex-token-pin --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp384r1 --usage-sign \
--label design0_sign --id 1
```

**Note:** The ID option in this step must be unique to each key, but it is used only by the HSM. This ID option is unrelated to the key cancellation ID assigned in the signature chain.

### 2.1.3. Creating the Signature Chain Root Entry

You use the `make_root` operation to convert the root public key into a signature chain root entry, stored on the local filesystem in the Intel Quartus Prime key (.qky) format file. Repeat this step for each root key you generate.

Run the following command to create a signature chain with a root entry, using a root public key from the file system.

```
quartus_sign --family=agilex --operation=make_root \
--key_type=owner root0_public.pem root0.qky
```

Run the following command to create a signature chain with a root entry, using the root key from the SoftHSM token established in the prior section.

```
quartus_sign --family=agilex --operation=make_root --key_type=owner \
--module=softHSM --module_args="--token_label=agilex-token \
--user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" root0 root0.qky
```

### 2.1.4. Creating a Signature Chain Public Key Entry

You use the `append_key` operation to create a new public key entry for a signature chain. You specify the prior signature chain, the private key for the last entry in the prior signature chain, the next level public key, the permissions and cancellation ID you assign to the next level public key, and the new signature chain file.

Depending on your use of keys on the filesystem or in an HSM, you use one of the following example commands to append the `design0_sign` public key to the root signature chain created in the prior section.

```
quartus_sign --family=agilex --operation=append_key \
--previous_pem=root0_private.pem --previous_qky=root0.qky \
--permission=6 --cancel=0 design0_sign_public.pem \
design0_sign_chain.qky
```

```
quartus_sign --family=agilex --operation=append_key --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root0 --previous_qky=root0.qky \
--permission=6 --cancel=0 design0_sign design0_sign_chain.qky
```
You may repeat the append_key operation up to two more times for a maximum of three public key entries between the root entry and header block entry in any one signature chain.

The following example assumes you created another authentication public key with the same permissions and assigned cancellation ID 1 called design1_sign_public.pem, and are appending this key to the signature chain from the previous example.

```
quartus_sign --family=agilex --operation=append_key \
--previous_pem=design0_sign_private.pem \
--previous_qky=design0_sign_chain.qky \
--permission=6 \
--cancel=1 design1_sign_public.pem design1_sign_chain.qky
```

```
quartus_sign --family=agilex --operation=append_key --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=design0_sign \
--previous_qky=design0_sign_chain.qky \
--permission=6 \
--cancel=1 design1_sign design1_sign_chain.qky
```

Intel Agilex devices support an additional key cancellation counter to facilitate the use of a key that may change periodically throughout the life of a given device. You may select this key cancellation counter by changing the argument of the --cancel option to pts:pts_value.

2.2. Signing a Configuration Bitstream

Intel Agilex devices support Security Version Number (SVN) counters, which allow you to revoke the authorization of an object without canceling a key. You assign the SVN counter and the appropriate SVN counter value during the signing of any object, such as a bitstream section, firmware .zip file, or compact certificate. You assign the SVN counter and SVN value using the --cancel option and svn_counter:svn_value as the argument. Valid values for svn_counter are svnA, svnB, svnC, and svnD; svn_value is an integer in the range [0,63].

2.2.1. Quartus Key File Assignment

You specify a signature chain in your Intel Quartus Prime project to enable the authentication feature for that design. From the Assignments menu, select Device ➤ Device and Pin Options ➤ Security ➤ Quartus Key File, then browse to the signature chain .qky file you created to sign this design.
Alternatively, you may add the following assignment statement to your Intel Quartus Prime Settings file (.qsf):

```
set_global_assignment -name QKY_FILE design0_sign_chain.qky
```

To generate a .sof file from a previously compiled design, that includes this setting, from the Processing menu, select Start ➤ Start Assembler. The new output .sof file includes the assignments to enable authentication with the provided signature chain.

### 2.2.2. Co-Signing SDM Firmware

You use the quartus_sign tool to extract, sign, and install the applicable SDM firmware .zip file. The co-signed firmware is then included by the programming file generator tool when you convert .sof file into a configuration bitstream .rbf file. You use the following commands to create a new signature chain and sign SDM firmware.

1. Create a new signing key pair.
a. Create a new signing key pair on the file system.

```bash
quartus_sign --family=agilex --operation=make_private_pem \
--curve=secp384r1 firmware1_private.pem

quartus_sign --family=agilex --operation=make_public_pem \
firmware1_private.pem firmware1_public.pem
```

b. Create a new signing key pair in the HSM.

```bash
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label=agilex-token --login --pin=agilex-token-pin \
--keypairgen --mechanism=ECDSA-KEY-PAIR-GEN --key-type=EC:secp384r1 \
--usage=sig --label=firmware1 --id=1
```

2. Create a new signature chain containing the new public key.

```bash
quartus_sign --family=agilex --operation=append_key \
--previous_pem=root0_private.pem --previous_qky=root0.qky \
--permission=0x1 --cancel=1 \nfirmware1_public.pem firmware1_sign_chain.qky

quartus_sign --family=agilex --operation=append_key \
--module=softHSM --module_args="--token_label=agilex-token \
--user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root0 --previous_qky=root0.qky \
--permission=1 --cancel=1 firmware1 firmware1_sign_chain.qky
```

3. Copy the firmware .zip file from your Intel Quartus Prime Pro Edition software installation directory (<install_dir>/quartus/common/devinfo/programmer/firmware/agilex.zip) to the current working directory.

```bash
quartus_sign --family=agilex --get_firmware=.
```

4. Sign the firmware .zip file. The tool automatically unpacks the .zip file and individually signs all firmware .cmf files, then rebuilds the .zip file for use by the tools in the following sections.

