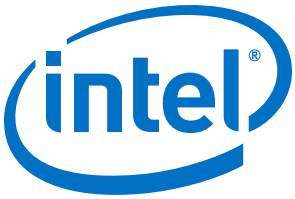


Security User Guide

Intel® Programmable Acceleration Card with Intel® Arria® 10 GX FPGA

Subscribe
Send Feedback

UG-20263 | 2020.03.06
Latest document on the web: PDF | HTML

Contents

1. Overview

1.1. About This Document

Reference this user guide to understand and enable the security features such as the Trusted Control Module (TCM) and AFU signing for the Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA (Intel PAC with Intel Arria 10 GX FPGA).

1.2. Prerequisites

You must ensure that the host and Intel FPGA PAC are using the current version of OPAE tools. Please refer to the latest version of the User Guide for your Intel FPGA PAC for directions on how to determine if you have the current version of tools.

1.3. Related Documentation

Refer to the following documentation while using this guide:

Table 1. Related Documentation

Document	Description
Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA	How to install and update OPAE and the FPGA Interface Manager (FIM).

Related Information

Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

1.4. Glossary

Table 2. Glossary

Acronym/Term	Expansion	Description
AFU	Accelerator Functional Unit	Hardware Accelerator implemented in FPGA logic which offloads a computational operation for an application from the CPU to improve performance.
ASE	AFU Simulation Environment	Co-simulation environment that allows you to use the same host application and AF in a simulation environment. ASE is part of the Intel Acceleration Stack for FPGAs.

continued...

Acronym/Term	Expansion	Description
BIP	Bitstream Authentication IP	Module loaded into the Intel Arria 10 GX FPGA PR region that contains the cryptographic blocks necessary to perform bitstream authentication operations.
CCI-P	Core Cache Interface	CCI-P is the standard interface AFUs use to communicate with the host.
CSK	Code Signing Key	A key used to validate integrity and authenticity of a block of code. Authenticity of this key is established through signing with a root key.
ECDSA	Elliptical Curve Digital Signature Algorithm	An algorithm based on elliptic curve cryptography to create signatures that can be used to evaluate the authenticity of an object.
FIU	FPGA Interface Unit	FIU is a platform interface layer that acts as a bridge between platform interfaces like PCIe* and AFU-side interfaces such as CCI-P.
FIM	FPGA Interface Manager	The FPGA functional block containing the FPGA Interface Unit (FIU) and external interfaces for memory, networking, etc. The FIM may also be referred to as BBS (Blue-Bits, Blue Bit Stream) in the Acceleration Stack installation directory tree and in source code comments. The Accelerator Function (AF) interfaces with the FIM at run time. The FIM is provided with the Intel PAC with Intel Arria 10 GX FPGA.
HSM	Hardware Security Module	A secure hardware device to hold, protect, and allow access to cryptographic objects; performs cryptographic operations in a trusted environment.
NIST p Curve	National Institute of Standards and Technology prime Curve	P256 is used throughout this document. Without any other associations added, P256 means NIST P256 curves, where p is a 256-bit prime.
OPAE	Open Programmable Acceleration Engine	The OPAE is a software framework for managing and accessing AFs.
PACSign	PAC image signing tool	A standalone tool to manage root entry hash bitstream creation, image signing, and cancellation bitstream creation
PKCS	Public Key Cryptography Standard	PKCS#11 is used throughout this document. PKCS#11 is a commonly used interface for commercial hardware security modules (HSMs).
PR	Partial Reconfiguration	The ability to dynamically reconfigure a portion of an FPGA while the remaining FPGA design continues to function.
Root Key	-	A key designated as the primary, constant value for authentication. Typically only used to sign other keys, forming the root of all key chains.
RoT	Root of Trust	A source that can be trusted, such as the TCM in the Intel FPGA PAC.

continued...

Acronym/Term	Expansion	Description
RSU	Remote System Update	Ability to update firmware and FPGA bitstreams over PCIe.
SR	Static Region	Portion of the FPGA design that does not change. In the Intel PAC with Intel Arria 10 GX FPGA, the static region is the FIM
TCM	Trusted Control Module	Functionality implemented in the SR of the Intel PAC with Intel Arria 10 GX FPGA to manage the secure updates of BMC firmware, FIM updates, GBS updates, and key cancellation.

