



# Intel® Stratix® 10 Device Security User Guide

Updated for Intel® Quartus® Prime Design Suite: **21.3**



**Online Version**



**Send Feedback**

**UG-S10SECURITY**

ID: **683642**

Version: **2021.11.09**

## Contents

---

<b>1. Intel Stratix 10 Device Security Overview.....</b>	<b>4</b>
1.1. Commitment to Product Security.....	4
<b>2. Authentication and Authorization.....</b>	<b>5</b>
2.1. Creating a Signature Chain.....	5
2.1.1. Creating Authentication Key Pairs on the Local Filesystem.....	6
2.1.2. Creating Authentication Key Pairs in SoftHSM.....	7
2.1.3. Creating the Signature Chain Root Entry.....	8
2.1.4. Creating a Signature Chain Public Key Entry.....	8
2.2. Signing a Configuration Bitstream.....	9
2.2.1. Quartus Key File Assignment.....	9
2.2.2. Co-Signing SDM Firmware.....	10
2.2.3. Configuring Bitstream Signing Using the quartus_sign Command.....	11
2.2.4. Verifying Configuration Bitstream Signature Chains.....	12
<b>3. AES Bitstream Encryption.....</b>	<b>15</b>
3.1. Creating the AES Root Key.....	15
3.2. Quartus Encryption Settings.....	15
3.3. Encrypting a Configuration Bitstream.....	16
3.3.1. Configuration Bitstream Encryption Using the Programming File Generator Graphical Interface.....	16
3.3.2. Configuration Bitstream Encryption Using the Programming File Generator Command Line Interface.....	18
3.3.3. Partially Encrypted Configuration Bitstream Generation Using the Command Line Interface.....	18
3.3.4. Partial Reconfiguration Bitstream Encryption.....	18
<b>4. Device Provisioning.....</b>	<b>21</b>
4.1. Using SDM Provision Firmware.....	21
4.2. Authentication Root Key Provisioning.....	21
4.3. Programming Key Cancellation ID Fuses.....	23
4.4. Security Setting Fuse Provisioning.....	24
4.5. AES Root Key Provisioning.....	26
4.5.1. AES Root Key Compact Certificate.....	26
4.5.2. Intrinsic ID® PUF AES Root Key Provisioning.....	27
4.5.3. Black Key Provisioning.....	35
4.6. Converting Owner Root Key, AES Root Key Certificates, and Fuse files to Jam STAPL File Formats.....	37
<b>5. Advanced Features.....</b>	<b>39</b>
5.1. Secure Debug Authorization.....	39
5.2. HPS Debug Certificates.....	39
5.3. Platform Attestation.....	41
5.4. Physical Anti-Tamper.....	41
5.4.1. Anti-Tamper Responses.....	42
5.4.2. Anti-Tamper Detection.....	44
5.4.3. Anti-Tamper Lite Intel FPGA IP.....	46
5.5. Using Design Security Features with Remote System Update.....	48

<b>6. Intel Stratix 10 Device Security User Guide Archives.....</b>	<b>50</b>
<b>7. Document Revision History for Intel Stratix 10 Device Security User Guide.....</b>	<b>51</b>

## 1. Intel Stratix 10 Device Security Overview

---

Intel® designs the Intel Stratix 10 devices with dedicated, highly configurable security hardware and firmware. The *Security Methodology User Guide* document available on the Intel Resource & Design Center contains detailed descriptions of the security features and technologies in Intel Programmable Solutions products to help you select the security features necessary to meet your security objectives. Contact Intel Support with reference number 14014613136 for access to the *Security Methodology User Guide*.

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 Stratix 10 devices.

This document organizes content as follows:

- **Authentication and Authorization:** Provides instructions to create authentication keys and signature chains, apply permissions and cancellation IDs, sign objects, and program authentication features on Intel Stratix 10 devices.
- **AES Bitstream Encryption:** Provides instructions to create an AES root key, encrypt configuration bitstreams, and provision the AES root key to Intel Stratix 10 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 Stratix 10 devices.
- **Advanced Features:** Provides instructions to enable advanced security features, including secure debug authorization, Hard Processor System (HPS) debug, platform attestation, physical anti-tamper, 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](#)

## 2. Authentication and Authorization

---

To enable the authentication features of an Intel Stratix 10 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 21 to program your root key into Intel Stratix 10 devices.

### Related Information

[Device Provisioning](#) on page 21

### 2.1. Creating a Signature Chain

You may use the `quartus_sign` tool or the `stratix10_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=Stratix10` option; all other options are equivalent. For example, the `quartus_sign` command found later in this section

```
quartus_sign --family=stratix10 --operation=make_root root_public.pem root.qky  
can be converted into the equivalent call to the reference implementation as follows  
pgm_py stratix10_sign.py --operation=make_root root_public.pem root.qky
```

Intel Quartus Prime Pro Edition software includes the `quartus_sign`, `pgm_py`, and `stratix10_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.

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

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 `stratix10_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.

**Option****Description***With passphrase*

```
quartus_sign --family=stratix10 --operation=make_private_pem \
--curve=secp384r1 root_private.pem
Enter the passphrase when prompted to do so.
```

*Without passphrase*

```
quartus_sign --family=stratix10 --operation=make_private_pem \
--curve=secp384r1 --no_passphrase root_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=stratix10 --operation=make_public_pem \
root_private.pem root_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=stratix10 --operation=make_private_pem \
--curve=secp384r1 design0_sign_private.pem
```

```
quartus_sign --family=stratix10 --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 s10-token --pin s10-token-pin \
--so-pin s10-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 s10-token --login --pin s10-token-pin --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp384r1 --usage-sign \
--label root --id 0
```

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label s10-token --login --pin s10-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.

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=stratix10 \
--operation=make_root root_public.pem root.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=stratix10 --operation=make_root \
--module=softHSM --module_args="--token_label=s10-token \
--user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" root root.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=stratix10 --operation=append_key \
--previous_pem=root_private.pem --previous_qky=root.qky \
--permission=6 --cancel=0 design0_sign_public.pem \
design0_sign_chain.qky
```

```
quartus_sign --family=stratix10 --operation=append_key --module=softHSM \
--module_args="--token_label=s10-token --user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root --previous_qky=root.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=stratix10 --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
```

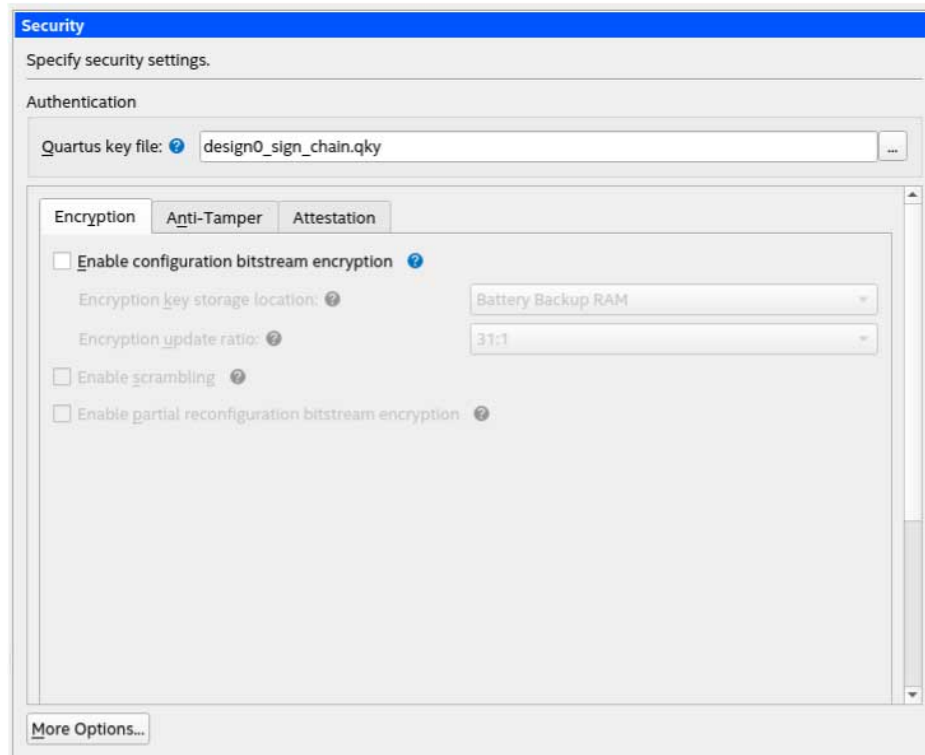
```
quartus_sign --family=stratix10 --operation=append_key --module=softHSM \  
--module_args="--token_label=s10-token --user_pin=s10-token-pin \  
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \  
--previous_pem=design0_sign_private.pem \  
--previous_qky=design0_sign_chain.qky \  
--permission=6 \  
--cancel=1 design1_sign_public.pem design1_sign_chain.qky
```

## 2.2. Signing a Configuration Bitstream

### 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.

**Figure 1. Enable Configuration Bitstream Setting**



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.

```
quartus_sign --family=stratix10 --operation=make_private_pem \
--curve=secp384r1 firmware1_private.pem
```

```
quartus_sign --family=stratix10 --operation=make_public_pem \
firmware1_private.pem firmware1_public.pem
```

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

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label s10-token --login --pin s10-token-pin \
--keypairgen --mechanism ECDSA-KEY-PAIR-GEN \
--key-type EC:secp384r1 --usage-sign --label firmware1 --id 1
```

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

```
quartus_sign --family=stratix10 --operation=append_key \
--previous_pem=root_private.pem --previous_qky=root.qky \
--permission=0x1 --cancel=1 \
firmware1_public.pem firmware1_sign_chain.qky
```

```
quartus_sign --family=stratix10 --operation=append_key \
--module=softHSM --module_args="--token_label=s10-token \
--user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root --previous_qky=root.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/stratix10.zip) to the current working directory.

```
quartus_sign --family=stratix10 --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.

```
quartus_sign --family=stratix10 --operation=sign \
--qky=firmware1_sign_chain.qky \
--pem=firmware1_private.pem stratix10.zip signed_stratix10.zip
```

```
quartus_sign --family=stratix10 --operation=sign --module=softHSM \
--module_args="--token_label=s10-token --user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" --pem=firmware1 \
--qky=firmware1_sign_chain.qky stratix10.zip signed_stratix10.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:

```
quartus_pfg -c -o fw_source=signed_Stratix10.zip design.sof \
-o sign_later=ON 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=stratix10 --operation=sign \
--qky=design0_sign_chain.qky --pem=design0_sign_private.pem \
unsigned_bitstream.rbf signed_bitstream.rbf
```

```
quartus_sign --family=stratix10 --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 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. 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=stratix10 --operation=fuse_info public_root.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:

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

Section
Type: CMF
Signature Descriptor ...
Signature chain #0 (entries: 3, offset: 96)
Entry #0
Fuse: A1B9545C CAC4152D 9511A9AB 321778ED 1180A280 6DC58F2C
5607433E 02A872E3 F52B2AE5 F7B8BDE0 53FA000D 8FC7AC04
Generate key ...
Curve : secp384r1
X: FC28C88662DF1437DD98E61336467DC9CDA788F22F949D8F488DA755A9F8CC11AEC10006E2
6490B3EAB8148E6C8AA8A1
Y: 95D1EA0FF4C7374B350FDF39CFAE3AD8D0AEA9451EA66B5B1DFD4084DA68BC4DAD3AF5CF37
8D7C6FB62A10BA7C512276
Entry #1
Generate key ...
Curve : secp384r1
```

```

X: B11534AA67A30EF884B89819281522F1D0326BBAFF108BC483946717A14F9630C682ECDAE5
40FECBADF3E66BC92A110A
Y: 0ED5F19E6A38D97148CE6F53B679227311198105BD9E1912AD41C075711F6185E1B095DE7F
E2F4855851E78F9BF3D2C6
Entry #2
Keychain permission: SIGN_CODE
Keychain can be cancelled by ID: 5
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: IO
Signature Descriptor ...
Signature chain #0 (entries: 5, offset: 96)
Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F1E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A80EFDA1552
Y: 9A8A74E440002AE72FF67716FE889C49DD5D0FD4FBC7195324DE267BFF06FF49

Entry #1
Generate key ...
Curve : prime256v1
X: 7EF9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A418847AAA82505
Y: 29EE71EAF4CDBB99414C2673EA7AD44B4EE4442E803D350590DA0D95A0F2EF5

Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF4B91136EAC43041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFF4B071BC8D291FD9477EE035AD9C46B696DD20F5702809

Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FBB3D3F0E5961E7FFF7D8E94AFD1836752169A9E66B79BB5861BBDA79E53F
Y: 361FE17E8C73DE0FB4277480FAED32363A3C134DD27D6961E6F046222F06D600

Entry #4
Keychain permission: SIGN_CORE, SIGN_HPS
Keychain can be cancelled by ID: 0, 0, 0
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: HPS
Signature Descriptor ...
Signature chain #0 (entries: 5, offset: 96)

Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F1E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A80EFDA1552
Y: 9A8A74E440002AE72FF67716FE889C49DD5D0FD4FBC7195324DE267BFF06FF49

Entry #1
Generate key ...
Curve : prime256v1
X: 7EF9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A418847AAA82505
Y: 29EE71EAF4CDBB99414C2673EA7AD44B4EE4442E803D350590DA0D95A0F2EF5

Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF4B91136EAC43041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFF4B071BC8D291FD9477EE035AD9C46B696DD20F5702809

```

```

Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FBB3D3F0E5961E7FFF7D8E94AFD1836752169A9E66B79BB5861BBDA79E53F
Y: 361FE17E8C73DE0FB4277480FAED32363A3C134DD27D6961E6F046222F06D600

Entry #4
Keychain permission: SIGN_CORE, SIGN_HPS
Keychain can be cancelled by ID: 0, 0, 0
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

Section
Type: CORE
Signature Descriptor ...
Signature chain #0 (entries: 5, offset: 96)

Entry #0
Fuse: 46D2D1CD 666F6FA3 8CA6DF11 F09F1E84 41162254 D5E811F0 0B72B678 52D29F2F
Generate key ...
Curve : prime256v1
X: DD4E3FB89EC29E0F2C9435A8D74E0780F2282367EABF4F84FD207A80EFDA1552
Y: 9A8A74E440002AE72FF67716FE889C49DD5D0FD4FBC7195324DE267BFF06FF49

Entry #1
Generate key ...
Curve : prime256v1
X: 7EF9D2C6D246339E6D58B937D4127F83FF590B64663FEC316A418847AAA82505
Y: 29EE71EAF4CDBB99414C2673EA7AD44B4EE4442E803D350590DA0D95A0F2EF5

Entry #2
Generate key ...
Curve : prime256v1
X: 3A9083FF4B91136EAC43041916C2E1FC887397ABCEA017DE42AF143DBEA17ED8
Y: 4DDDD1670C3F846EFFC4B071BC8D291FD9477EE035AD9C46B696DD20F5702809

Entry #3
Generate key ...
Curve : prime256v1
X: 8A1FBB3D3F0E5961E7FFF7D8E94AFD1836752169A9E66B79BB5861BBDA79E53F
Y: 361FE17E8C73DE0FB4277480FAED32363A3C134DD27D6961E6F046222F06D600

Entry #4
Keychain permission: SIGN_CORE, SIGN_HPS
Keychain can be cancelled by ID: 0, 0, 0
Signature chain #1 (entries: 0, offset: 0)
Signature chain #2 (entries: 0, offset: 0)
Signature chain #3 (entries: 0, offset: 0)

```



## 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.

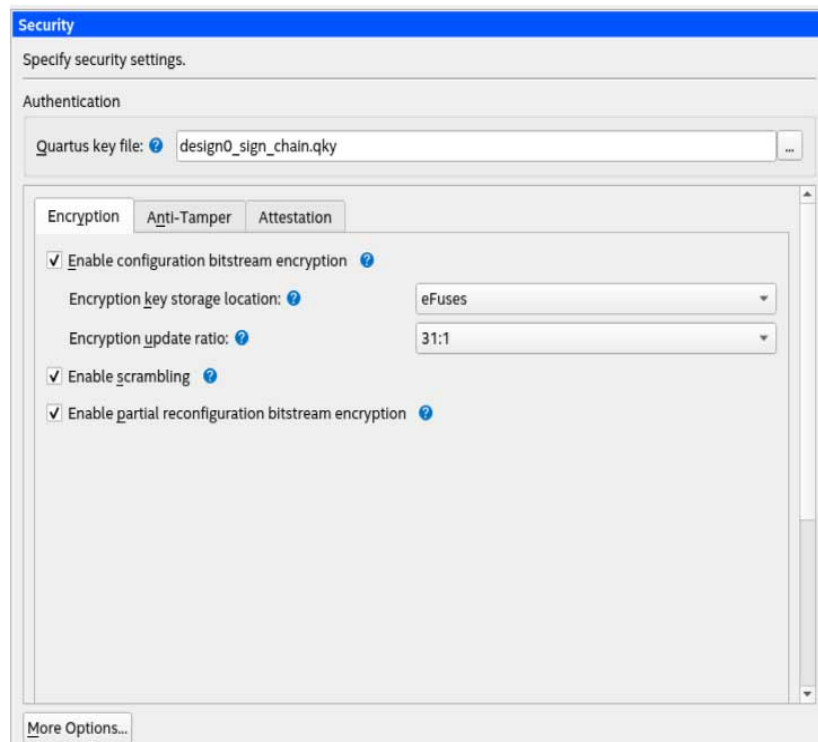
You may optionally specify the base key used to derive the AES root key and key derivation key, the value for the AES root key directly, the number of intermediate keys, and the maximum use per intermediate key. You must specify the device family, output `.qek` file location, and passphrase when prompted. 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.

```
quartus_encrypt --family=stratix10 --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.

**Figure 2. Intel Quartus Prime Encryption Settings**



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.

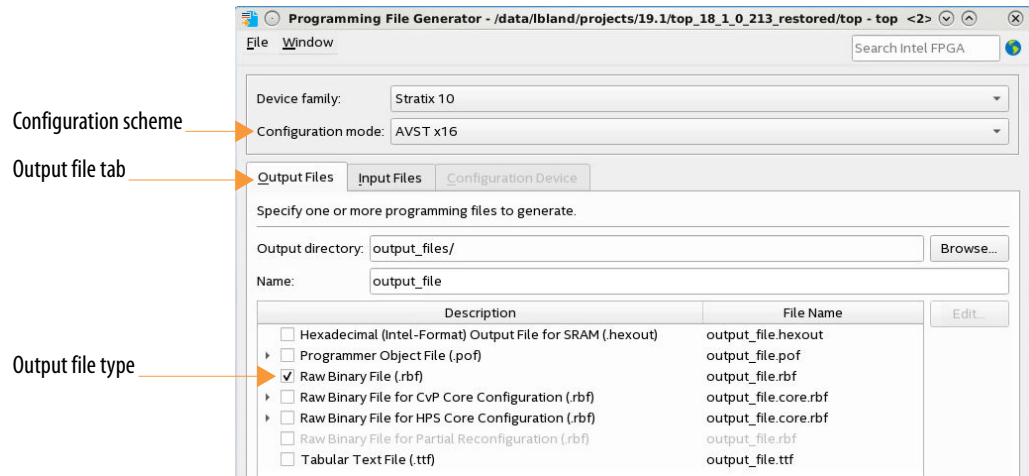
#### 3.3.1. Configuration Bitstream Encryption Using the Programming File Generator Graphical Interface

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.

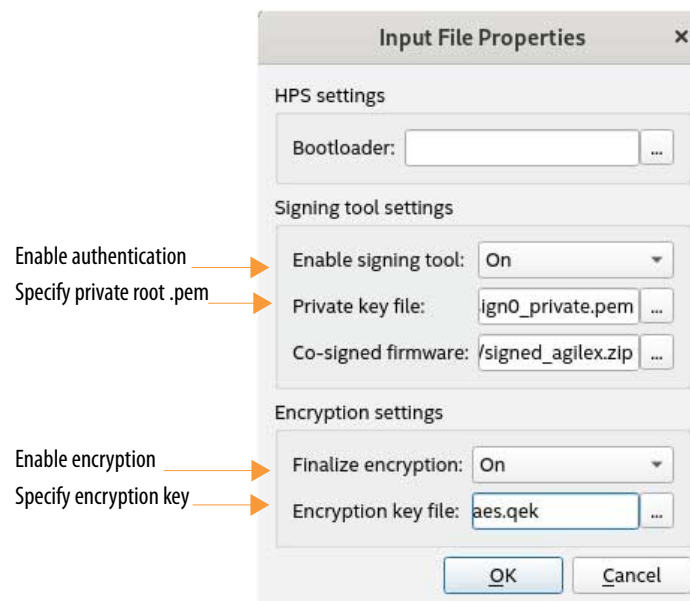


Figure 3. Output File Specification



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.

Figure 4. Input (.sof) File Properties for Authentication and Encryption



5. To generate the signed and encrypted bitstream, on the **Input Files** tab, click **Generate**.

The password dialog box prompts you to input your passphrase for the .qek. The programming file generator generates output\_file.rbf if the passphrase is correct.

### 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.

```
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.

```
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.

```
quartus_encrypt --family=stratix10 \
--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=stratix10 --operation=sign \
--pem=design0_sign_private.pem --qky=design0_sign_chain.qky \
encrypted_top.rbf signed_encrypted_top.rbf
```

```
quartus_sign --family=stratix10 --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 encrypted_top.rbf signed_encrypted_top.rbf
```

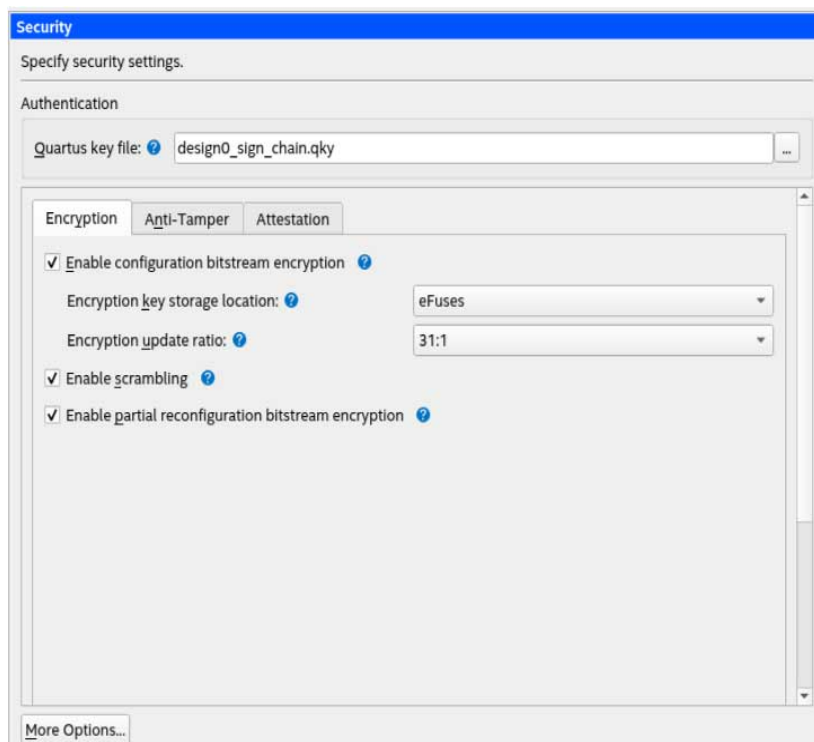
### 3.3.4. Partial Reconfiguration Bitstream Encryption

You can enable bitstream encryption on some Intel Stratix 10 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.

**Figure 5. Partial Reconfiguration Bitstream Encryption Setting**



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_encryption_later=ON \
-o sign_later=ON encryption_enabled_personal.pmsf personal.rbf
```

You use the quartus\_encrypt command line tool to finalize bitstream encryption.

```
quartus_encrypt --family=stratix10 \
--operation=ENCRYPT --key=aes_root.qek personal.rbf encrypted_personal.rbf
```

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

```
quartus_sign --family=stratix10 --operation=SIGN \
--qky=design0_sign_chain.qky \
--pem=design0_sign_private.pem encrypted_personal.rbf \
signed_encrypted_personal.rbf
```

```
quartus_sign --family=stratix10 --operation=SIGN \
--module=softHSM --module_args="--token_label=s10-token \
--user_pin=s10-token-pin --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--qky=design0_sign_chain.qky --pem=design0_sign encrypted_personal.rbf \
signed_encrypted_personal.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=1SX280LH2 -o subtype=PROVISION \
-o fw_source=signed_Stratix10.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.