```bash
quartus_sign --family=agilex --operation=sign \
--qky=firmware1_sign_chain.qky \
--cancel=svnA:0 --pem=firmware1_private.pem agilex.zip signed_agilex.zip

quartus_sign --family=agilex --operation=sign --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--pem=firmware1 --cancel=svnA:0 \
--qky=firmware1_sign_chain.qky agilex.zip signed_agilex.zip
```

### 2.2.3. Configuring Bitstream Signing Using the `quartus_sign` Command

To sign a configuration bitstream using the `quartus_sign` command, you first convert the .sof file to the unsigned raw binary file (.rbf) format. You may optionally specify co-signed firmware using the `fw_source` option during the conversion step.

You can generate the unsigned raw bitstream in .rbf format using the following command:

```bash
quartus_pfg -c -o fw_source=signed_agilex.zip -o signLater=ON \
design.sof unsigned_bitstream.rbf
```
Run one of the following commands to sign the bitstream using the `quartus_sign` tool depending on the location of your keys:

```
quartus_sign --family=agilex--operation=sign \
--qky=design0_sign_chain.qky --pem=design0_sign_private.pem \
--cancel=svnA:0 unsigned_bitstream.rbf signed_bitstream.rbf
```

```
quartus_sign --family=agilex--operation=sign --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=design0_sign \
--qky=design0_sign_chain.qky \
--cancel=svnA:0 unsigned_bitstream.rbf signed_bitstream.rbf
```

You may convert signed .rbf files to other configuration bitstream file formats.

For example, if you are using the Jam* Standard Test and Programming Language (STAPL) Player to program a bitstream over JTAG, you use the following command to convert an .rbf file to the .jam format that the Jam STAPL Player requires:

```
quartus_pfg -c signed_bitstream.rbf signed_bitstream.jam
```

### 2.2.4. Partial Reconfiguration Multi-Authority Support

Intel Agilex devices support partial reconfiguration multi-authority authentication, where the device owner creates and signs the static bitstream, and a separate PR owner creates and signs PR persona bitstreams. Intel Agilex devices implement multi-authority support by assigning the first authentication root key slots to the device or static bitstream owner, and assigning the final authentication root key slot to the partial reconfiguration persona bitstream owner.

**Note:** Partial Reconfiguration static and persona bitstream encryption when multi-authority support is enabled is planned in a future release.

Implementing partial reconfiguration multi-authority support requires several steps.

1. The device or static bitstream owner generates one or more authentication root keys as described in Creating Authentication Key Pairs in SoftHSM on page 8, where the `--key_type` option has value `owner`.

2. The partial reconfiguration bitstream owner generates an authentication root key, but changes the `--key_type` option value to `secondary_owner`.

3. Both the static bitstream and partial reconfiguration design owners ensure that the Enable Multi-Authority support checkbox is enabled in the Assignments ➤ Device ➤ Device and Pin Options ➤ Security tab.
4. Both the static bitstream and partial reconfiguration design owners create signature chains based on their respective root keys as described in Creating a Signature Chain on page 6.

5. Both the static bitstream and partial reconfiguration design owners convert their compiled designs to `.rbf` format files and sign the `.rbf` files.

6. The device or static bitstream owner generates and signs a PR public key program authorization compact certificate.

    ```bash
    quartus_pfg --ccert -o ccert_type=PR_PUBKEY_PROG_AUTH \
    -o owner_qky_file="root0.qky;root1.qky" unsigned_pr_pubkey_prog.ccert
    quartus_sign --family=agilex --operation=sign \ 
    --qky=design0_sign_chain.qky --pem=design0_sign_private.pem \ 
    --cancel=svnA:0 unsigned_pr_pubkey_prog.ccert signed_pr_pubkey_prog.ccert
    quartus_sign --family=agilex --operation=sign --module=softHSM \ 
    --module_args="--token_label=s10-token --user_pin=s10-token-pin \ 
    --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \ 
    --pem=design0_sign --qky=design0_sign_chain.qky \ 
    --cancel=svnA:0 unsigned_pr_pubkey_prog.ccert signed_pr_pubkey_prog.ccert
    ```

7. The device or static bitstream owner provisions their authentication root key hashes to the device, then programs the PR public key program authorization compact certificate, and finally provisions the partial reconfiguration bitstream owner root key to the device. This provisioning process is described in detail in Device Provisioning on page 25.

8. Intel Agilex device is configured with the static region `.rbf` file.

9. Intel Agilex device is partially reconfigured with the persona design `.rbf` file.

**Related Information**

- Creating a Signature Chain on page 6
- Creating Authentication Key Pairs in SoftHSM on page 8
- Device Provisioning on page 25
2.2.5. Verifying Configuration Bitstream Signature Chains

After you create signature chains and signed bitstreams, you may verify that a signed bitstream correctly configures a device programmed with a given root key. You first use the fuse_info operation of the quartus_sign command to print the hash of the root public key to a text file:

```
quartus_sign --family=agilex --operation=fuse_info root0.qky hash_fuse.txt
```

You then use the check_integrity option of the quartus_pfg command to inspect the signature chain on each section of a signed bitstream in .rbf format. The check_integrity option prints the status of the overall bitstream integrity check, the contents of each entry in each signature chain attached to each section in the bitstream .rbf file, and the expected fuse value for the hash of the root public key for each signature chain. The value from the fuse_info output should match the Fuse lines in the check_integrity output.

```
quartus_pfg --check_integrity signed_bitstream.rbf
```

Here is an example of the check_integrity command output:

```plaintext
Info: Command: quartus_pfg --check_integrity signed_bitstream.rbf
Integrity status: OK