2. Intel FPGA PAC Security Features

The Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA contains logic in the static region (SR) called the Trusted Control Module (TCM). The TCM acts as a Root of Trust (RoT) and enables the secure update features of the Intel FPGA PAC. The TCM RoT includes features that may help prevent the following:

- Loading or executing of unauthorized code or designs.
- Disruptive operations attempted by unprivileged software, privileged software, or the host BMC.
- Unintended execution of older code or designs with known bugs or vulnerabilities by enabling the TCM to revoke authorization.

The TCM RoT also enforces several other security policies relating to access through various interfaces, as well as protecting the on-board flash through write rate limitation.

The TCM RoT verifies:

- Board Management Controller (BMC) firmware updates
- FIM images.
- AFU (partial reconfiguration region) images.

The TCM RoT is programmed with Intel root entry hashes for Intel FIM images during a one-time secure update (OTSU) to preproduction units or at manufacturing, but does not contain a root entry hash for AFUs. You must create your AFU root entry hash bitstream using the PACSign tool provided by Intel. The TCM RoT accepts and programs exactly one AFU root entry hash bitstream.

Note: This operation cannot be reversed, and after this operation, AFUs without correct signatures are refused by the Intel PAC with Intel Arria 10 GX FPGA. A correct signature is one created by a Code Signing Key (CSK) that is both signed by the root key and not yet canceled.

In cases where you have a pre-security production Intel FPGA PAC, you must perform a one-time secure update. Please refer to the *One-Time Secure Update* section in the *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* for more information.

Related Information

Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

2.1. Secure Image Updates

The TCM RoT requires that all BMC firmware and FIM images are authenticated using ECDSA before loading and executing on the card. The TCM RoT may optionally require that AFU images are authenticated before loading and executing as well. The TCM RoT achieves this by storing the root entry hashes for the three image types in a write-once location in on-board flash memory, and subsequently verifying the signature of all images against these hashes. The board manufacturer provides the root entry hash for the BMC firmware. Intel provides the root entry hash for FIM images. You create and program the root entry hash bitstream for AFU images. Until you program the AFU root entry hash bitstream, the Intel FPGA PAC does not authenticate an AFU image prior to loading and executing the image