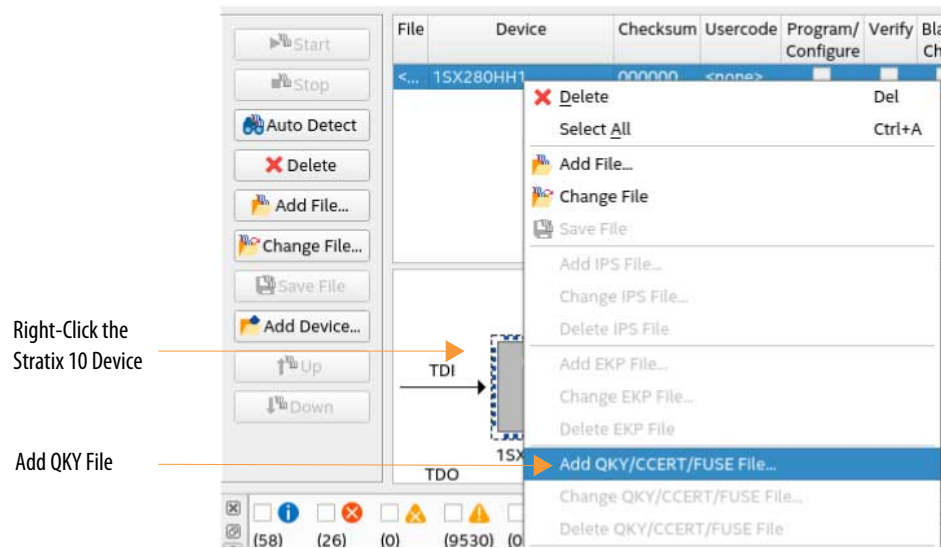
If you plan to use firmware co-signing, you may use a co-signed helper image on an unprovisioned device as the unprovisioned device ignores non-Intel signature chains over SDM firmware.

### 4.2. Authentication Root Key Provisioning

To program the owner root key hash, you must load the provision firmware first following a power-on reset, program the owner root key hash, and immediately perform another power-on reset.

To provision the owner root key hash using the Intel Quartus Prime Programmer graphical interface, select **Programmer** from the **Tools** menu in Intel Quartus Prime software.

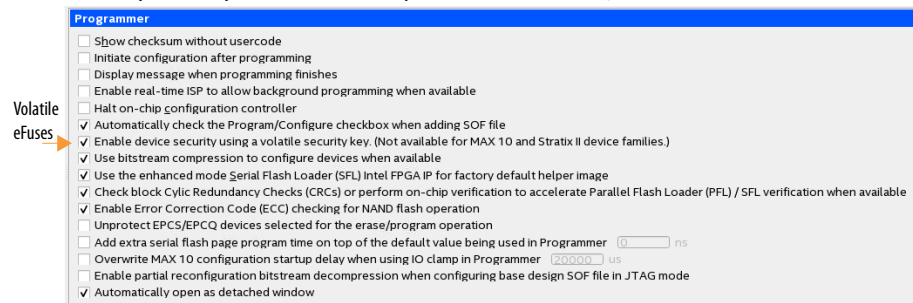
1. Right click the image of the Intel Stratix® 10 device and select **Edit ► Add QKY/CCERT/Fuse file ...**.



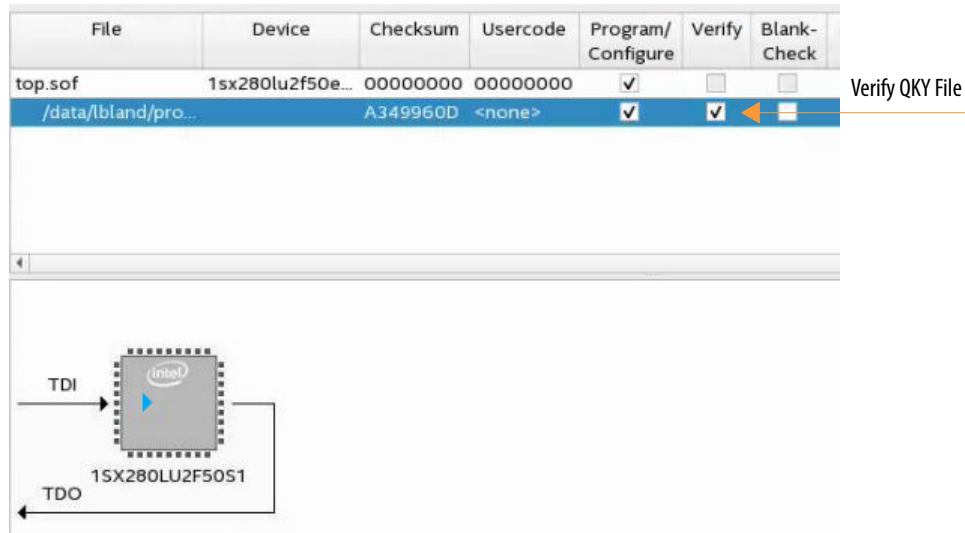
2. Browse to the owner root public key file and click **Open**.
3. You can choose to program the non-volatile eFuses or simulate the actual hardware using virtual eFuses.

**Caution:** Incorrect fuse programming can make your device unusable. Intel recommends that you test all eFuse programming sequences using virtual fusing before you program physical eFuses on your first device.

- To select virtual eFuses, on the Programmer Tools menu, select **Options**. Turn on **Enable device security using a volatile security key** if this option is not already on. By default this option is on. Then, select **OK**.



- To select the actual non-volatile eFuses, on the Programmer Tools menu, select **Options**. Turn off the **Enable device security using a volatile security key** option.
4. Click **Start** to program the owner root public key hash.
  5. Power cycle your device.
  6. To verify that the fuse value and the hash value of the owner root public key match, turn off the **Program/Configure** option and turn on the **Verify** option in the Intel Quartus Prime software.



7. Click **Start** to verify the owner root public key hash programming.

To program the authentication root key hash using the command line interface, run the following command to load the provision firmware helper image.

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

Then, run one of the following command to program the root key .qky file.

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

```
// For virtual (volatile) eFuses
quartus_pgm -c 1 -m jtag -o "p;root.qky"
```

Then, power cycle your device.

### 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.

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.

```
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.

```
quartus_sign --family=stratix10 --operation=SIGN \
--qky=design0_sign_chain.qky --pem=design0_private.pem \
unsigned_cancel_intel7.ccert signed_cancel_intel7.ccert
```

```
quartus_sign --family=stratix10 --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 \
unsigned_cancel_intel7.ccert signed_cancel_intel7.ccert
```

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

```
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.

```
quartus_sign --family=stratix10 --operation=SIGN \
--qky=design0_sign_chain.qky --pem=design0_private.pem \
unsigned_cancel_owner2.ccert signed_cancel_owner2.ccert
```

```
quartus_sign --family=stratix10 --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 \
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. Security Setting Fuse Provisioning

You use the Intel Quartus Prime Programmer to examine device security setting fuses and write them to a text-based .fuse file.

```
quartus_pgm -c 1 -m jtag -o "ei;programming_file.fuse;1SX280LH2"
```

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.

```
# Co-signed firmware = "Not blown"
# Device not secure = "Not blown"
# Disable HPS debug = "Not blown"
# Disable Intrinsic ID PUF enrollment = "Not blown"
# Disable JTAG = "Not blown"
# Disable PUF-wrapped encryption key = "Not blown"
# Disable owner encryption key in BBRAM = "Not blown"
# Disable owner encryption key in eFuses = "Not blown"
# Disable virtual eFuses = "Not blown"
# Force SDM clock to internal oscillator = "Not blown"
# Force encryption key update = "Not blown"
# Intel key cancellation = "1"
# Lock security eFuses = "Not blown"
# Owner encryption key program done = "Not blown"
# Owner encryption key program start = "Not blown"
# Owner fuses =
"0x0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000"
# Owner key cancellation = " "
# Owner public key hash = " "
# Owner public key size = " "
# QSPI start up delay = "10ms"
# RMA Counter = "0"
# SDMIO0 is I2C = "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
- Intel key cancellation
- Owner encryption key program start
- Owner encryption key program done
- Owner key cancellation
- Owner public key hash
- Owner public key size
- QSPI start up delay
- RMA counter
- SDMIO0 is I2C

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"
```

## 4.5. AES Root Key Provisioning

You must use a signed AES root key compact certificate to program an AES root key to an Intel Stratix 10 device.

### 4.5.1. 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=stratix10 --operation=make_private_pem \
--curve=secp384r1 aescert1_private.pem
```

```
quartus_sign --family=stratix10 --operation=make_public_pem \
aescert1_private.pem aescert1_public.pem
```

```
pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token_label s10-token --login --pin s10-token-pin \
--keypairgen -mechanism ECDSA-KEY-PAIR-GEN \
--key-type EC:secp384r1 --usage-sign --label aescert1 --id 2
```

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

```
quartus_sign --family=stratix10 --operation=append_key \
--previous_pem=root_private.pem --previous_qky=root.qky \
--permission=0x40 --cancel=1 \
aescert1_public.pem aescert1_sign_chain.qky
```

```
quartus_sign --family=stratix10 --operation=append_key \
--module=softHSM --module_args="--token_label=s10-token \
--user_pin=s10-token-pin --hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root --previous_qky=root.qky \
--permission=0x40 --cancel=1 aescert1 aescert1_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_efusel.ccert
```

```
//Create BBRAM AES root key unsigned certificate
quartus_pfg --ccert -o ccert_type=BBRAM_WRAPPED_AES_KEY \
-o gek_file=aes.gek unsigned_bbraml.ccert
```

```
//Create IID PUF AES root key unsigned certificate
quartus_pfg --ccert \
-o ccert_type=IID_PUF_WRAPPED_AES_KEY \
-o gek_file=aes.gek \
-o IV=1234567890ABCDEF1234567890ABCDEF unsigned_pufl.ccert
```

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

```
quartus_sign --family=stratix10 --operation=sign \
--pem=aescert1_private.pem --qky=aescert1_sign_chain.qky \
unsigned_<location>1.ccert signed_<location>1.ccert
```

```
quartus_sign --family=stratix10 --operation=sign --module=softHSM \
--module_args="--token_label=s10-token --user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softHSM/libsoftHSM2.so" --pem=aescert1 \
--qky=aescert1_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 Stratix 10 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_efusel.ccert" --non_volatile_key

//For virtual (volatile) eFuse AES root key
quartus_pgm -c 1 -m jtag -o "pi;signed_efusel.ccert"

//For BBRAM AES root key
Quartus_pgm -c 1 -m jtag -o "pi;signed_bbraml.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.

#### 4.5.2. Intrinsic ID® PUF AES Root Key Provisioning

Implementing the Intrinsic\* ID PUF wrapped AES Key includes the following steps:

1. Enrolling the Intrinsic ID PUF via JTAG.
2. Wrapping the AES root key.
3. Programming the helper data and wrapped key into quad SPI flash memory.
4. Querying the Intrinsic ID PUF activation status.

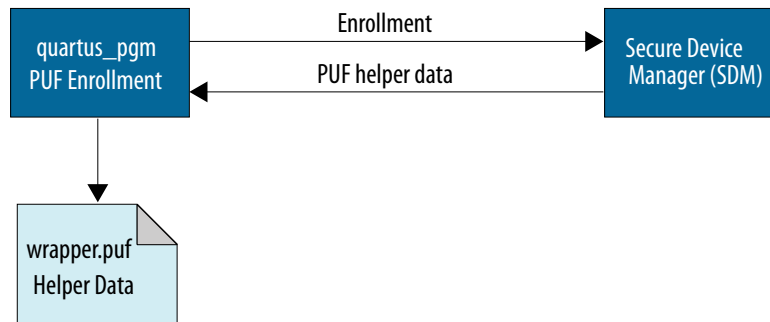
The use of Intrinsic ID technology requires a separate license agreement with Intrinsic ID. Intel Quartus Prime Pro Edition software restricts PUF operations, including enrollment and key wrapping, without the appropriate license.

#### 4.5.2.1. Intrinsic ID PUF Enrollment

To enroll the PUF, you must use the SDM provision firmware. The provision firmware must be the first firmware loaded after a power cycle, and you must issue the PUF enrollment command before any other command. The provision firmware supports other commands after PUF enrollment, including AES root key wrapping and programming quad SPI, however, you must power cycle the device to load a configuration bitstream.

You use the Intel Quartus Prime Programmer to trigger PUF enrollment and generate the PUF helper data .puf file.

**Figure 6. Intrinsic ID PUF Enrollment**



The Programmer automatically loads a provision firmware helper image when you specify both the `i` operation and a `.puf` argument.