Section
Type: CMF
Signature Descriptor ...
Signature chain #0 (entries: -1, offset: 96)
Entry #0
Fuse: 34FD3B5F 7829001F 3A7EAE29 C7786DB1 D65BC3C 52741C79 72978B22 0731B082 6F596899 40F32048 AD766A24
Generate key ...
Curve : secp384r1
X     : 29C39C0364AE594A36DA85602D6AF0B278CBBB0B270C4D97C9F6B967961E5F0ECA
Y     : 3E81D0CBBBECAD360124A79D53F4A831308A24CA0BDDFA40351EE76438C7B5D2
2826F7E94A169023AFAE1D1DF4A31C2
Generate key ...
Curve : secp384r1
X     : 29C39C0364AE594A36DA85602D6AF0B278CBBB0B270C4D97C9F6B967961E5F0ECA
Y     : 3E81D0CBBBECAD360124A79D53F4A831308A24CA0BDDFA40351EE76438C7B5D2
2826F7E94A169023AFAE1D1DF4A31C2
Entry #1
Generate key ...
Curve : secp384r1
X     : 015290C556F1533E5631322953E2F9E91258472F43EC954E05D6A4B63D611E04B
C120C7E7A744357346B424052100A9
Y     : 68696DEAC4773FF3D5A16A42619754244AB4248196CF5142858E016242FB828
08A80F3F3F156D5F0A5959957BE505
Entry #2
Keychain permission: SIGN_CODE
Keychain can be cancelled by ID: 3
Signature chain #1 (entries: -1, offset: 648)
Entry #0
Fuse: FA6528B8 9281F2DB B787E805 6BF6EEOE 28983C56 D56BB141 8EE4EF6
DAC2D422 0A30A27 81EFC6CD 67E973BF AC286EAE
Generate key ...
Curve : secp384r1
X     : 47A453747A4B886A8B5615EB1AB88A75BAC90C46E564CB5B5DC91328244E765
0411C49527FACC71DE36A105B0478
Y     : 6087D3B45C86464B4DACC65B863CD0E705BD0C92141DE4E7BDDEB85C0410D8
```
**2. Authentication and Authorization**

**Intel® Agilex™ Device Security User Guide**

---

**Keychain permission:** SIGN_CODE

Keychain can be cancelled by ID: 15

Signature chain #2 (entries: -1, offset: 0)
Signature chain #3 (entries: -1, offset: 0)
Signature chain #4 (entries: -1, offset: 0)
Signature chain #5 (entries: -1, offset: 0)
Signature chain #6 (entries: -1, offset: 0)
Signature chain #7 (entries: -1, offset: 0)

Section
Type: IO
Signature Descriptor ...
Signature chain #0 (entries: -1, offset: 96)

Entry #0
Fuse: FA6528BE 9281F2DB B787E805 6BF6E0E0 28983C56 D568B141 8EEE4BF6 DACC2D42 0A3A0F27 81ECFC6C 67E973BF AC286EAE

Generate key ...
Curve : secp384r1
X : 646B51F66D88CC36D572B899BA082FD7E79B00CDB750DA0C984DC5891CDF57BD21 44758CA747B1AB8315024A24427F25E51
Y : 53513111E21E151FD55D7ECD8293AF6C98A74D52E0DA2527948A64FABDF7E7C F4EA8BB22921BD38A969EE15947E570

Entry #1
Generate key ...
Curve : secp384r1
X : 646B51F66D88CC36D572B899BA082FD7E79B00CDB750DA0C984DC5891CDF57BD21 44758CA747B1AB8315024A24427F25E51
Y : 53513111E21E151FD55D7ECD8293AF6C98A74D52E0DA2527948A64FABDF7E7C F4EA8BB22921BD38A969EE15947E570

Entry #2
Generate key ...
Curve : secp384r1
X : 13986DDECAB697A2EB26B8EBD25095A8C28B1A0B0C766D029CD2F2A21BE3432 76896E7719AC8A6EC8CDB30CF4CB83C
Y : 0A13B4E90D20238FF10D86B557414955354EE6681D55309A507A78FC05A1 49F91CABA72F6A3A1C2D1990CDAEA3D
Entry #3
Keychain permission: SIGN_CORE
Keychain can be cancelled by ID: 15
Signature chain #1 (entries: -1, offset: 0)
Signature chain #2 (entries: -1, offset: 0)
Signature chain #3 (entries: -1, offset: 0)
Signature chain #4 (entries: -1, offset: 0)
Signature chain #5 (entries: -1, offset: 0)
Signature chain #6 (entries: -1, offset: 0)
Signature chain #7 (entries: -1, offset: 0)

Section
Type: HPS
Signature Descriptor ...
Signature chain #0 (entries: -1, offset: 96)

Entry #0
Fuse: FA6528BE 9281F2DB B787E805 6BF6EE0E 28983C56 D568B141 8EEE4BF6
DAC2D42 0A3A0F27 81EFC6CD 67E973BF AC286EAE
Generate key ...
Curve : secp384r1
X     : 47A453474A8D886A8B058615EB1A8B38A75BAC9F0C46E564CB55DC1328244E765 0411C4592FAFCC71DE36A105B054781
Y     : 6087DB4A5C8646B4DAC6B5C863CD8E705BD09C2141DE4DE78DEBE85C0410D8 6B7312EE82411894742629501FCD

Entry #1
Generate key ...
Curve : secp384r1
X     : 47A453474A8D886A8B058615EB1A8B38A75BAC9F0C46E564CB55DC1328244E765 0411C4592FAFCC71DE36A105B054781
Y     : 6087DB4A5C8646B4DAC6B5C863CD8E705BD09C2141DE4DE78DEBE85C0410D8 6B7312EE82411894742629501FCD

Entry #2
Generate key ...
Curve : secp384r1
X     : 1396DDDCAB6972EB268EB82D5095A8CC2BA0A0B0C766D029CDF2A2FE21BE3432 76896771A9C6C9A5A2D3C08CF4C8B3C
Y     : 0A1384ED2D9238FF11D867B55714955354EE6681D553509A57A787CFC50A51 49F91CABA72F6A3A1C2D990CDAE3AD

Entry #3
Keychain permission: SIGN_HPS
Keychain can be cancelled by ID: 15
Signature chain #1 (entries: -1, offset: 0)
Signature chain #2 (entries: -1, offset: 0)
Signature chain #3 (entries: -1, offset: 0)
Signature chain #4 (entries: -1, offset: 0)
Signature chain #5 (entries: -1, offset: 0)
Signature chain #6 (entries: -1, offset: 0)
Signature chain #7 (entries: -1, offset: 0)

Section
Type: CORE
Signature Descriptor ...
Signature chain #0 (entries: -1, offset: 96)

Entry #0
Fuse: FA6528BE 9281F2DB B787E805 6BF6EE0E 28983C56 D568B141 8EEE4BF6
DAC2D42 0A3A0F27 81EFC6CD 67E973BF AC286EAE
Generate key ...
Curve : secp384r1
X     : 47A453474A8D886A8B058615EB1A8B38A75BAC9F0C46E564CB55DC1328244E765 0411C4592FAFCC71DE36A105B054781
Y     : 6087DB4A5C8646B4DAC6B5C863CD8E705BD09C2141DE4DE78DEBE85C0410D8 6B7312EE82411894742629501FCD
<table>
<thead>
<tr>
<th>Entry #1</th>
<th>Generate key ...</th>
<th>Curve : secp384r1</th>
<th>X : 47A53474A8D886A058615E81A838A75BAC9F0C46E564CB55DC1328244E7650411C4592FAFFC71DE36A105B054781</th>
<th>Y : 608703B4A5C8646B4DACC6B5C863C0DE705BD0C9D2C141DE4DE7BDEB85C0410D86B7312EEE8241189474262629501FCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry #2</td>
<td>Generate key ...</td>
<td>Curve : secp384r1</td>
<td>X : 646B51F668D8CC365D72B889B8082FDE79B00CDB750DA0C984DC5891CDF57BD2144758CA747B1A8315024A8247F12E51</td>
<td>Y : 5351311882E161515D5D7ECDE8293AF6C98A74D52E0DA2527948A64FABDFE7C4FEA8B8E22921BD38A869EE15476750</td>
</tr>
<tr>
<td>Entry #3</td>
<td>Keychain permission: SIGN_CORE</td>
<td>Keychain can be cancelled by ID: 15</td>
<td>Signature chain #1 (entries: -1, offset: 0)</td>
<td>Signature chain #2 (entries: -1, offset: 0)</td>
</tr>
</tbody>
</table>
3. AES Bitstream Encryption

3.1. Creating the AES Root Key

You may use the quartus_encrypt tool or stratix10_encrypt.py reference implementation to create an Advanced Encryption Standard (AES) root key in the Quartus encryption key (.qek) format file.

You may optionally specify the base key used to derive the AES root key and key derivation key, the value for the AES root key directly, the number of intermediate keys, and the maximum use per intermediate key. You must specify the device family, output .qek file location, and passphrase when prompted. Run the following command to generate the AES root key using random data for the base key and default values for number of intermediate keys and maximum key use.

To use the reference implementation, you substitute a call to the Python interpreter included with Intel Quartus Prime and omit the --family=agilex option; all other options are equivalent. For example, the quartus_encrypt command found later in the section