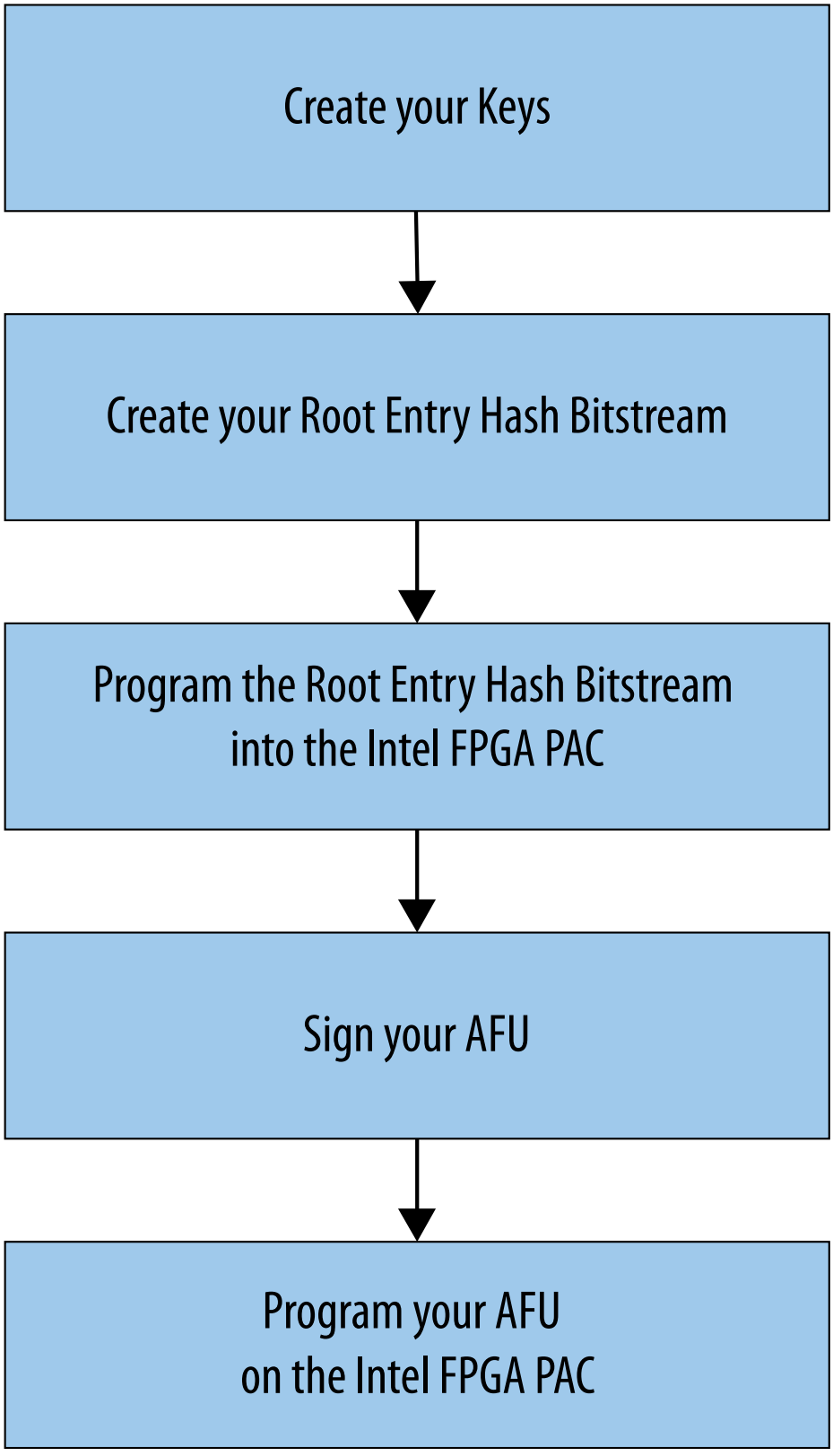
Table 3. Keys and Authentication

Root Key	Origin	Used to Authenticate
BMC root key	Intel FPGA PAC manufacturer	BMC Firmware Updates
Intel FIM root key	Intel	Intel FIM Updates
Partial reconfiguration (PR) AFU root key	Customer	AFUs

When you are in the development or validation phase and have not programmed your root entry hash bitstream, you create AFU images that contain the appropriate headers but are not signed using keys. This process is called creating an unsigned image. An Intel FPGA PAC that has not had the AFU root entry hash bitstream programmed runs any unsigned or signed AFU image. This capability allows you to test and validate the functionality of your AFU image prior to fully signing the image for deployment into a production environment. Please refer to the *Creating Unsigned Images* section for more information.

You program your AFU root entry hash bitstream to enable AFU image authentication. This process establishes you as the owner of the Intel PAC with Intel Arria 10 GX FPGA. The Intel PAC with Intel Arria 10 GX FPGA then requires you to create signatures based on this root entry for each AFU you intend to load on the Intel FPGA PAC. Intel strongly recommends that you program the root entry hash bitstream for Intel FPGA PACs used in production environments. You must follow the following flow to enable user AFU image authentication on your Intel FPGA PAC.

Figure 1. Secure User Image Flow



The chapters within this user guide cover the steps in this flow:

1. **Create your keys:** Create your keys using a Hardware Security Module (HSM) or OpenSSL. You need at least two keys, one which you designate as a root key and another you designate as a code signing key (CSK). These keys are asymmetric keys, meaning they consist of an underlying pair of keys. The first is called a private key and the second is a public key that is derived from the private key. A private key is used to create signatures over objects that can be verified with the corresponding public key. The private key must be kept confidential, as anyone in possession of the private key can create a signature; conversely, if you maintain the confidentiality of the private key, then signatures can be trusted to originate only from you. The public key cannot create signatures or be used to derive the original private key. Therefore, it is not required nor important to protect the confidentiality of the public key; the public key is considered public information.
2. **Create your root entry hash bitstream:** Use the PACSign tool to create a bitstream that contains the root entry hash. You create a root entry hash bitstream from your root public key. This hash is a representation of your root public key and can only be created with an exact copy of the root public key. The root entry hash bitstream is then programmed to the Intel FPGA PAC. The Intel FPGA PAC then uses this hash to verify the integrity of the root public key, which is included with all images transmitted to an Intel FPGA PAC. After the integrity of the root public key is confirmed, it can be used in the signature verification process.
3. **Program your root entry hash bitstream into the Intel FPGA PAC**. You must use the `fpgasupdate` command to program the bitstream containing your root entry hash into the flash on the board. Until you program the root entry hash bitstream, the Intel FPGA PAC loads and executes any signed or unsigned image. Intel strongly recommends that you create and program a root entry hash bitstream for Intel FPGA PACs deployed in production environments. Please refer to the *Using fpgasupdate* chapter for more information.