```
quartus_pgm -c 1 -m jtag -o "ei;help_data.puf;1SX280LH2"
```

If you are using co-signed firmware, you program the co-signed firmware helper image prior to using the PUF enrollment command.

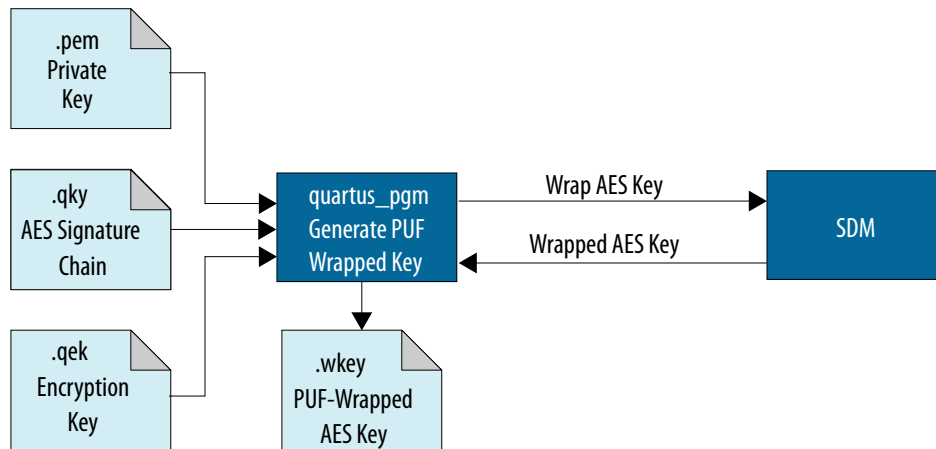
```
quartus_pgm -c 1 -m jtag -o "p;signed_provision_helper_image.rbf" --force
quartus_pgm -c 1 -m jtag -o "e;help_data.puf;1SX280LH2"
```

#### 4.5.2.2. Wrapping the AES Root Key

You generate the IID PUF wrapped AES root key (`.wkey`) file by sending a signed certificate to the SDM.

You can use the Intel Quartus Prime Programmer to automatically generate, sign, and send the certificate to wrap your AES root key, or you may use the Intel Quartus Prime Programming File Generator to generate an unsigned certificate. You sign the unsigned certificate using your own tools or the Quartus signing tool. You then use the Programmer to send the signed certificate and wrap your AES root key. The signed certificate may be used to program all devices that can validate the signature chain.

**Figure 7. Wrapping the AES Key Using the Intel Quartus Prime Programmer**



1. You may generate the IID PUF wrapped AES root key (.wkey) with the Programmer using the following arguments:
  - The .qky file containing a signature chain with AES root key certificate permission
  - The private .pem file for the last key in the signature chain
  - The .qek file holding the AES root key
  - The 16-byte initialization vector (iv)

```
quartus_pgm -c 1 -m jtag --qky_file=aes0_sign_chain.qky \
--pem_file=aes0_sign_private.pem --qek_file=aes.qek \
--iv=1234567890ABCDEF1234567890ABCDEF -o "ei:aes.wkey;1SX280LH2"
```

2. Alternatively, you may generate an unsigned IID PUF wrapping AES root key certificate with the Programming File Generator using the following arguments:

```
quartus_pfg --ccert -o ccert_type=IID_PUF_WRAPPED_AES_KEY \
-o qek_file=aes.qek --iv=1234567890ABCDEF1234567890ABCDEF unsigned_aes.ccert
```

3. You sign the unsigned certificate with your own signing tools or the quartus\_sign tool using the following command:

```
quartus_sign --family=stratix10 --operation=sign \
--qky=aes0_sign_chain.qky --pem=aes0_sign_private.pem \
unsigned_aes.ccert signed_aes.ccert
```

4. You then use the Programmer to send the signed AES certificate and return the wrapped key (.wkey) file:

```
quartus_pgm -c 1 -m jtag --ccert_file=signed_aes.ccert \
-o "ei:aes.wkey;1SX280LH2"
```

**Note:** The i operation is not necessary if you previously loaded the provision firmware helper image, for example, to enroll the PUF.

#### 4.5.2.3. Programming Helper Data and Wrapped Key to QSPI Flash Memory

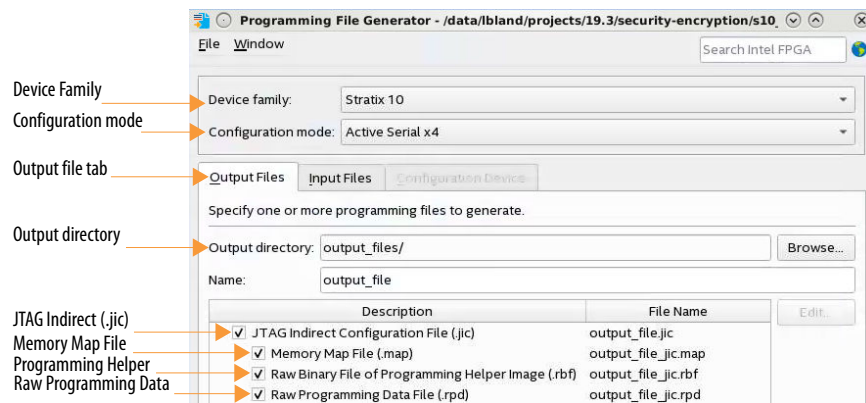
You use the Quartus Programming File Generator graphical interface to build an initial QSPI flash image containing a PUF partition. You must generate and program an entire flash programming image to add a PUF partition to the QSPI flash. Creation of the PUF

data partition and use of the PUF helper data and wrapped key files for flash image generation is not supported through the Programming File Generator command line interface.

The following steps demonstrate building a flash programming image with the PUF helper data and wrapped key:

1. On the **File** menu, click **Programming File Generator**. On the **Output Files** tab make the following selections:
  - a. For **Device Family** select **Stratix 10**.
  - b. For **Configuration mode** select **Active Serial x4**.
  - c. For **Output directory** browse to your output file directory. This example uses **output\_files**
  - d. For **Name**, specify a name for the programming file to be generated. This example uses **output\_file**.
  - e. Under **Description** select the programming files to generate. This example generates the **JTAG Indirect configuration File (.jic)** for device configuration and the **Raw Binary File of Programming Helper Image (.rbf)** for device helper image. This example also selects the optional **Memory Map File (.map)** and **Raw Programming Data File (.rpd)**. The raw programming data file is necessary only if you plan to use a third-party programmer in the future.

**Figure 8. Programming File Generator - Output Files Tab - Select JTAG Indirect Configuration**

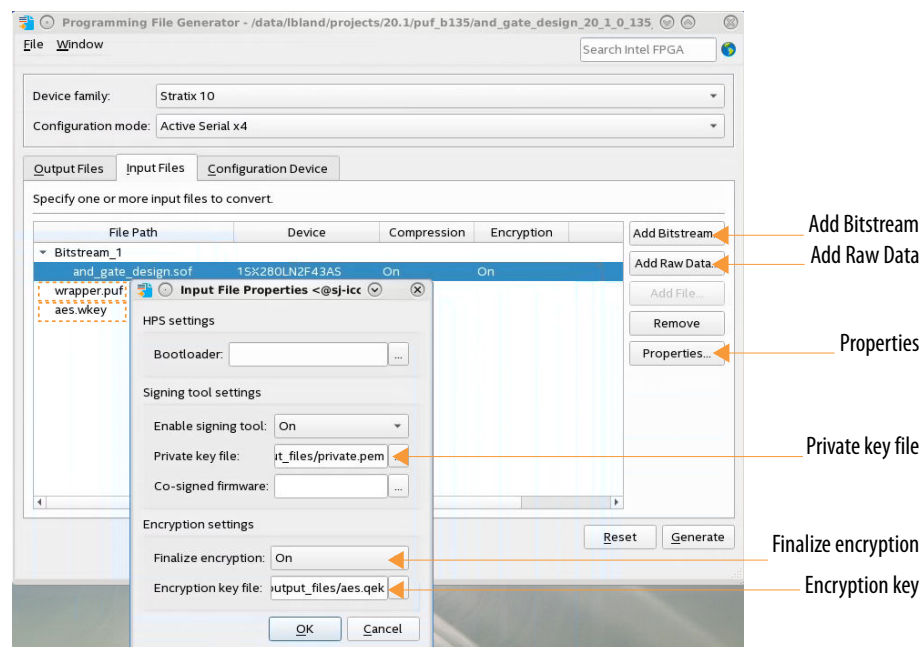


On the **Input Files** tab, make the following selections:

1. Click **Add Bitstream** and browse to your **.sof**.
2. Select your **.sof** file and then click **Properties**.

- a. Turn **On Enable signing tool**.
  - b. For **Private key file** select your `.pem` file.
  - c. Turn **On Finalize encryption**.
  - d. For **Encryption key file** select your `.qek` file.
  - e. Click **OK** to return to the prior window.
3. To specify your PUF helper data file, click **Add Raw Data**. Change the **Files of type** drop-down menu to **Quartus Physical Unclonable Function File (\*.puf)**. Browse to your `.puf` file.
  4. To specify your wrapped AES key file, click **Add Raw Data**. Change the **Files of type** drop-down menu to **Quartus Wrapped Key File (\*.wkey)**. Browse to your `.wkey` file.

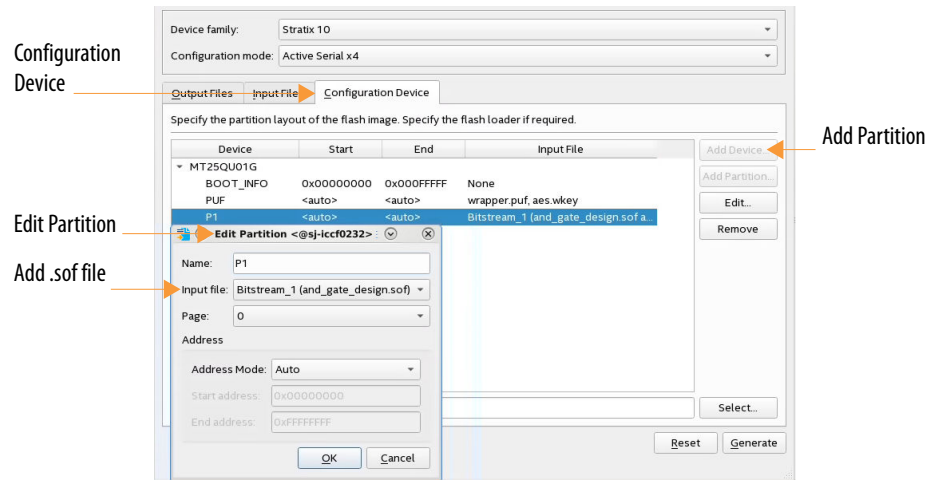
**Figure 9. Specify Input Files for Configuration, Authentication, and Encryption**



On the **Configuration Device** tab, make the following selections:

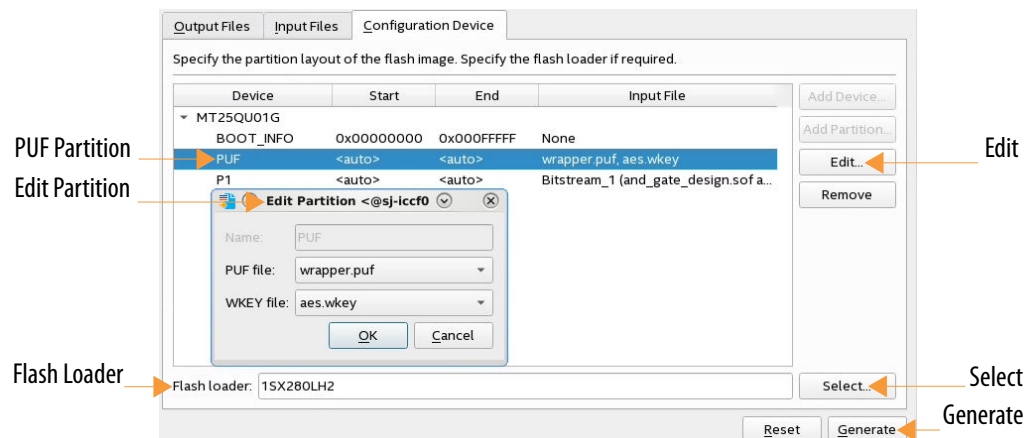
1. Click **Add Device** and select your flash device from the list of available flash devices.
2. Select the configuration device you have just added and click **Add Partition**.
3. In the **Edit Partition** dialog box for the **Input file** and select your `.sof` from the dropdown list. You can retain the defaults or edit the other parameters in the **Edit Partition** dialog box.

Figure 10. Specifying your .sof Configuration Bitstream Partition



- When you add the .puf and .wkey as input files, the Programming File Generator automatically creates a PUF partition in your **Configuration Device**. To store the .puf and .wkey in the **PUF** partition, select the **PUF** partition and click **Edit**. In the **Edit Partition** dialog box, select your .puf and .wkey files from the drop-down lists. If you remove the PUF partition, you must remove and re-add the configuration device for the Programming File Generator to create another PUF partition.

Figure 11. Add the .puf and .wkey files to the PUF Partition



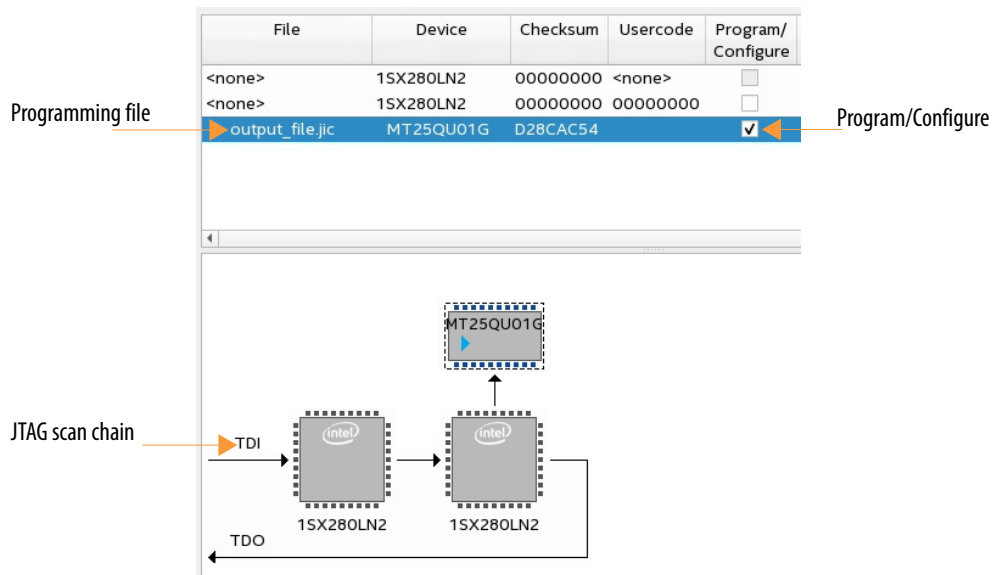
- For the **Flash Loader** parameter select the Intel Stratix 10 device family and device name that matches your Intel Stratix 10 OPN.
- Click **Generate** to generate the output files that you specified on the **Output Files** tab.
- The Programming File Generator reads your .qek file and prompts you for your passphrase if you generated your AES key with a passphrase. Type your passphrase in response to the **Enter QEK passphrase** prompt. Click the Enter key.
- Click **OK** when the Programming File Generator reports successful generation.



You use the Intel Quartus Prime Programmer to write the QSPI programming image to QSPI flash memory.

1. On the Intel Quartus Prime **Tools** menu select **Programmer**.
2. In the Programmer, click **Hardware Setup** and then select a connected Intel FPGA Download Cable.
3. Click **Add File** and browse to your .jic file.

Figure 12. Program .jic



4. Unselect the box associated with the Helper image.
5. Select **Program/Configure** for the .jic output file.
6. Turn on **Start** button to program your quad SPI flash memory.
7. Power cycle your board. The design programmed to the quad SPI flash memory device subsequently loads into the target FPGA.

You must generate and program an entire flash programming image to add a PUF partition to the quad SPI flash.

When a PUF partition already exists in the flash, it is possible to use the Intel Quartus Prime Programmer to directly access the PUF helper data and wrapped key files. For example, if activation is unsuccessful, it is possible to re-enroll the PUF, re-wrap the AES key, and subsequently only program the PUF files without having to overwrite the entire flash.

The Intel Quartus Prime Programmer supports the following operation argument for PUF files in a pre-existing PUF partition:

- **p**: program
- **v**: verify
- **r**: erase
- **b**: blank check

You must follow the same restrictions for PUF enrollment, even if a PUF partition exists.

1. Use the `i` operation argument to load the provision firmware helper image for the first operation. For example, the following command sequence re-enrolls the PUF, re-wrap the AES root key, erase the old PUF helper data and wrapped key, then program and verify the new PUF helper data and AES root key.