```
quartus_encrypt --family=agilex --operation=MAKE_AES_KEY aes_root.qek
```

can be converted into the equivalent call to the reference implementation as follows

```
pgm_py stratix10_encrypt.py --operation=MAKE_AES_KEY aes_root.qek
```

3.2. Quartus Encryption Settings

To enable bitstream encryption for a design, you must specify the appropriate options using the Assignments ➤ Device ➤ Device and Pin Options ➤ Security panel. You select the Enable configuration bitstream encryption checkbox, and the desired Encryption key storage location using the dropdown menu.

*Other names and brands may be claimed as the property of others.*
If you want to enable additional mitigations against side-channel attack vectors, you may enable the **Encryption update ratio** dropdown and **Enable scrambling** checkbox.

### 3.3. Encrypting a Configuration Bitstream

You encrypt a configuration bitstream prior to signing the bitstream. The Intel Quartus Prime Programming File Generator tool can automatically encrypt and sign a configuration bitstream using the graphical user interface or command line. You may optionally create a partially encrypted bitstream for use with the `quartus_encrypt` and `quartus_sign` tools or reference implementation equivalents.

#### 3.3.1. Configuration Bitstream Encryption Using the Programming File Generator Graphical Interface

You can use the Programming File Generator to encrypt and sign the owner image.

1. On the Intel Quartus Prime File menu select **Programming File Generator**.
2. On the **Output Files** tab, specify the output file type for your configuration scheme.
3. On the **Input Files** tab, click **Add Bitstream** and browse to your .sof.

4. To specify encryption and authentication options select the .sof and click **Properties**.
   a. Turn **Enable signing tool** on.
   b. For **Private key file** select your signing key private .pem file.
   c. Turn **Finalize encryption** on.
   d. For **Encryption key file**, select your AES .qek file.
To generate the signed and encrypted bitstream, on the Input Files tab, click Generate. Password dialog boxes appear for you to input your passphrase for your AES key .qek file and signing private key .pem file. The programming file generator creates the encrypted and signed output_file.rbf.

### 3.3.2. Configuration Bitstream Encryption Using the Programming File Generator Command Line Interface

You use the quartus_pfg command line interface to generate an encrypted and signed configuration bitstream in .rbf format.

```bash
quartus_pfg -c encryption_enabled.sof top.rbf -o finalize_encryption=ON \ -o qek_file=aes_root.qek -o signing=ON -o pem_file=design0_sign_private.pem
```

You may convert an encrypted and signed configuration bitstream in .rbf format to other configuration bitstream file formats.

### 3.3.3. Partially Encrypted Configuration Bitstream Generation Using the Command Line Interface

You may generate a partially encrypted programming file to finalize encryption and sign the image at a later time. You use the quartus_pfg command line interface to generate the partially encrypted programming file in the .rbf format.

```bash
quartus_pfg -c -o finalize_encryption_later=ON \ -o sign_later=ON top.sof top.rbf
```

You use the quartus_encrypt command line tool to finalize bitstream encryption.

```bash
quartus_encrypt --family=agilex \ --operation=ENCRYPT --key=aes_root.qek top.rbf encrypted_top.rbf
```
You use the `quartus_sign` command line tool to sign the encrypted configuration bitstream.

```
quartus_sign --family=agilex --operation=SIGN --qky=design0_sign_chain.qky --pem=design0_sign_private.pem --cancel=svnA:0 encrypted_top.rbf signed_encrypte_top.rbf
```

```
quartus_sign --family=agilex --operation=sign --module=softHSM --module_args="--token_label=agilex-token --user_pin=agilex-token-pin --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=design0_sign --qky=design0_sign_chain.qky --cancel=svnA:0 encrypted_top.rbf signed_encrypte_top.rbf
```

### 3.3.4. Partial Reconfiguration Bitstream Encryption

You can enable bitstream encryption on some Intel Agilex FPGA designs that use partial reconfiguration.

Partial reconfiguration designs utilizing the Hierarchical Partial Reconfiguration (HPR) or Static Update Partial Reconfiguration (SUPR) do not support the bitstream encryption. You must use the same encryption key to encrypt the static region and all personas unless multi-authority support is enabled. If your design contains multiple PR regions, you must encrypt all personas. To enable partial reconfiguration bitstream encryption, follow the same procedure in all design revisions.

You enable partial reconfiguration bitstream encryption by selecting **Assignments ➤ Device ➤ Device and Pin Options ➤ Security** menu. Select the desired encryption key storage location.

![Partial Reconfiguration Bitstream Encryption Setting](image)

After you compile your base design and revisions, the software generates a `.sof` file and one or more `.pmsf` files, representing the personas. You create encrypted and signed programming files from `.sof` and `.pmsf` files in a similar fashion to designs with no partial reconfiguration enabled.
Use the following command to convert the compiled persona .pmsf file to a partially encrypted .rbf file:

```
quartus_pfg -c -o finalize_encryptionLater=ON \
-o signLater=ON encryption_enabled_persona1.pmsf persona1.rbf
```

You use the `quartus_encrypt` command line tool to finalize bitstream encryption.

```
quartus_encrypt --family=agilex \ 
--operation=ENCRYPT --key=aes_root.qek persona1.rbf encrypted_persona1.rbf
```

You use the `quartus_sign` command line tool to sign the encrypted configuration bitstream.