   Note: Only the owner who is deploying the Intel FPGA PAC must program the root entry hash bitstream.
4. **Sign your AFU image.** Using PACSign you can sign your image with the root public key and code signing key. Please refer to the *Intel FPGA PAC Security Flow* chapter for more information.
5. **Program your AFU image onto the Intel FPGA PAC.** Use the `fpgasupdate` command to program your AFU into flash. The Intel FPGA PAC verifies the AFU to ensure only an authorized bitstream is loaded. The root public key, code signing public key, signature of the code signing public key, and signature of the code or design are all attached to the image transmitted to the Intel FPGA PAC. The card first verifies the integrity of the root public key, then verifies the signature of the code signing public key using the root public key, and finally proceeds to verify the signature of the code or design using the code signing public key. The code or design is only accepted if all three of these steps are completed successfully.

Related Information

- Creating Unsigned Images on page 17
- Intel FPGA PAC Security Flow on page 13

2.2. Anti-Rollback Capability

The TCM RoT provides anti-rollback capability through the code signing key ID cancellation feature. A CSK is assigned an ID, a number between 0-127, during the signing process. CSK ID cancellation information is stored in 128-bit fields in write-once locations in flash. When a code signing key ID is canceled, the TCM RoT rejects all signatures created with a CSK that is assigned that ID. If a CSK ID that is used in an old update is canceled after applying a new update with a different CSK ID, the TCM RoT rejects the signature of the old update, preventing a rollback to the old update.

Note: If you cancel an AFU CSK ID and do not update your AFU image, the image continues to be operational until you update it. The new image must be signed with a CSK that is assigned an uncanceled ID.

2.3. Key Management

The Intel TCM RoT uses ECDSA with a key length of 256 bits to authenticate:

- BMC firmware update images
- FIM images
- AFU (partial reconfiguration) images

The Intel TCM RoT supports separate key chains for each image, and each key chain must consist of a root key and a CSK.

The Intel TCM RoT does not support a signature of any image with a root key. You must use a key designated as a CSK to sign your image. Steps you are responsible for when creating keys, root entry hashes and programming your image on the Intel FPGA PAC are:

- You must manage assigning CSK IDs to CSKs and consistently using the same ID for a given CSK. Neither an Intel FPGA PAC nor the PACSign tool associate a particular key's value with its ID. It is possible to assign a given CSK multiple IDs, or multiple CSKs to a given ID. This may result in unintended consequences when attempting to cancel a CSK. Intel recommends exclusive ID assignments for each CSK.
- You are responsible for creating the appropriate key cancellation bitstreams. You must use the same ID number for key cancellation as the one you assigned to the CSK at key creation. Key cancellation bitstreams must be signed with the applicable root key. This helps avoid denial of service through an unintended cancellation of all key values.
- You are responsible for generating and managing your AFU image root key and CSKs. You generate the AFU image root entry hash bitstream using your root key.
- You are also responsible for programming this root entry hash bitstream on the Intel FPGA PAC. If your Intel FPGA PAC does not have a programmed AFU root entry hash bitstream stored, it executes any signed or unsigned AFU.

  Note: Intel strongly recommends programming an AFU root entry hash bitstream. You must protect the confidentiality of the root private key throughout the life of the Intel FPGA PAC.

The Intel TCM RoT stores a root entry hash bitstreams in the on-board flash for:

1. BMC firmware images
2. FIM images
3. AFU (partial reconfiguration region) images

The TCM is architected so that all root entry hashes cannot be revoked, changed, or erased once programmed.

If you have a board that has not been updated with the TCM RoT, you must use the one-time secure update to program the Intel root entry hash bitstreams for the BMC firmware and Intel FIM images on your existing Intel FPGA PAC. New Intel FPGA PACs come with these root entry hashes programmed at manufacturing time.

The *Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA* further describes:

- Determining whether your board has been updated with the required hashes
- Using one-time secure update

In the future, updates to the BMC firmware or FIM images may necessitate a respective key cancellation in order to help prevent an unintended rollback to a prior version. In this case, Intel provides the update with a signed CSK that has a different ID than all prior updates. Intel provides a separate key cancellation bitstream to cancel the appropriate Intel keys. You may test an update by applying it before programming the key cancellation bitstream. The prior BMC firmware or FIM update images continue to be accepted as valid updates until the new key cancellation bitstream is applied.