```
quartus_pgm -c 1 -m jtag -o "ei;new.puf;1SX280LH2"
quartus_pgm -c 1 -m jtag --ccert_file=signed_aes.ccert \
-o "e;new.wkey;1SX280LH2"
quartus_pgm -c 1 -m jtag -o "r;old.puf"
quartus_pgm -c 1 -m jtag -o "r;old.wkey"
quartus_pgm -c 1 -m jtag -o "p;new.puf"
quartus_pgm -c 1 -m jtag -o "p;new.wkey"
quartus_pgm -c 1 -m jtag -o "v;new.puf"
quartus_pgm -c 1 -m jtag -o "v;new.wkey"
```

#### 4.5.2.4. Querying Intrinsic ID PUF Activation Status

After you enroll the Intrinsic ID PUF, wrap an AES key, generate the flash programming files, and update the quad SPI flash, you power cycle your device to trigger PUF activation and configuration from the encrypted bitstream. The SDM reports the PUF activation status along with the configuration status. If PUF activation fails, the SDM instead reports the PUF error status. Use the `quartus_pgm` command to query the configuration status.

1. Use the following command to query the activation status:

```
quartus_pgm -c 1 -m jtag --status --status_type="CONFIG"
```

Here is sample output from a successful activation:

```
Response of CONFIG_STATUS
Device is running in user mode
00006000 RESPONSE_CODE=OK, LENGTH=6
00000000 STATE=IDLE
00000000 Version
C000000F MSEL=JTAG, nSTATUS=1, nCONFIG=1
80000002 CONF_DONE=0, INIT_DONE=1, CVP_DONE=0,
SEU_ERROR=0, PROVISION_FW=1
00000000 Error location
00000000 Error details
Response of PUF_STATUS
00001000 RESPONSE_CODE=OK, LENGTH=1
00000400 STATUS=PUF_ACTIVATION_SUCCESS,
RELIABILITY_DIAGNOSTIC_SCORE=4
```

#### 4.5.2.5. Location of the PUF in Flash Memory

The location of the PUF file is different for designs that support RSU and designs that do not support the RSU feature.

For designs that do not support RSU, you must include the `.puf` and `.wkey` files when you create updated flash images. For designs that support RSU, the SDM does not overwrite the PUF data sections during factory or application image updates.

**Table 1. Flash Sub-Partitions Layout without RSU Support**

Flash Offset	Size (in bytes)	Contents	Description
0 K	256 K	Configuration Management Firmware	Firmware that runs on SDM.
256 K	256 K	Configuration Management Firmware	
512 K	256 K	Configuration Management Firmware	
768 K	256 K	Configuration Management Firmware	
1M	32 K	PUF data copy 0	Data structure for storing PUF helper data and PUF-wrapped AES root key copy 0
1M+32 K	32 K	PUF data copy 1	Data structure for storing PUF helper data and PUF-wrapped AES root key copy 1

**Table 2. Flash Sub-Partitions Layout with RSU Support**

Flash Offset	Size (in bytes)	Contents	Description
0 K	256 K	Decision firmware	Firmware to identify and load the highest priority image.
256 K	256 K	Decision firmware	
512 K	256 K	Decision firmware	
768 K	256 K	Decision firmware	
1 M	8 K + 24K Padding	Decision firmware data	Reserved for Decision firmware use.
1 M + 32 K	Variable	Factory image	A simple image that you create as a backup if all other application images fail to load. This image includes the CMF that runs on the SDM.
Next	32 K	PUF data copy 0	Data structure for storing PUF helper data and PUF-wrapped AES root key copy 0
Next +32 K	32 K	PUF data copy 1	Data structure for storing PUF helper data and PUF-wrapped AES root key copy 1
Next + 256 K	4 K	Sub-partition table copy 0	Data structure to facilitate the management of the flash storage.
Next +32 K	4 K	Sub-partition table copy 1	
Next +32 K	4 K	CMF pointer block copy 0	A list of pointers to application images in order of priority. When you add an image, that image becomes the highest.
Next +32 K		CMF pointer block copy 1	A second copy of the list of pointers to application images.
Variable	Variable	Application image 1	Your first application image.
Variable	Variable	Application image 2	Your second application image.

### 4.5.3. Black Key Provisioning

Intel strongly recommends the use of Intel Stratix 10 FPGAs that have a -BK OPN suffix with the black key provisioning feature. For more information about devices with a -BK suffix and setting up black key provisioning service, please contact your Intel sales representative or Intel Support.

The Intel Quartus Prime Programmer assists in establishing a mutually-authenticated secure connection between the Intel Stratix 10 device and the black key provisioning service. The secure connection is established via https and requires several certificates identified using a text file.

The `bkp_tls_ca_cert` certificate authenticates your black key provisioning service instance to your black key provisioning programmer instance.

The `bkp_tls_*` certificates authenticate your black key provisioning programmer instance to your black key provisioning service instance.

You create a text file containing the necessary information for the Intel Quartus Prime Programmer to connect to the black key provisioning service. To initiate black key provisioning, use the Programmer command line interface to specify the black key provisioning options text file. The black key provisioning then proceeds automatically. For access to the black key provisioning service and associated documentation, please contact Intel Support.

You can enable the black key provisioning using the `quartus_pgm` command:

```
quartus_pgm -c <cable> -m <programming_mode> --device <device_index> \
--bkp_options=bkp_options.txt
```

The command arguments specify the following information:

- `-c`: cable number
- `-m`: specifies the programming mode such as JTAG
- `--device`: specifies a device index on the JTAG chain. Default value is 1.
- `--bkp_options`: specifies a text file containing black key provisioning options.

#### 4.5.3.1. Black Key Provisioning Options

The black key provisioning options is a text file passed to the Programmer through the `quartus_pgm` command. The file contains required information to trigger black key provisioning.

Example of `bkp_options.txt` file.

```
bkp_cfg_id = 1
bkp_ip = 192.167.1.1
bkp_port = 10034
bkp_tls_ca_cert = root.cert
bkp_tls_prog_cert = prog.cert
bkp_tls_prog_key = prog_key.pem
bkp_tls_prog_key_pass = 1234
bkp_proxy_address = https://192.167.5.5:5000
bkp_proxy_user = proxy_user
bkp_proxy_password = proxy_password
```

**Table 3. Black Key Provisioning Options**

This table displays the options required to trigger black key provisioning.

Option Name	Type	Description
<code>bkp_ip</code>	Required	Specifies the server IP address running the black key provisioning service.
<code>bkp_port</code>	Required	Specifies black key provisioning service port required to connect to the server.
<i>continued...</i>		

Option Name	Type	Description
bkp_cfg_id	Required	Identifies the black key provisioning configuration flow ID. Black key provisioning service creates the black key provisioning configuration flows including an AES root key, desired eFuse settings, and other black key provisioning authorization options. The number assigned during the black key provisioning service setup identifies the black key provisioning configuration flows. <i>Note:</i> Multiple devices may refer to the same black key provisioning service configuration flow.
bkp_tls_ca_cert	Required	The root TLS certificate used to identify the black key provisioning services to the Intel Quartus Prime Programmer (Programmer). A trusted Certificate Authority for the black key provisioning service instance issues this certificate. If you run the Programmer on a computer with Microsoft® Windows® operating system (Windows), you must install this certificate in the Windows certificate store.
bkp_tls_prog_cert	Required	A certificate created for the instance of the black key provisioning Programmer (BKP Programmer). This is the https client certificate used to identify this BKP programmer instance to the black key provisioning service. You must install and authorize this certificate in the black key provisioning service prior to initiating a black key provisioning session. If you run the Programmer on Windows, this option is not available. In this case, the bkp_tls_prog_key already includes this certificate.
bkp_tls_prog_key	Required	The private key corresponding to the BKP Programmer certificate. The key validates the identity of the BKP Programmer instance to black key provisioning service. If you run the Programmer on Windows, the .pfx file combines the bkp_tls_prog_cert certificate and the private key. The bkp_tls_prog_key option passes the .pfx file in the bkp_options.txt file.
bkp_tls_prog_key_pass	Optional	The password for the bkp_tls_prog_key private key. Not required in the black key provisioning configuration options (bkp_options.txt) text file.
bkp_proxy_address	Optional	Specifies the proxy server URL address.
bkp_proxy_user	Optional	Specifies the proxy server user name.
bkp_proxy_password	Optional	Specifies the proxy authentication password.

## 4.6. 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=1SX280LH2 root.qky RootKey.jam
quartus_pfg -c -o helper_device=1SX280LH2 root.qky RootKey.jbc
quartus_pfg -c -o helper_device=1SX280LH2 aes.ccert aes_ccert.jam
quartus_pfg -c -o helper_device=1SX280LH2 aes.ccert aes_ccert.jbc
quartus_pfg -c -o helper_device=1SX280LH2 settings.fuse settings_fuse.jam
quartus_pfg -c -o helper_device=1SX280LH2 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:

```
// 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 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;1SX280HH1" --dev_info
```

The debug owner transfers the generated authenticated public key and device information to the device owner. 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=Stratix10 --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=Stratix10 --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.

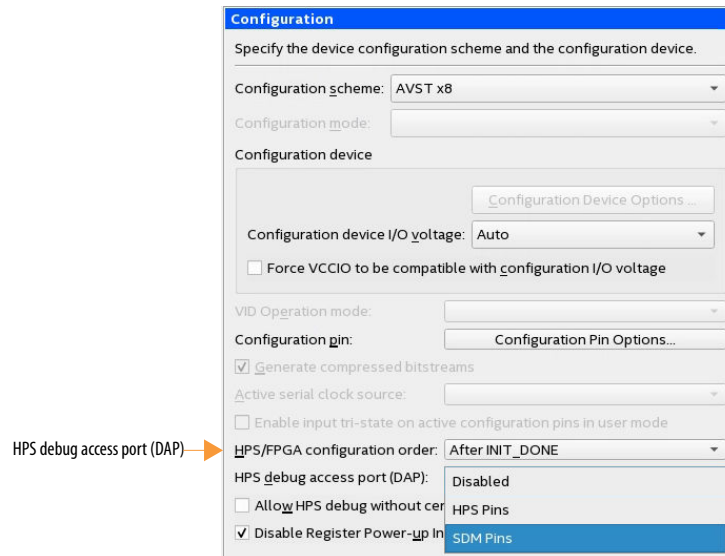
```
quartus_pgm -c 1 -m jtag -o "p;authorized_debug_design.rbf"
```

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.

**Figure 13. 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=Stratix10 --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;1SX280HH2"
```