```
quartus_sign --family=agilex --operation=SIGN \ 
--qky=design0_sign_chain.qky \ 
--pem=design0_sign_private.pem encrypted_persona1.rbf \ 
signed_encrypted_persona1.rbf
```

```
quartus_sign --family=agilex--operation=SIGN \ 
--module=softHSM --module_args="--token_label=agilex-token \ 
--user_pin=agilex-token-pin --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \ 
--qky=design0_sign_chain.qky --cancel=svnA:0 \ 
--pem=design0_sign encrypted_persona1.rbf signed_encrypted_persona1.rbf
```
4. Device Provisioning

Initial security feature provisioning is only supported in the SDM provision firmware. Use the Intel Quartus Prime Programmer to load the SDM provision firmware and perform provisioning operations.

4.1. Using SDM Provision Firmware

The Intel Quartus Prime Programmer automatically loads the provision firmware when the initialize operation is selected and the command performs actions that require the provision firmware, such as programming the authentication root key hash, security setting fuses, PUF enrollment, or black key provisioning.

You may alternately create a firmware-only helper image using the Quartus Programming File Generator command line tool. You specify your device type, the provision subtype, and optionally a co-signed firmware zip file.

```
quartus_pfg --helper_image -o helper_device=AGFB014R24A -o subtype=PROVISION -o fw_source=signed_agilex.zip signed_provision_helper_image.rbf
```

You use the Intel Quartus Prime Programmer tool to program the helper image.

```
quartus_pgm -c 1 -m jtag -o "p;signed_provision_helper_image.rbf" --force
```

You may omit the initialize operation from examples provided in this chapter if you have already programmed a provision helper image.

If you plan to use firmware co-signing, you may use a co-signed helper image on an unprovisioned device as the unprovisioned device ignores non-Intel signature chains over SDM firmware.

4.2. Authentication Root Key Provisioning

To program the owner root key hashes to physical fuses, you must load the provision firmware first following a power-on reset, program the owner root key hashes, and immediately perform another power-on reset. A power-on reset is not required if programming root key hashes to virtual fuses.

To program authentication root key hashes, you program the provision firmware helper image and run one of the following commands to program the root key .qky files.

```
// For physical (non-volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;root0.qky;root1.qky;root2.qky" --non_volatile_key

// For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;root0.qky;root1.qky;root2.qky"
```
4.2.1. Partial Reconfiguration Multi-Authority Root Key Programming

After provisioning the device or static region bitstream owner root keys, you again load the device provision helper image, program the signed PR public key program authorization compact certificate, and then provision the PR persona bitstream owner root key.

```bash
quartus_pgm -c 1 -m jtag -o "p;signed_pr_pubkey_prog.ccert"
quartus_pgm -c 1 -m jtag -o "p;root_pr.qky" --pr_pubkey <--non_volatile_key>
```

4.3. Programming Key Cancellation ID Fuses

Starting with Intel Quartus Prime Pro Edition software version 21.1, programming Intel and owner key cancellation ID fuses requires the use of a signed compact certificate. The key cancellation ID compact certificate may be signed with a signature chain that has FPGA section signing permissions. You create the compact certificate with the programming file generator command line tool. You sign the unsigned certificate using the `quartus_sign` tool or reference implementation.

Intel Agilex devices support separate banks of owner key cancellation IDs for each root key. When an owner key cancellation ID compact certificate is programmed into an Intel Agilex FPGA, the SDM determines which root key signed the compact certificate and blows the key cancellation ID fuse that corresponds to that root key.

The following examples create an Intel key cancellation certificate for Intel key ID 7. You may replace 7 with the applicable Intel key cancellation ID from 0-31.

Run the following command to create an unsigned Intel key cancellation ID compact certificate.

```bash
quartus_pfg --ccert -o ccert_type=CANCEL_INTEL_KEY -o cancel_key=7 
unsigned_cancel_intel7.ccert
```

Run one of the following commands to sign the unsigned Intel key cancellation ID compact certificate.

```bash
quartus_sign --family=agilex--operation=SIGN 
--qky=design0_sign_chain.qky --pem=design0_private.pem 
--cancel=svnA:0 unsigned_cancel_intel7.ccert signed_cancel_intel7.ccert
```

Run the following command to create an unsigned owner key cancellation ID compact certificate.

```bash
quartus_pfg --ccert -o ccert_type=CANCEL_OWNER_KEY -o cancel_key=2 
unsigned_cancel_owner2.ccert
```
Run one of the following commands to sign the unsigned owner key cancellation ID compact certificate.

```bash
quartus_sign --family=agilex --operation=SIGN \
--qky=design0_sign_chain.qky --pem=design0_private.pem \
--cancel=svnA:0 unsigned_cancel_owner2.ccert signed_cancel_owner2.ccert

quartus_sign --family=agilex --operation=SIGN --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=design0_sign \
--qky=design0_sign_chain.qky --cancel=svnA:0 \
unsigned_cancel_owner2.ccert signed_cancel_owner2.ccert
```

Once you have created a signed key cancellation ID compact certificate, you use the Intel Quartus Prime Programmer to program the compact certificate to the device via JTAG.

//For physical (non-volatile) eFuses
quartus_pgm -c 1 -m jtag -o "pi;signed_cancel_intel7.ccert" --non_volatile_key
quartus_pgm -c 1 -m jtag -o "pi;signed_cancel_owner2.ccert" --non_volatile_key

//For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "pi;signed_cancel_intel7.ccert"
quartus_pgm -c 1 -m jtag -o "pi;signed_cancel_owner2.ccert"

You may additionally send the compact certificate to the SDM using the FPGA or HPS mailbox interface.

### 4.4. Canceling Root Keys

Intel Agilex devices support the capability to cancel root key hashes when another uncanceled root key hash is present. You cancel a root key hash by first configuring the device with a design whose signature chain is rooted in a different root key hash, then program a signed root key hash cancellation compact certificate. You must sign the root key hash cancellation compact certificate with a signature chain rooted in the root key to be canceled.

Run the following command to generate an unsigned root key hash cancellation compact certificate.

```bash
quartus_pfg --ccert -o --ccert_type=CANCEL_KEY_HASH \
unsigned_root_cancel.ccert
```

Run one of the following commands to sign the unsigned root key hash cancellation compact certificate.

```bash
quartus_sign --family=agilex --operation=SIGN \
--cancel=svnA:0 unsigned_root_cancel.ccert signed_root_cancel.ccert

quartus_sign --family=agilex --operation=sign --module=softHSM \
--module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=design0_sign \
--qky=design0_sign_chain.qky --cancel=svnA:0 \
unsigned_root_cancel.ccert signed_root_cancel.ccert
```

You may program a root key hash cancellation compact certificate via JTAG, FPGA, or HPS mailboxes.
4.5. Programming Counter Fuses

You increment Security Version Number (SVN) and Pseudo Time Stamp (PTS) counter fuses using signed compact certificates.

**Note:**
The SDM keeps track of the minimum counter value seen during a given configuration and does not accept counter increment certificates when the counter value is greater than the minimum value. You must update all objects assigned to a counter and reconfigure the device prior to programming a counter increment compact certificate.

Run one of the following commands that corresponds to the counter increment certificate you want to generate.