Related Information

Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

2.4. Authentication

To enable authentication:

1. Use the PACSign tool to create a root entry hash bitstream.
2. Use the `fpgasupdate` tool to program the bitstream onto the Intel FPGA PAC.

```
$ sudo fpgasupdate [--log-level=<level>] file [bdf]
```

Note: After the root entry hash bitstream is programmed, the Intel FPGA PAC must be power cycled.

All key operations are done using PACSign. PACSign is a standalone tool that is not required to be run on a machine with the Intel FPGA PAC installed. Key creation, signing, and cancellation bitstream creation are not runtime operations and can be performed at any time. The signing process prepends the signature to the AFU image file. The TCM RoT does not need access to the HSM at any point to verify a signature.

The signing process requires a root key and a Code Signing Key (CSK). PACSign first signs the CSK with the root key, and then signs the image with the CSK. The signature process prepends two “blocks” of data to the image file.

Note: If you are using an Intel Acceleration Stack version 1.2.1 or greater, your AFUs must have prepended signature blocks, even if the corresponding root entry hash bitstream has not been programmed. PACSign allows you to prepend the required blocks with an empty signature chain.

2.5. Encryption

AFU Encryption is not supported on the Intel PAC with Intel Arria 10 GX FPGA.

3. Intel FPGA PAC Security Flow

The following steps describe the flow to enable Intel FPGA PAC security. See the corresponding sections in this chapter for detailed instructions on each step.

1. **Install PACSign.**
2. If you are in development, you may optionally create an unsigned AFU image to test and validate the functionality of your AFU image prior to fully signing the image for deployment into a production environment. Please refer to the *Creating Unsigned Images* section for more information.
3. **Create your root key and CSK(s). You can use OpenSSL or an HSM for this action.**

Figure 2. Key Creation Using OpenSSL

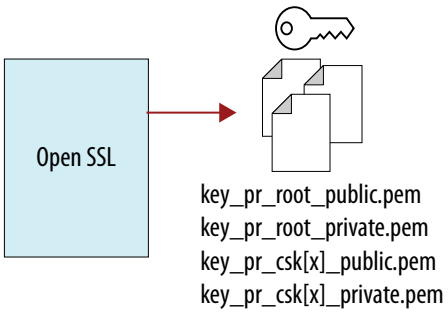
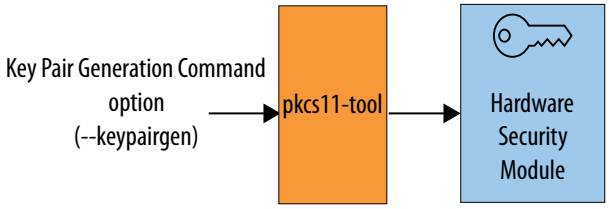