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

```
quartus_sign --family=stratix10 --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"
```

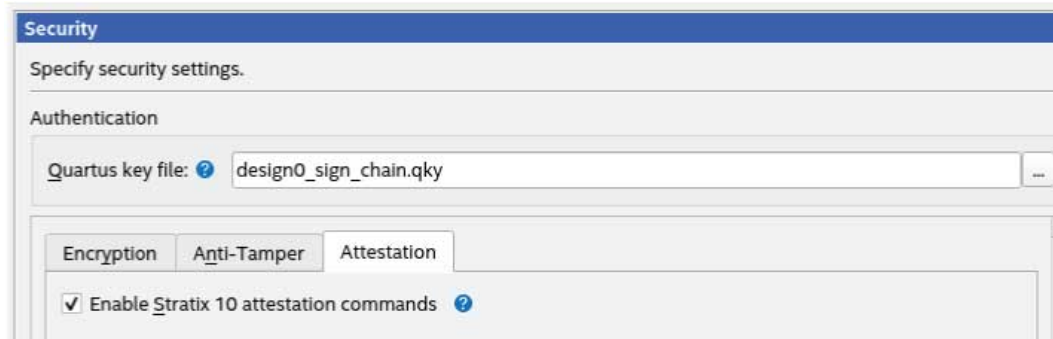
The HPS debug certificate is only valid from the time it was generated until the next power cycle of the device or until a different type or version of SDM firmware is loaded. You must generate, sign, and program the signed HPS debug certificate, and perform all debug operations, prior to power cycling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



### 5.3. Platform Attestation

You enable platform attestation for a given Intel Stratix 10 design by either navigating to the **Device > Device and Pin Options > Security > Attestation** tab and selecting the **Enable Stratix 10 attestation commands** checkbox, or adding the `ENABLE_S10_ATTESTATION_COMMANDS=ON` parameter to the Intel Quartus Prime settings (.qsf) file.

**Figure 14. Selecting Platform Attestation in the Intel Quartus Prime Software**



For access to the attestation verifier service and associated documentation, please contact Intel Support with reference number 14014604265.

Platform attestation is not supported on -V devices. Certain features are not available when the platform attestation feature is enabled. Refer to [Table 4](#) on page 41 for more details.

### 5.4. Physical Anti-Tamper

You enable the physical anti-tamper features using the following steps:

1. Selecting the desired response to a detected tamper event
2. Configuring the desired tamper detection methods and parameters
3. Including the anti-tamper IP in your design logic to help manage anti-tamper events

Anti-tamper features are not supported on -V devices. To enable anti-tamper features, ensure you have a non-V device. For a non-V device, when you use either or both attestation or the anti-tamper security features, you cannot use the Secure Debug Authorization feature, and you must program a virtual owner root key hash using the provision firmware. Similarly, if you need to use Secure Debug Authorization, you cannot turn on either or both a device attestation or the anti-tamper security features in your design.

**Table 4. Available Security Features for a Non-V Device**

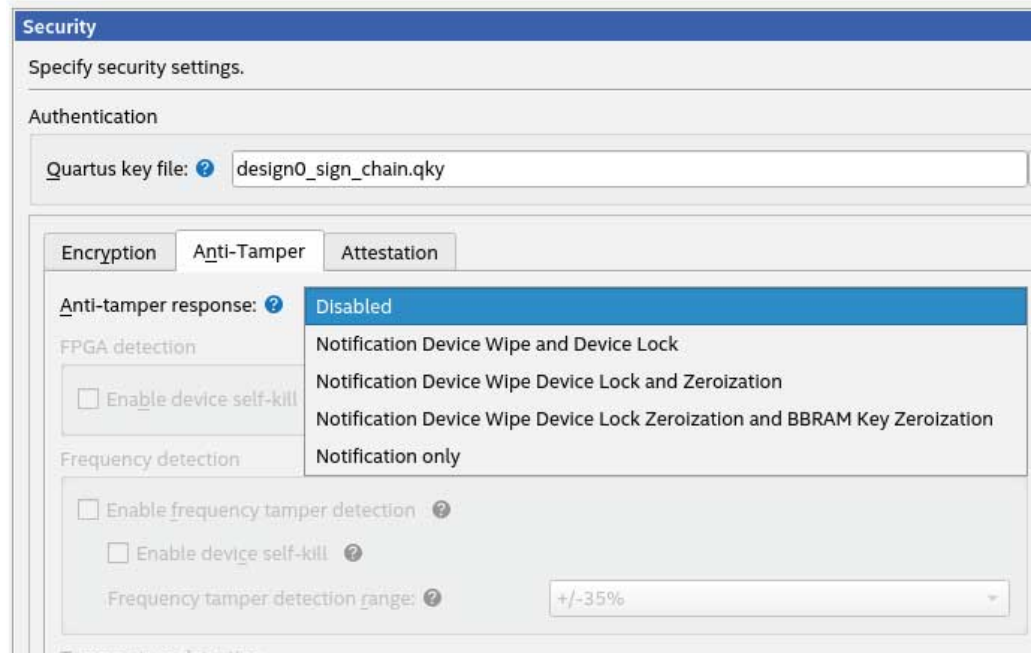
Security Features	Attestation / Anti-Tamper	
	On	Off
Conditional public key entry	Not supported	Supported
Owner root key virtual fusing	Provision firmware only	Provision firmware only

### 5.4.1. Anti-Tamper Responses

You enable physical anti-tamper by selecting a response from the **Anti-tamper response:** dropdown on the **Assignments > Device > Device and Pin Options > Security > Anti-Tamper** tab. By default, the anti-tamper response is disabled.

Five categories of anti-tamper response are available. When you select your desired response, the options to enable one or more detection methods are enabled.

**Figure 15. Available Anti-Tamper Response Options**



You may individually select the **Enable device self-kill** response for each detection method.

If you enable **Enable device self-kill** response for any detection method, you must also generate a permit kill compact certificate, sign the compact certificate, and program the compact certificate to your device prior to loading a design with the self-kill response enabled.

Use one of the following commands to create a signature chain capable of signing a permit-type compact certificate. Note that permission bit 10 is used in this operation.

```
quartus_sign --family=stratix10 --operation=append_key \
--previous_pem=root_private.pem --previous_qky=root.qky \
--permission=0x400 --cancel=0 \
permit0_sign_public.pem permit0_sign_chain.qky
```

```
quartus_sign --family=stratix10 --operation=append_key --module=softHSM \
--module_args="--token_label=s10-token --user_pin=s10-token-pin \
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \
--previous_pem=root --previous_qky=root.qky \
--permission=0x400 --cancel=0 permit0_sign permit0_sign_chain.qky
```

Use the following command to create an unsigned permit kill compact certificate.

```
quartus_pfg --ccert -o ccert_type=DEVICE_PERMIT_KILL unsigned_permit_kill.ccert
```

Use one of the following commands to sign the permit kill compact certificate.

```
quartus_sign --family=stratix10 --operation=sign \  
--pem=permit0_sign_private.pem --qky=permit0_sign_chain.qky \  
unsigned_permit_kill.ccert signed_permit_kill.ccert
```

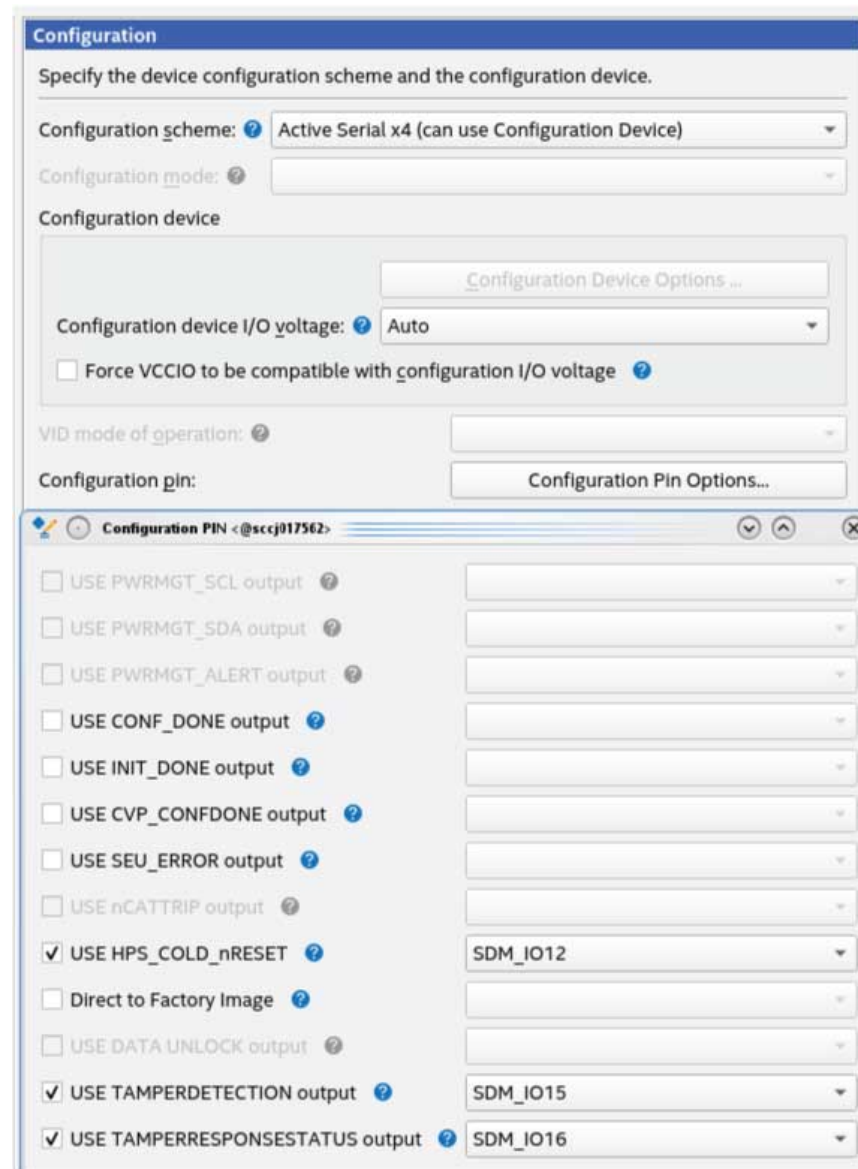
```
quartus_sign --family=stratix10 --operation=sign --module=softHSM \  
--module_args="--token_label=s10-token --user_pin=s10-token-pin \  
--hsm_lib=/usr/local/lib/softhsm/libsofthsm2.so" \  
--pem=permit0_sign --qky=permit0_sign_chain.qky \  
unsigned_permit_kill.ccert signed_permit_kill.ccert
```

Use the following command to program the compact certificate to your device.

```
quartus_pgm -c 1 -m jtag -o "p;signed_permit_kill.ccert"
```

When you enable an anti-tamper response, you may choose two available SDM dedicated I/O pins to output the tamper event detection and response status using the **Assignments > Device > Device and Pin Options > Configuration > Configuration Pin Options** window.

**Figure 16. Available SDM dedicated I/O Pins for Tamper Event Detection**



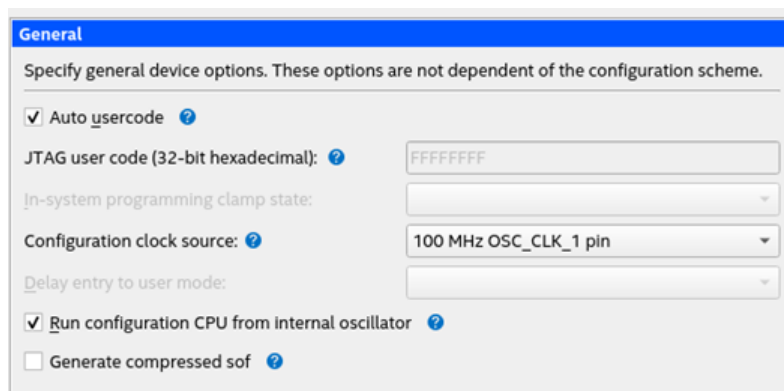
### 5.4.2. Anti-Tamper Detection

You may individually enable the frequency, temperature, and voltage detection features of the SDM. FPGA detection depends on including the Anti-Tamper Lite Intel FPGA IP in your design.