```bash
quartus_pfg --ccert -o ccert_type=PTS_COUNTER -o counter=<-1:495> unsigned_pts.ccert
quartus_pfg --ccert -o ccert_type=SVN_COUNTER_0 -o counter=<-1:63> unsigned_svnA.ccert
quartus_pfg --ccert -o ccert_type=SVN_COUNTER_1 -o counter=<-1:63> unsigned_svnB.ccert
quartus_pfg --ccert -o ccert_type=SVN_COUNTER_2 -o counter=<-1:63> unsigned_svnC.ccert
quartus_pfg --ccert -o ccert_type=SVN_COUNTER_3 -o counter=<-1:63> unsigned_svnD.ccert
```

A counter value of –1 creates a counter increment authorization certificate. Programming a counter increment authorization compact certificate enables you to program further unsigned counter increment certificates to update the respective counter. You use the `quartus_sign` tool to sign the counter compact certificates in a similar fashion to key cancellation ID compact certificates.

You may program a root key hash cancellation compact certificate via JTAG, FPGA, or HPS mailboxes.

4.6. Secure Data Object Service Root Key Provisioning

You use the Intel Quartus Prime Programmer to provision the Secure Data Object Service (SDOS) root key. The Programmer automatically loads the provision firmware helper image to provision the SDOS root key.

```bash
quartus_pgm -c 1 -m jtag --service_root_key --non_volatile_key
```

4.7. Security Setting Fuse Provisioning

The `.fuse` file contains a list of fuse name-value pairs. The value specifies whether a fuse has been blown or the contents of the fuse field.

The following example shows the format of the `.fuse` file.

```plaintext
# Co-signed firmware = "Not blown"
# Device Permit Kill = "Not blown"
# Device not secure = "Not blown"
# Disable HPS debug = "Not blown"
# Disable Intrinsic ID PUF enrollment = "Not blown"
```
You modify the `.fuse` file to set the desired security setting fuses. A line that begins with `#` is treated as a comment line. To program a security setting fuse, you must remove the leading `#` and set the value to `Blown`. For example, to enable the Co-signed Firmware security setting fuse, you modify the first line of the fuse file to the following:

```
Co-signed firmware = "Blown"
```

You may also allocate and program the Owner Fuses according to your requirements.

The following fields are not writable through the `.fuse` file method; however, they are included during the examine operation output for verification:

- Device not secure
- Device permit kill
- Disable owner root public key hash 0
- Disable owner root public key hash 1
- Disable owner root public key hash 2
- Intel key cancellation
- Owner encryption key program start
• Owner encryption key program done
• Owner key cancellation
• Owner public key hash
• Owner public key size
• Owner root public key hash 0
• Owner root public key hash 1
• Owner root public key hash 2
• PTS counter
• PTS counter base
• QSPI start up delay
• RMA counter
• SDMIO0 is I2C
• SVN counter 0
• SVN counter 1
• SVN counter 2
• SVN counter 3

You use the Intel Quartus Prime Programmer to program the .fuse file back to the device. If you add the i option, the Programmer automatically loads the provision firmware to program the security setting fuses.

//For physical (non-volatile) eFuses
quartus_pgm -c 1 -m jtag -o "pi;programming_file.fuse" --non_volatile_key
//For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "pi;programming_file.fuse"

4.8. AES Root Key Provisioning

You must use a signed AES root key compact certificate to program an AES root key to an Intel Agilex device.

4.8.1. AES Root Key Compact Certificate

You use the quartus_pfg command line tool to convert your AES root key .qek file into the compact certificate .ccert format. You specify the key storage location while creating the compact certificate. You may use the quartus_pfg tool to create an unsigned certificate for later signing. You must use a signature chain with the AES root key certificate signing permission, permission bit 6, enabled in order to successfully sign an AES root key compact certificate.
Run one of the following command examples to create an additional key pair used to sign AES key compact certificate.

```
quartus_sign --family=agilex --operation=make_private_pem \
  --curve=secp384r1 aesccert1_private.pem
```

```
quartus_sign --family=agilex --operation=make_public_pem \
  aesccert1_private.pem aesccert1_public.pem
```

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
  --token_label=agilex-token --login --pin agilex-token-pin \
  --keypairgen --mechanism ECDSA-KEY-PAIR-GEN \
  --key-type EC:secp384r1 --usage-sign --label aesccert1 --id 2
```

Run one of the following commands to create a signature chain with the correct permission bit set.

```
quartus_sign --family=agilex --operation=append_key \
  --previous_pem=root_private.pem --previous_qky=root.qky \
  --permission=0x40 --cancel=1 \
  aesccert1_public.pem aesccert1_sign_chain.qky
```

```
quartus_sign --family=agilex --operation=append_key \
  --module=softHSM --module_args="--token_label=agilex-token \
  --user_pin=agilex-token-pin --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
  --previous_pem=root --previous_qky=root.qky \
  --permission=0x40 --cancel=1 aesccert1 aesccert1_sign_chain.qky
```

Run one of the following commands to create an unsigned AES compact certificate depending on the desired AES root key storage location.

```
//Create eFuse AES root key unsigned certificate
quartus_pfg --ccert -o ccert_type=EFUSE_WRAPPED_AES_KEY \
  -o qek_file=aes.qek unsigned_efuse1.ccert
```

```
//Create BBRAM AES root key unsigned certificate
quartus_pfg --ccert -o ccert_type=BBRAM_WRAPPED_AES_KEY \
  -o qek_file=aes.qek unsigned_bbram1.ccert
```

You use the `quartus_sign` command or reference implementation to sign the compact certificate.

```
quartus_sign --family=agilex --operation=sign \
  --pem=aesccert1_private.pem --qky=aesccert1_sign_chain.qky \
  unsigned_<location>1.ccert signed_<location>1.ccert
```

```
quartus_sign --family=agilex --operation=sign --module=softHSM \
  --module_args="--token_label=agilex-token --user_pin=agilex-token-pin \
  --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=aesccert1 \
  --qky=aesccert1_sign_chain.qky \
  unsigned_<location>1.ccert signed_<location>1.ccert
```

You use the Intel Quartus Prime Programmer to program the AES root key compact certificate to the Intel Agilex device via JTAG. The Quartus Programmer defaults to programming virtual eFuses when using the EFUSE_WRAPPED_AES_KEY compact certificate type. You add the `--non_volatile_key` option to specify programming physical fuses.

```
//For physical (non-volatile) eFuse AES root key
quartus_pgm -c 1 -m jtag -o "pi;signed_efuse1.ccert" --non_volatile_key
```

```
//For virtual (volatile) eFuse AES root key
quartus_pgm -c 1 -m jtag -o "pi;signed_efuse1.ccert"
```
// For BBRAM AES root key
Quartus_pgm -c 1 -m jtag -o "pi;signed_bbram1.ccert"

The SDM provision firmware and main firmware support AES root key certificate programming. You may also use the SDM mailbox interface from the FPGA fabric or HPS to program an AES root key certificate.