Figure 3. Key Creation Using HSM pkcs11_tool



4. **Create your root entry hash bitstream.**

Figure 4. Creating Root Entry Hash Bitstream with OpenSSL

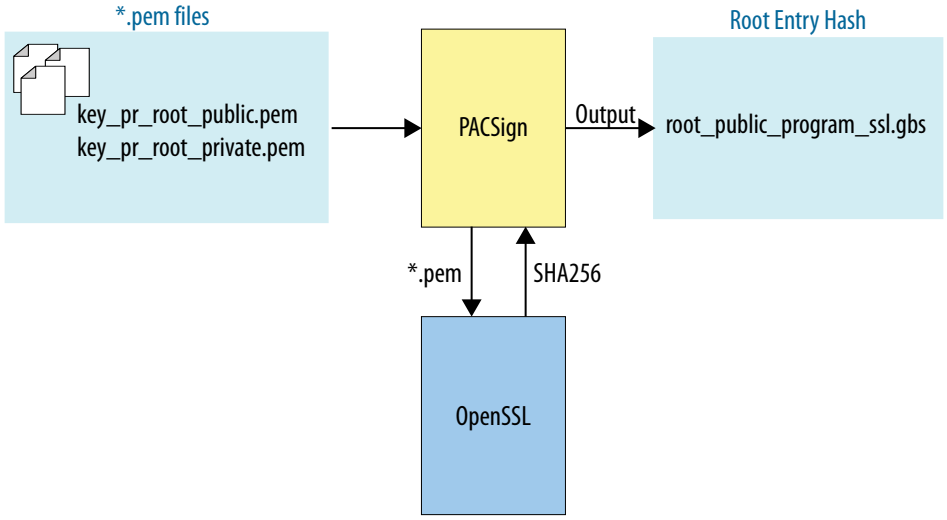
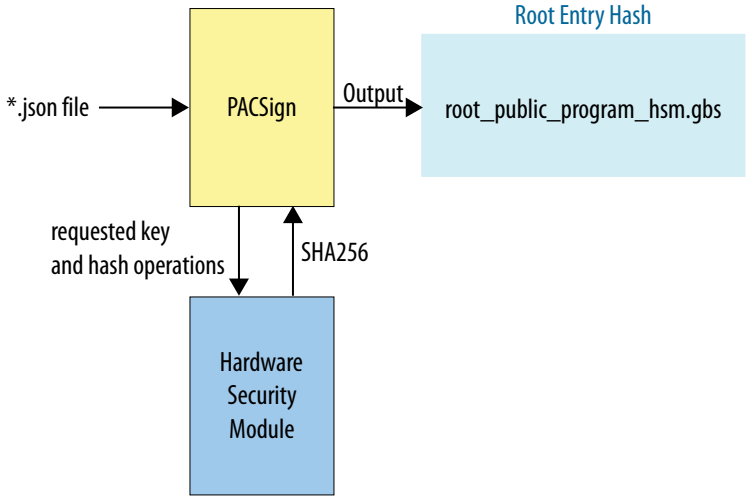


Figure 5. Creating Root Entry Hash Bitstream with HSM pkcs11_manager



5. **Program your root entry hash bitstream onto the Intel FPGA PAC.** You must power cycle the Intel FPGA PAC after you have programmed the root entry hash bitstream.
6. **Sign your AFU.**

Figure 6. Signing your image with OpenSSL

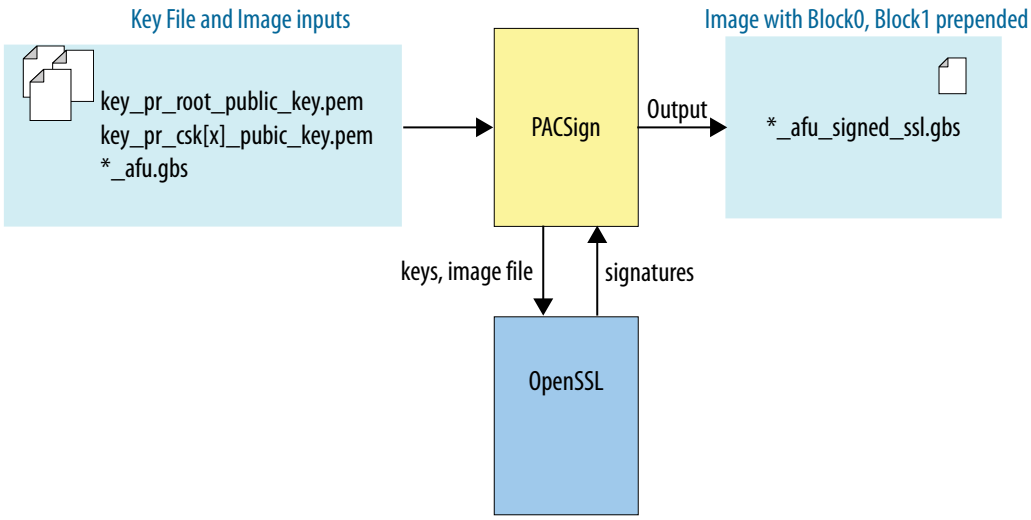
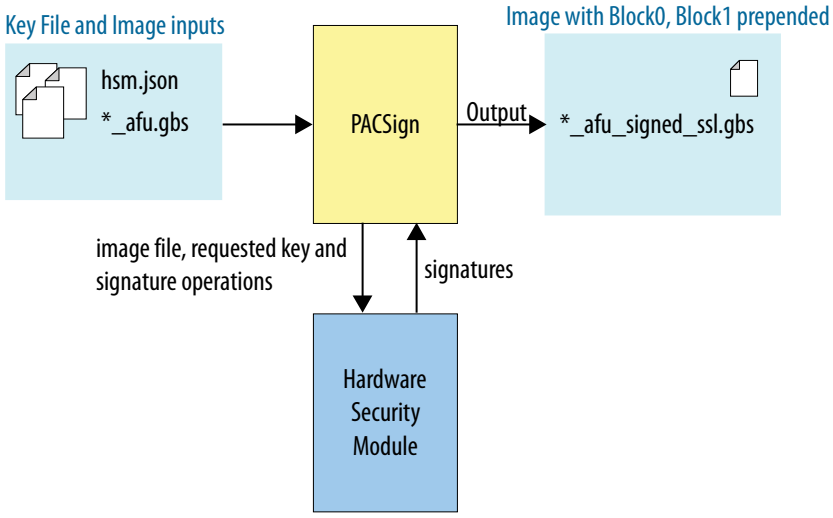