**Note:** SDM frequency and voltage tamper detection methods are dependent on internal references and measurement hardware that can vary across devices. Intel recommends that you characterize the behavior of tamper detection settings.

Frequency tamper detection operates on the configuration clock source. To enable frequency tamper detection, you must specify an option other than **Internal Oscillator** in the **Configuration clock source** dropdown on the **Assignments > Device > Device and Pin Options > General** tab. You must ensure that the **Run configuration CPU from internal oscillator** checkbox is enabled prior to enabling the frequency tamper detection.

Figure 17. Setting the SDM to Internal Oscillator



**General**

Specify general device options. These options are not dependent of the configuration scheme.

☒ Auto usercode ?

JTAG user code (32-bit hexadecimal): ? FFFFFFFF

In-system programming clamp state: [dropdown]

Configuration clock source: ? 100 MHz OSC\_CLK\_1 pin [dropdown]

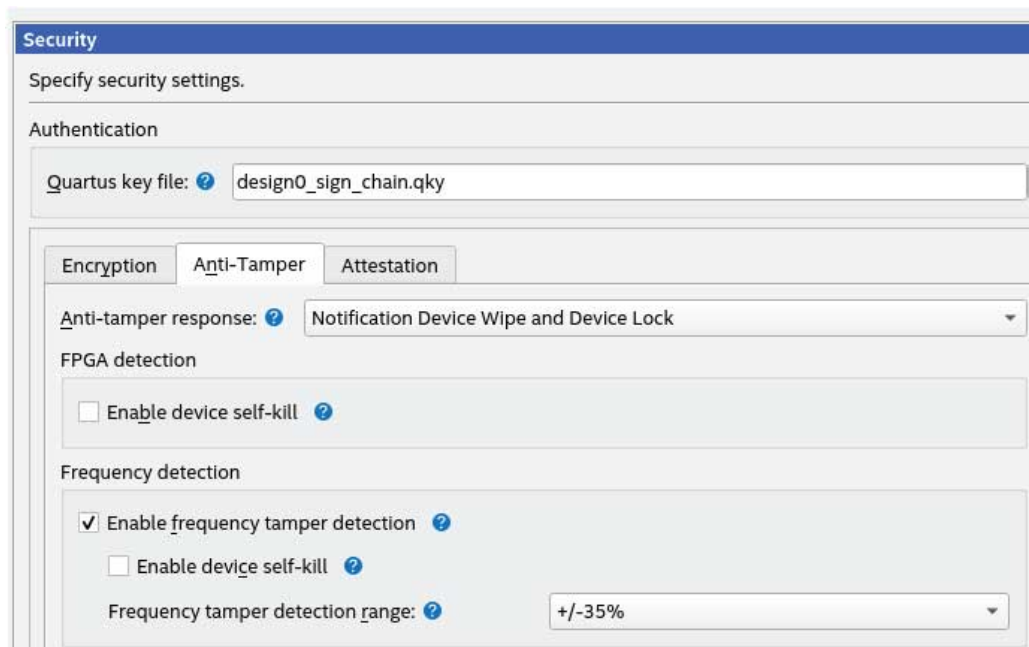
Delay entry to user mode: [dropdown]

☒ Run configuration CPU from internal oscillator ?

☐ Generate compressed sof ?

To enable frequency tamper detection, select the **Enable frequency tamper detection** checkbox and select the desired **Frequency tamper detection range** from the dropdown menu.

Figure 18. Enabling Frequency Tamper Detection



**Security**

Specify security settings.

Authentication

Quartus key file: ? design0\_sign\_chain.qky

Encryption Anti-Tamper Attestation

Anti-tamper response: ? Notification Device Wipe and Device Lock [dropdown]

FPGA detection

☐ Enable device self-kill ?

Frequency detection

☒ Enable frequency tamper detection ?

☐ Enable device self-kill ?

Frequency tamper detection range: ? +/-35% [dropdown]

To enable temperature tamper detection, select the **Enable temperature tamper detection** checkbox and select the desired temperature upper and lower bounds in the corresponding fields. The upper and lower bounds are populated by default with the related temperature range for the device selected in the design.

To enable voltage tamper detection, you select either or both of the **Enable VCCL voltage tamper detection** or **Enable VCCL\_SDM voltage tamper detection** checkboxes and select the desired **Voltage tamper detection trigger** percentage in the corresponding field.

**Figure 19. Enabling Voltage Tamper Detection**

Temperature detection

- ☒ Enable temperature tamper detection
- ☐ Enable device self-kill
- Temperature tamper upper bound: 100°C
- Temperature tamper lower bound: -40°C

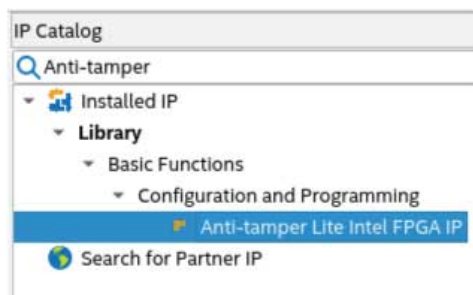
Voltage detection

- ☒ Enable VCCL voltage tamper detection
- ☒ Enable VCCL SDM voltage tamper detection
- ☐ Enable device self-kill
- Voltage tamper detection trigger: 15%

### 5.4.3. Anti-Tamper Lite Intel FPGA IP

The Anti-Tamper Lite Intel FPGA IP, available in the IP catalog in Intel Quartus Prime Pro Edition software, facilitates bidirectional communication between your design and the SDM for tamper events.

**Figure 20. Anti-Tamper Lite Intel FPGA IP**



The IP provides the following signals that you connect to your design as needed:

**Table 5. Anti-Tamper Lite Intel FPGA IP I/O Signals**

Signal Name	Direction	Description
gpo_sdm_at_event	Output	SDM signal to FPGA fabric logic that an SDM has detected a tamper event. The FPGA logic has approximately 5ms to perform any desired cleaning and respond to the SDM via <code>gpi_fpga_at_response_done</code> and <code>gpi_fpga_at_zeroization_done</code> . The SDM proceeds with the tamper response actions when <code>gpi_fpga_at_response_done</code> is asserted or after no response is received in the allotted time.
gpi_fpga_at_event	Input	FPGA interrupt to SDM that your designed anti-tamper detection circuitry has detected a tamper event and the SDM tamper response should be triggered.
gpi_fpga_at_response_done	Input	FPGA interrupt to SDM that FPGA logic has performed desired cleaning.
gpi_fpga_at_zeroization_done	Input	FPGA signal to SDM that FPGA logic has completed any desired zeroization of design data. This signal is sampled when <code>gpi_fpga_at_response_done</code> is asserted.

**5.4.3.1. Release Information**

The IP versioning scheme (X.Y.Z) number changes from one software version to another. A change in:

- X indicates a major revision of the IP. If you update your Intel Quartus Prime software, you must regenerate the IP.
- Y indicates the IP includes new features. Regenerate your IP to include these new features.
- Z indicates the IP includes minor changes. Regenerate your IP to include these changes.

**Table 6. Anti-Tamper Lite Intel FPGA IP Release Information**

Item	Description
IP Version	20.1.0
Intel Quartus Prime Version	21.2
Release Date	2021.06.21