4.9. Converting Owner Root Key, AES Root Key Certificates, and Fuse files to Jam STAPL File Formats

You may use the `quartus_pfg` command-line command to convert .qky, AES root key .ccert, and .fuse files to Jam STAPL Format File (.jam) and Jam Byte Code Format File (.jbc). You can use these files to program Intel FPGAs using the Jam STAPL Player and the Jam STAPL Byte-Code Player, respectively.

A single .jam or .jbc contains several functions including a firmware helper image configuration and program, blank check, and verification of key and fuse programming.

Caution: When you convert the AES root key .ccert file to .jam format, the .jam file contains the AES key in plaintext but obfuscated form. Consequently, you must protect the .jam file when storing the AES key. You can do this by provisioning the AES key in a secure environment.

Here are examples of `quartus_pfg` conversion commands:

```
quartus_pfg -c -o helper_device=AFGB014R24A "root0.qky;root1.qky;root2.qky" RootKey.jam
quartus_pfg -c -o helper_device=AFGB014R24A "root0.qky;root1.qky;root2.qky" RootKey.jbc
quartus_pfg -c -o helper_device=AFGB014R24A aes.ccert aes_ccert.jam
quartus_pfg -c -o helper_device=AFGB014R24A aes.ccert aes_ccert.jbc
quartus_pfg -c -o helper_device=AFGB014R24A settings.fuse settings_fuse.jam
quartus_pfg -c -o helper_device=AFGB014R24A settings.fuse settings_fuse.jbc
```

For more information about the using the Jam STAPL Player for device programming refer to [AN 425: Using the Command-Line Jam STAPL Solution for Device Programming](#).
Run the following commands to program the owner root public key and AES encryption key:

```plaintext
// To load the helper bitstream into the FPGA.
// The helper bitstream include SDM firmware
quartus_jli -c 1 -a CONFIGURE RootKey.jam

// To program the owner root public key into virtual eFuses
quartus_jli -c 1 -a PUBKEY_PROGRAM RootKey.jam

// To program the owner root public key into physical eFuses
quartus_jli -c 1 -a PUBKEY_PROGRAM -e DO_UNI_ACT_DO_EFUSES_FLAG RootKey.jam

// To program the PR owner root public key into physical efuses
quartus_jli -c 1 -a PUBKEY_PROGRAM -e DO_UNI_ACT_DO_PR_PUBKEY_FLAG pr_rootkey.jam

// To program the AES encryption key CCERT into BBRAM
quartus_jli -c 1 -a CCERT_PROGRAM EncKeyBBRAM.jam

// To program the AES encryption key CCERT into physical eFuses
quartus_jli -c 1 -a CCERT_PROGRAM -e DO_UNI_ACT_DO_EFUSES_FLAG EncKeyEFuse.jam
```
5. Advanced Features

5.1. Secure Debug Authorization

To enable Secure Debug Authorization, the debug owner needs to generate an authentication key pair and use the Intel Quartus Prime Pro Programmer to generate a device information file for the device that runs the debug image:

```
quartus_pgm -c 1 -m jtag -o "ei;device_info.txt;AGFB014R24A" --dev_info
```

The device owner uses the `quartus_sign` tool or the reference implementation to append a conditional public key entry to a signature chain intended for debug operations using the public key from the debug owner, the necessary authorizations, the device information text file, and applicable further restrictions.

```
quartus_sign --family=agilex --operation=append_key \
--previous_pem=debug_chain_private.pem --previous_qky=debug_chain.qky \
--permission=0x6 --cancel=1 \
--dev_info=device_info.txt --restriction="1,2,17,18" \
debug_authorization_public_key.pem secure_debug_auth_chain.qky
```

The device owner sends the full signature chain back to the debug owner, who uses the signature chain and their private key to sign the debug image.

```
quartus_sign --family=agilex --operation=sign \
--qky=secure_debug_auth_chain.qky --pem=debug_authorization_private_key.pem \
unsigned_debug_design.rbf authorized_debug_design.rbf
```

The debug owner can then program the securely authorized debug design.

```
quartus_pgm -c 1 -m jtag -o "p;authorized_debug_design.rbf"
```

The device owner may revoke the secure debug authorization by canceling the explicit key cancellation ID assigned in the secure debug authorization signature chain.

5.2. HPS Debug Certificates

To enable only authorized access to the HPS debug access port (DAP) via JTAG interface, you click the Intel Quartus Prime Assignments menu and select Device ➤ Device and Pin Options ➤ Configuration tab, then enable the HPS debug access port (DAP) by selecting either HPS Pins or SDM Pins from the drop down menu, and ensuring the Allow HPS debug without certificates checkbox is not selected.
Specify Either HPS or SDM Pins for the HPS DAP

You then compile and load the design with these settings.

You create a signature chain with the appropriate permissions to sign an HPS debug certificate.

```
quartus_sign --family=agilex --operation=append_key \
   --previous_pem=root_private.pem --previous_qky=root.qky \
   --permission=0x8 --cancel=1 \
   hps_debug_cert_public_key.pem hps_debug_cert_sign_chain.qky
```

You use the Intel Quartus Prime Programmer to request an unsigned HPS debug certificate from the device where the debug design is loaded.

```
quartus_pgm -c 1 -m jtag -o "e;unsigned_hps_debug.cert;AGFB014R24A"
```

You sign the unsigned HPS debug certificate using the `quartus_sign` tool or reference implementation and the HPS debug signature chain.

```
quartus_sign --family=agilex --operation=sign \
   --qky=hps_debug_cert_sign_chain.qky \
   --pem=hps_debug_cert_private_key.pem \
   unsigned_hps_debug.cert signed_hps_debug.cert
```

You use the Intel Quartus Prime Programmer to send the signed HPS debug certificate back to the device to enable access to the HPS DAP.

```
quartus_pgm -c 1 -m jtag -o "p;signed_hps_debug.cert"
```