Figure 7. Signing your image with pkcs11_manager



7. **Program your AFU into the Intel FPGA PAC.** For directions on how to program your AFU, refer to the *Using fpgasupdate* chapter.

Related Information

- Creating Unsigned Images on page 17
- Using fpgasupdate on page 36

3.1. Installing PACSign

PACSign is a standalone tool that interfaces with your HSM to manage root entry hash bitstream creation, image signing, and cancellation bitstream creation. PACSign is implemented in Python and requires Python 3. Using PACSign with the PKCS11

interface requires the python-pkcs11 package. PACSign does not need an Intel FPGA PAC installed in the system to run. Systems where signed images are being deployed to an Intel FPGA PAC do not need PACSign installed nor access to the HSM.

Note: You must install Python 3 to use PACSign.

Note: The Acceleration Stack includes the PACSign package. You can check if you already have this package by typing: `rpm -qa| grep opae`.

1. Unpack the `opae.pac_sign-1.0.3.tar.gz` tarball, which contains the `opae.pac_sign-1.0.3-1.x86_64.rpm` package.

   ```
   sudo yum install opae.pac_sign-1.0.3-1.x86_64.rpm
   ```

2. Ensure you have installed Python 3, the Python 3 development libraries, and the Python 3 version of the python-pkcs11 package on your system.
3. Use your system package installer to install the .rpm package. PACSign installs to your `/usr/local/bin` directory and the necessary Python3.6 modules install to your `/usr/local/lib` directory.

   Note: PACSign depends on a Python3 interpreter version 3.6 or later. You must either install Python3 to, or create a symlink in, `/usr/local/bin` for PACSign to work. You must also ensure that the python modules PACSign depends on are visible to your python3 interpreter. You can do this by including the path `/usr/local/lib/python3.6/site-packages/` in the PYTHONPATH environment variable.

   ```
   export PYTHONPATH=/usr/local/lib/python3.6/site-packages/
   ```

3.2. PACSign Tool

The PACSign utility is installed on your path.

- Use PACSign by simply calling it directly with the command PACSign
- Calling PACSign with the `-h` option shows a help message describing the tool usage.
- Typing PACsign *<image_type>* `-h` shows options available for that image type.

```
[PACSign_Demo]$ PACSign -h
usage: PACSign [-h] {SR,FIM,BBS,BMC,BMC_FW,PR,AFU,GBS} ...

Sign PAC bitstreams

optional arguments:
-h, --help show this help message and exit

Commands:
Image types
{SR,FIM,BBS,BMC,BMC_FW,PR,AFU,GBS}
Allowable image types
SR (FIM, BBS)   Static FPGA image
BMC (BMC_FW)    BMC image
PR (AFU, GBS)   Reconfigurable FPGA image

[PACSign_Demo]$ PACSign AFU -h
usage: PACSign PR [-h] -t {UPDATE,CANCEL,RK_256,RK_384} -H HSM_MANAGER
                  [-C HSM_CONFIG] [-r ROOT_KEY] [-k CODE_SIGNING_KEY]
                  [-d CSK_ID] [-i INPUT_FILE] [-o OUTPUT_FILE] [-y] [-v]
```

```
optional arguments:
  -h, --help            show this help message and exit
  -t {UPDATE,CANCEL,RK_256,RK_384}, --cert_type {UPDATE,CANCEL,RK_256,RK_384}
                        Type of certificate to generate
  -H HSM_MANAGER, --HSM_manager HSM_MANAGER
                        Module name for key / signing manager
  -C HSM_CONFIG, --HSM_config HSM_CONFIG
                        Config file name for key / signing manager (optional)
  -r ROOT_KEY, --root_key ROOT_KEY
                        Identifier for the root key. Provided as-is to the key
                        manager
  -k CODE_SIGNING_KEY, --code_signing_key CODE_SIGNING_KEY
                        Identifier for the CSK. Provided as-is to the key
                        manager
  -d CSK_ID, --csk_id CSK_ID
                        CSK number. Only required for cancellation certificate
  -i INPUT_FILE, --input_file INPUT_FILE
                        File name for the image to be acted upon
  -o OUTPUT_FILE, --output_file OUTPUT_FILE
                        File name in which the result is to be stored
  -y, --yes             Answer all questions with "yes"
  -v, --verbose         Increase verbosity. Can be specified multiple times
```