## 5.5. Using Design Security Features with Remote System Update

Remote System Update (RSU) is an Intel Stratix 10 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 Stratix 10 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 the *Generating Remote System Update Image Files Using the Programming File Generator* of the *Intel Stratix 10 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_stratix10.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_stratix10.zip
```

### Related Information

[Intel Stratix 10 Configuration User Guide: Generating Remote System Update Image Files Using the Programming File Generator.](#)

## 6. Intel Stratix 10 Device Security User Guide Archives

---

Intel Quartus Prime Version	User Guide
21.2	<a href="#">Intel Stratix 10 Device Security User Guide</a>
21.1	<a href="#">Intel Stratix 10 Device Security User Guide</a>
20.4	<a href="#">Intel Stratix 10 Device Security User Guide</a>
20.3	<a href="#">Intel Stratix 10 Device Security User Guide</a>
20.1	<a href="#">Intel Stratix 10 Device Security User Guide</a>
19.3	<a href="#">Intel Stratix 10 Device Security User Guide</a>
19.1	<a href="#">Intel Stratix 10 Device Security User Guide</a>

---

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.

ISO  
9001:2015  
Registered

## 7. Document Revision History for Intel Stratix 10 Device Security User Guide

Document Version	Intel Quartus Prime Version	Changes
2021.11.09	21.3	<p>Made the following change:</p> <ul style="list-style-type: none"> <li>Added step to extract firmware in <i>Co-Signing SDM Firmware</i>.</li> <li>Corrected minor errors and typos.</li> </ul>
2021.09.02	21.2	<p>Made the following changes:</p> <ul style="list-style-type: none"> <li>Removed <i>Important Notice to Customers Regarding Features Added in Intel Quartus Prime Pro Edition Software Version 21.1</i> and <i>Planned Security Features</i> topics. The anti-tamper feature is supported starting in Intel Quartus Prime Pro Edition software version 21.2.</li> <li>Added information about Intel Support in <i>Intel Stratix 10 Device Security Overview</i>.</li> <li>Revised <i>Creating a Signature Chain</i>. Added information about Hardware Security Module (HSM) and SoftHSM.</li> <li>Updated the <i>Input (.sof) File Properties for Authentication and Encryption</i> figure.</li> <li>Added anti-tamper and attestation features. <ul style="list-style-type: none"> <li>Globally updated sections and screenshots with anti-tamper and attestation content.</li> <li>Updated <i>Platform Attestation</i> and <i>Physical Anti-Tamper</i> sections to include features description and usage.</li> <li>Added new topics: <ul style="list-style-type: none"> <li><i>Anti-Tamper Responses</i></li> <li><i>Anti-Tamper Detection</i></li> <li><i>Anti-Tamper Intel FPGA IP</i></li> </ul> </li> </ul> </li> <li>Globally added HSM instructions to the existing command examples.</li> <li>Revised <i>Secure Debug Authorization</i>. Clarified the debug owner role.</li> </ul>
2021.04.30	21.1	<p>Made the following changes:</p> <ul style="list-style-type: none"> <li>Restructured the entire document as follows: <ul style="list-style-type: none"> <li>The <i>Security Methodology User Guide</i> contains the security features descriptions.</li> <li>This <i>Intel Stratix 10 Device Security User Guide</i> contains specific instructions for Intel Quartus Prime software to implement security features on Intel Stratix 10 FPGA devices.</li> </ul> </li> <li>Updated <i>Using Design Security Features with Remote System Update</i>. Added examples to generate the initial RSU image, application image, and a factory update image.</li> </ul>
2021.02.17	20.4	<p>Made the following changes:</p> <ul style="list-style-type: none"> <li>Revised Encryption section in the <i>Intel Stratix 10 Device Security Overview</i> topic. Added text stating that devices with advanced security enabled can only load a secured firmware.</li> <li>Removed <i>Partial Reconfiguration Bitstream Encryption (PRBE)</i> topic from the <i>Planned Security Features</i> section. Intel Quartus Prime software version 20.4 supports PRBE feature.</li> <li>Updated key permission content in the <i>Signature Block</i> section. Added permission bit used to sign black key provisioning.</li> </ul>
continued...		

Intel Corporation. All rights reserved. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Intel warrants performance of its FPGA and semiconductor products to current specifications in accordance with Intel's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Intel assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Intel. Intel customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.

\*Other names and brands may be claimed as the property of others.

Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> <li>Added firmware ID=8 along with its firmware release in the <i>Intel Firmware IDs</i> table.</li> <li>Restructured content in <i>Powering On In JTAG Mode After Implementing Co-Signed Firmware</i> and <i>Programing eFuses</i> sections.</li> <li>Revised <i>Step 1: Enrolling the Intrinsic ID PUF via JTAG</i>. <ul style="list-style-type: none"> <li>Revised PUF enrollment text to emphasize that the Intel Quartus Prime Pro Edition Programmer automatically loads the provision firmware.</li> <li>Added note suggesting to toggle the <code>nCONFIG</code> signal in order to perform a successful reconfiguration.</li> <li>Updated text to emphasize that Intel Quartus Prime Programmer restricts PUF operations without the appropriate license.</li> </ul> </li> <li>Revised <i>Black Key Provisioning</i> section. Added recommendation to use -BK OPN suffix for devices using black key provisioning features.</li> <li>Updated <i>Partial Reconfiguration Bitstream Encryption</i> topic. Added new sections: <ul style="list-style-type: none"> <li><i>Enabling Partial Reconfiguration Bitstream Encryption</i></li> <li><i>Generating Encrypted Partial Reconfiguration Persona Programming Files Using the Command Line Interface</i></li> <li><i>Generating Partially Encrypted Partial Reconfiguration Partial Reconfiguration Persona Programming Files Using the Command Line Interface</i></li> </ul> </li> <li>Revised <i>Using eFuses</i> topic. Clarified restriction of programming physical eFuses after virtual eFuses.</li> <li>Added new topics: <ul style="list-style-type: none"> <li><i>Signing Tool (with Source Code)</i></li> <li><i>Encryption Tool (with Source Code)</i></li> <li><i>Security Option eFuses</i></li> <li><i>Using Design Security Features with Remote System Update</i></li> </ul> </li> <li>Corrected minor errors and typos.</li> </ul>
2020.10.13	20.3	<p>Made the following changes:</p> <ul style="list-style-type: none"> <li>Revised <i>Important Notice to Customers Regarding Features Added in Intel Quartus Prime Pro Edition Software Version 20.3</i> notice. The IID PUF-based AES key storage feature is a production feature. The anti-tamper features were removed in this release.</li> <li>Added limitation for Intel Stratix 10 GX 10M devices in the <i>Intel Stratix 10 Device Security Overview: Encryption</i> section. The Intel Stratix 10 GX 10 devices don't support the advanced security features.</li> <li>Updated the <i>Owner Security Keys and Storage Options</i> section: <ul style="list-style-type: none"> <li>Revised virtual eFuses, physical eFuses, and BBRAM typical applications in the <i>Comparison of AES Key Storage Options</i> table.</li> <li>Revised Black Key Provisioning description in the <i>Owner AES Key</i> section.</li> <li>Removed <i>Owner AES Key Programming</i> section. The content is already available in the <i>Encryption and Decryption</i> chapter.</li> </ul> </li> <li>Updated the <i>Planned Security Features</i> section: <ul style="list-style-type: none"> <li>Added the <i>Anti-Tampering</i> topic.</li> <li>Added the <i>Partial Reconfiguration Bitstream Encryption</i> topic.</li> <li>Removed the <i>Black Key Provisioning</i> topic.</li> </ul> </li> <li>Updated the <i>Signature Chain Content</i> table: <ul style="list-style-type: none"> <li>Added bit 6: AES root key certificate in the Public Key Entry description.</li> <li>Revised Header Block Entry description.</li> </ul> </li> <li>Updated the <i>Canceling Intel Firmware ID</i> section: <ul style="list-style-type: none"> <li>Added firmware ID=7 along with its firmware release in the <i>Intel Firmware IDs</i> table.</li> <li>Revised and added steps to prevent using older firmware versions once you upgraded to a new firmware version.</li> </ul> </li> <li>Corrected argument value for signing firmware in the <i>Append Key to Signature Chain</i> section. The value to sign firmware is 1, not 0.</li> </ul>

continued...

Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> <li>Added assignment to specify the owner cancellation ID in the <i>Step 4a: Signing the Bitstream Using the Programming File Generator</i> section.</li> <li>Revised note in the <i>Using the Co-Signed Feature</i> section. The text points to the <i>Using eFuses</i> section for information on programming the co-signed eFuses.</li> <li>Revised the <i>Prerequisites for Co-Signing Device Firmware</i> section: <ul style="list-style-type: none"> <li>Rename argument from <code>--key_storage</code> to <code>--non_volatile_key</code>.</li> <li>Updated the <code>quartus_pgm</code> commands.</li> <li>Updated the instruction links.</li> </ul> </li> <li>Added new topic: <i>Step 2c: Generating Partially Encrypted Programming File Using the Command Line Interface</i>.</li> <li>Added Quad SPI Intrinsic ID PUF-wrapped option in the <i>Step 3a: Specifying Keys and Configuring the Encrypted Image Using the Intel Quartus Prime Programmer</i>.</li> <li>Updated description of <code>-i</code> option and added a note in the <i>Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line</i> section. The note states that you can program the co-signed helper image prior to programming the <code>.qek</code> encryption key.</li> <li>Removed <i>Anti-Tamper Monitoring and Mitigation</i> section and all related anti-tamper content.</li> <li>Removed (Beta) label from all IID PUF-based AES key storage content. In this release, IID PUF is a production feature.</li> <li>Revised all PUF-related content.</li> <li>Added support for black key provisioning and new sections describing enabling black key provisioning, including the <code>bkp_options</code> description.</li> <li>Revised <i>Using eFuses</i> and <i>Key Cancellation eFuses</i> topics. <ul style="list-style-type: none"> <li>Added statement to emphasize the usage of virtual eFuses. The virtual eFuses are meant for testing purposed only. They don't guarantee security in the production environment.</li> </ul> </li> <li>Revised the <i>Using an HPS Debug Certificate</i> topic. <ul style="list-style-type: none"> <li>Revised text to emphasize the debug certificate usage in order to ensure the bitstream security: <i>Intel strongly recommends restricting such a bitstream from release and canceling the signing key ID after this configuration bitstream is no longer needed.</i></li> <li>Revised the required conditions to create HPS debug certificate. <ul style="list-style-type: none"> <li>Updated JTAG related condition to emphasize that the JTAG disable fuse disables JTAG.</li> <li>Added new condition to emphasize that the HPS debug disable fuse permanently disables the HPS debugging.</li> </ul> </li> </ul> </li> <li>Updated <i>Enabling HPS JTAG Debugging</i> topic. <ul style="list-style-type: none"> <li>Corrected the statement on the <code>permission=8</code> for the HPS debug certificate.</li> <li>Renamed a key file from <code>&lt;design0_sign_chain.qky&gt;</code> to <code>&lt;debug_cert_sign_chain.qky&gt;</code>.</li> </ul> </li> <li>Updated <code>.puf</code> file description in the <i>File Types for Security</i> appendix.</li> <li>Added new <code>quartus_pgm</code> command arguments in the <i>quartus_pgm Command Operation Argument</i> appendix: <ul style="list-style-type: none"> <li><code>-o p;file.ccert</code></li> <li><code>-o pvbi;file.puf</code></li> <li><code>-o pvbi;file.wkey</code></li> <li><code>-o ei;file.ccert;device_name</code></li> <li><code>-o ei;file.puf;device_name</code></li> <li><code>-o ei;file.wkey;device_name</code></li> </ul> </li> <li>Corrected minor errors and spelling mistakes.</li> </ul>
2020.04.13	20.1	Made the following changes:
continued...		

Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> <li>Added topic: <i>Important Notice to Customers Regarding Features Added in Intel Quartus Prime Pro Edition Software Version 20.1</i>. It states that the IID PUF-based AES key storage and Anti-Tamper features are a beta release in Intel Quartus Prime Pro Edition software version 20.1.</li> <li>Added support for a PUF-wrapped AES key. Refer to <i>Using a PUF-Wrapped AES Key (Beta)</i> for more information. This feature is a beta release in Intel Quartus Prime Pro Edition software version 20.1.</li> <li>Added support for monitors that can trigger an anti-tamper response when the temperature, voltage, or external clock frequency exceeds the values you specify. Refer to <i>Anti-Tamper Monitoring and Mitigation (Beta)</i> for more information. This feature is a beta release in Intel Quartus Prime Pro Edition software version 20.1.</li> <li>Added the <i>Comparison of AES Key Storage Options</i> table that describes the features of each the 4 possible storage locations.</li> <li>Added firmware ID=6 along with its firmware release in the <i>Intel Firmware IDs</i> table.</li> <li>Reorganized user guide.</li> <li>Added support for a <code>quartus_pfg</code> command that checks the integrity of a signed configuration bitstream. Refer to <i>Verifying a Configuration Bitstream Signature</i> for more information.</li> <li>Added an appendix covering acronyms and definitions of security terminology.</li> <li>Added an appendix describing file types that implement security features.</li> <li>Added an appendix showing help for the operation (-o) argument to the <code>quartus_pgm</code> command.</li> <li>Updated Security Category figures to show the new <b>Permitted owner cancellation id</b> and Anti-Tamper tab.</li> <li>Removed statement that the JTAG disable eFuse eliminates boundary scan. In the Intel Quartus Prime Release 20.1 release, disabling JTAG does not disable boundary scan.</li> <li>Corrected minor errors and spelling mistakes.</li> </ul>
2020.01.15	19.3	<p>Corrected the <code>pem_file</code> argument in <i>7.1.3. Step 2b: Generating Programming Files Using the Command Line</i>. The correct command uses <code>pem_file=design0_sign_private.pem</code>:</p> <pre>quartus_pfg -c encryption_enabled.sof top.rbf \ -o finalize_encryption=ON -o gek_file=aes.gek \ -o signing=ON -o pem_file=design0_sign_private.pem</pre>
2020.01.06	19.3	<p>Made the following changes:</p> <ul style="list-style-type: none"> <li>Corrected the <code>quartus_encrypt</code> command in the <i>Step 1: Preparing the Owner Image and AES Key File</i> topic. The <code>ik_count</code> and <code>max_key_use</code> arguments must be preceded by <code>--</code>.</li> <li>Added command showing how to convert an <code>.rbf</code> to <code>.jam</code> format in the <i>Step 4: Signing the Bitstream</i> topic.</li> <li>Added the following note to the <i>Converting Key, Encryption, and Fuse Files to Jam Staple File Formats</i> topic: <p><b>Caution:</b> When you convert the AES <code>.gek</code> file to <code>.jam</code> format, the <code>.jam</code> file contains the AES key in plaintext but obfuscated form. Consequently, you must protect the <code>.jam</code> file when storing the AES key. You can protect the <code>.jam</code> file by provisioning the AES key in a secure environment.</p> </li> <li>Added a link to the <a href="#">How can I write or erase the Intel Stratix 10 AES BBRAM encryption key using the Mailbox Client Intel FPGA IP interface and System Console?</a> article in <i>Storing the AES Key in BBRAM using the JTAG Mailbox</i>.</li> </ul>
2019.10.30	19.3	<p>Added the following new security features:</p>
continued...		

Document Version	Intel Quartus Prime Version	Changes
		<ul style="list-style-type: none"> <li>Added support for physical (non-volatile) eFuses.</li> <li>Changed the way you specify virtual (volatile) or physical (non-volatile) eFuses. The <code>--non_volatile_key</code> parameter is now an argument to the <code>quartus_pgm</code> command. Consequently, you no longer need to recompile to change the eFuse storage location.</li> <li>Increased the number of public keys entries supported from 2 to 3.</li> <li>Added support for a signed secure HPS debug certificate to prevent unauthorized remote or physical access to the HPS.</li> <li>Decreased the encryption update ratio from 127:1 to 31:1.</li> <li>Revised description the <i>Using the Authentication Feature</i> example. The example now specifies permission 6 to allow the key to sign both the Core (permission=2) and HPS (permission=4) sections of the configuration bitstream. You must create separate key chains to limit the permissions to either Core or HPS.</li> <li>Added support for 10 additional eFuses described in the <i>Owner Programmable eFuses</i> table.</li> <li>Added examples of advanced security features.</li> <li>Added descriptions of side-channel mitigation features.</li> <li>Added the following topics: <ul style="list-style-type: none"> <li>Step 4a: Protecting the AES Key when Storing the AES in eFuses</li> <li>Step 4b: Protecting the AES Key when Storing the AES Key in BBRAM</li> <li>Encryption Command Detailed Description</li> <li>Make AES Key</li> <li>Encrypt the Bitstream</li> <li>Programming eFuses</li> <li>Canceling eFuses</li> </ul> </li> <li>Added examples of <code>.jam</code> commands under the <i>Using the .jam Files to Program Root Key and AES Encryption Key</i> heading.</li> <li>Corrected <i>AES Update Mode</i> figure. The number of data bits in a data block is 256, not 128.</li> <li>Corrected the cancellation ID Numbers in <i>Figure 5: Three-Key Signature Chain</i>. The cancellation IDs are 0 and 1.</li> <li>Removed recommendation to use separate signing keys for core and HPS in Intel Stratix 10 SX devices. Changed <i>Using the Authentication Feature</i> example to set permissions to 6 which can sign both the core and HPS.</li> <li>Revised <i>Anti-Tampering</i> topic.</li> <li>Revised the <i>Using eFuses</i> topic.</li> <li>Corrected minor errors and typos.</li> </ul>
2019.05.30	19.1	<p>Made the following corrections:</p> <ul style="list-style-type: none"> <li>Corrected the <i>Signing Command Argument Summary</i> table. The references to <code>.key</code> format should say <code>.qky</code> format.</li> </ul>
2019.05.10	19.1	<p>Made the following corrections:</p> <ul style="list-style-type: none"> <li>Removed spaces before the fuse programming file name in the <code>quartus_pgm</code> commands in <i>Step 3b: Programming the AES Key and Configuring the Encrypted Image Using the Command Line</i>.</li> <li>Changed file name argument to <code>-o "p;my_fuse.fuse"</code> in <i>Step 4 of Canceling Non-Volatile eFuses</i>.</li> </ul>
2019.05.07	19.1	Initial release.