The HPS debug certificate is only valid from the time it was generated until the next power cycle of the device or until a different type or version of SDM firmware is loaded. You must generate, sign, and program the signed HPS debug certificate, and perform all debug operations, prior to power cycling the device. You may invalidate the signed HPS debug certificate by power cycling the device.
5.3. Platform Attestation

You can generate a reference integrity manifest (.rim) file using the programming file generator tool:

```
quartus_pfg -c signed_encrypted_top.rbf top_rim.rim
```

Follow these steps to ensure the platform attestation in your design:

1. Use the Intel Quartus Prime Pro Programmer to configure your device with the design you created a reference integrity manifest for.
2. Use a platform attestation verifier to enroll the device by issuing commands to the SDM via the SDM mailbox to create the device ID certificate and firmware certificate on reload.
3. Use the Intel Quartus Prime Pro Programmer to reconfigure your device with the design.
4. Use the platform attestation verifier to issue commands to the SDM to get the attestation device ID, firmware, and alias certificates.
5. Use the attestation verifier to issue the SDM mailbox command to get the attestation evidence and the verifier checks the returned evidence.

You may implement your own verifier service using the SDM mailbox commands, or use the Intel platform attestation verifier service. For more information about Intel platform attestation verifier service software, availability, and documentation, contact Intel Support.
5.4. Using Design Security Features with Remote System Update

Remote System Update (RSU) is an Intel Agilex FPGAs feature that assists in updating configuration files in a robust way. RSU is compatible with design security features such as authentication, firmware, co-signing, and bitstream encryption as RSU does not depend on the design contents of configuration bitstreams.

In order to use design security features with RSU images, you follow the instructions in Generating Remote System Update Image Files Using the Programming File Generator of the Intel Agilex Configuration User Guide to generate RSU images with .sof file inputs. For every .sof file specified on the Input Files tab, you click the Properties... button and specify the appropriate settings and keys for the signing and encryption tools. The programming file generator tool automatically signs and encrypts factory and application images while creating the RSU programming files.

You may build RSU images with .rbf format files as inputs. You must encrypt and sign .rbf format files prior to selecting them as input files for RSU images; however, the RSU boot info .rbf file must not be encrypted, only signed. The Programming File Generator does not support modifying properties of .rbf format files.

The following examples demonstrate the necessary modifications to the commands in Generating Remote System Update Image Files Using the Programming File Generator of the Intel Agilex Configuration User Guide.

Generating the Initial RSU Image Using .rbf Files: Command Modification

From Generating the Initial RSU Image Using .rbf Files, you modify the commands in Step 1. to enable the design security features as desired using instructions from earlier sections of this document.

In step 2, if you have enabled firmware co-signing, you must use an additional option in the creation of the boot .rbf from the factory image file:

```
quartus_pfg -c factory.sof boot.rbf -o rsu_boot=ON \\
-o fw_source=signed_agilex.zip
```

Generating an Application Image: Command Modification

To generate an application image with design security features, you modify the command in Generating an Application Image to use a .rbf with design security features enabled, including co-signed firmware if required, instead of the original application .sof file.

```
quartus_pfg -c coSigned_fw_signed_encrypted_application.rbf \\
secured.rsu_application.rpd -o mode=ASX4 \\
-o start_address=<start_address> -o bitswap=ON
```
Generating a Factory Update Image: Command Modification

To generate an RSU factory update image, you modify the command from Generating a Factory Update Image to use a .rbf file with design security features enabled and add the option to indicate the co-signed firmware usage.

```
quartus_pfg -c cosigned_fw_signed_encrypted_factory.rbf \
secured_rsu_factory_update.rpd \
-o mode=ASX4 -o start_address=<start_address> \
-o bitswap=ON -o rsu_upgrade=ON \
-o fw_source=signed_agilex.zip
```

Related Information
Intel Agilex Configuration User Guide

5.5. SDM Cryptographic Services

The Secure Device Manager (SDM) on Intel Agilex devices provides cryptographic services that FPGA fabric logic or the HPS may request via the mailbox commands.

To access the SDM mailbox interface for SDM cryptographic services, you instantiate the Mailbox Client Intel FPGA IP in your design and set the HAS_OFFLOAD parameter to 1 and connect the exposed AXI initiator interface to a memory in your design.

Figure 8. Enabling SDM Cryptographic Services in the Mailbox Client Intel FPGA IP

For more information about integrating the Mailbox Client IP in your design, refer to Mailbox Client Intel FPGA IP User Guide.

Related Information
Mailbox Client Intel FPGA IP User Guide

5.5.1. Vendor Authorized Boot

Intel provides a reference implementation for HPS software that utilizes the vendor authorized boot feature to authenticate HPS boot software from the first stage boot loader through to the Linux kernel.
5. Advanced Features

5.5.2. Secure Data Object Service

You may use the Secure Data Object Service (SDOS) after following steps in Secure Data Object Service Root Key Provisioning on page 28. You send the commands through the SDM mailbox to perform SDOS object encryption and decryption.

5.5.3. SDM Cryptographic Primitive Services

You send the commands through the SDM mailbox to initiate SDM cryptographic primitive service operations.

If the table does not list a software version, the user guide for the previous software version applies.

<table>
<thead>
<tr>
<th>Intel Quartus Prime Version</th>
<th>User Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.2</td>
<td>Intel Agilex Device Security User Guide</td>
</tr>
</tbody>
</table>
# 7. Revision History for the Intel Agilex Device Security User Guide

<table>
<thead>
<tr>
<th>Document Version</th>
<th>Intel Quartus Prime Version</th>
<th>Changes</th>
</tr>
</thead>
</table>
| 2021.11.09       | 21.3                        | Made the following changes:  
|                  |                             | • Added note in the *Planned Security Features* section.  
|                  |                             | • Updated *Planned Security Features* sub-sections:  
|                  |                             | — Removed *Crypto Services* and *Platform Attestation* sections. The features are available for Intel Agilex devices.  
|                  |                             | • Added support for the platform attestation feature. Added new topic:  
|                  |                             | — *Platform Attestation*  
|                  |                             | • Added support for the crypto services feature. Added new topics:  
|                  |                             | — *SDM Cryptographic Services*  
|                  |                             | — *Vendor Authorized Boot*  
|                  |                             | — *Secure Data Object Service*  
|                  |                             | — *SDM Cryptographic Primitive Services*  
|                  |                             | • Added step to extract firmware in *Co-Signing SDM Firmware*. |
| 2021.09.10       | 21.2                        | Initial release |