3.3. Creating Unsigned Images

The TCM does not accept an AFU without the prepended authentication blocks generated by PACSign, even if an AFU root entry hash bitstream has not been programmed. If you want to operate an Intel FPGA PAC without a root entry hash bitstream programmed, such as in a development environment, you must still use PACSign to prepend the authentication blocks but you may do so with an empty signature chain. An image with prepended authentication blocks containing an empty signature chain is called an unsigned image. PACSign supports the creation of an unsigned image by using the UPDATE operation without specifying keys. Intel recommends using signed images in production deployments.

1. Create unsigned bitstream.

   Using OpenSSL:

   ```
   [PACSign_Demo]$ PACSign PR -t UPDATE -H openssl_manager -i hello_afu.gbs \
   -o hello_afu_unsigned_ssl.gbs
   ```

   Using HSM:

   ```
   [PACSign_Demo]$ PACSign PR -t UPDATE -H pkcs11_manager -C softhsm.json \
   -i hello_afu.gbs -o hello_afu_unsigned_hsm.gbs
   ```

   The output prompts you to enter Y or N to continue generating an unsigned bitstream.

   ```
   No root key specified. Generate unsigned bitstream? Y = yes, N = no: Y
   No CSK specified. Generate unsigned bitstream? Y = yes, N = no: Y
   ```

2. Program the unsigned bitstream.

   ```
   [PACSign_Demo]$ sudo fpgasupdate hello_afu_unsigned_ssl.gbs b2:00.0
   ```

   Note: If you attempt to program an AFU without the prepended authentication blocks, the TCM rejects the update and the Intel FPGA PAC requires a power cycle before the next programming attempt.

3.4. Using an HSM Manager

The PACSign tool does not implement any cryptographic functions. PACSign must interact with a cryptographic service, and it does this through modules called Hardware Security Module (HSM) managers. PACSign provides the following managers:

- openssl_manager: interfaces with OpenSSL
- pkcs11_manager: interfaces with any HSM implementing PKCS#11

Use the `-H` option with the `PACSign` command to select an HSM manager. The following sections provide examples for the PACSign OpenSSL manager using OpenSSL v1.1.1d, and the PACSign PKCS #11 manager using SoftHSM v2.5.0. Examples of key creation and management with both OpenSSL and SoftHSM (through the utilities softhsm2-util and pkcs11-tool) are also provided. To create your own custom HSM manager, refer to the *Custom HSM Manager Creation* topic more information.

Related Information

Creating a Custom HSM Manager on page 30

3.5. Creating Keys

Create your root and code signing keys using your desired key management utility (HSM or OpenSSL). Assign your key CSK IDs during key creation. Intel recommends that you consistently use the same ID for a given key across all image signings.

3.5.1. OpenSSL Key Creation

When using OpenSSL, create a private key and then create the corresponding public key. The PACSign OpenSSL manager requires specific tags in the key file names using a format: *key_<image_type>_<key_type>_<key_visibility>_key.pem*.

Table 4. PACSign OpenSSL Manager Key File Name Requirements

Filename Tag	Options	Description
image_type	• `pr` • `sr`	Identifies image type, partial reconfiguration or static region, for which the key is intended. • For Intel PAC with Intel Arria 10 GX FPGA, use; `key_pr_<key_type>_<key_section>_key.pem`
key_type	• `root` • `csk<x>`	Identifies key type. <x> specifies an ID that you use for cancellation. • Example: `key_pr_csk12_private_key.pem`
key_visibility	• `public` • `private`	Identifies the key visibility.

The following example creates a root key and two code signing keys using OpenSSL.

1. Create the root private key:

```
[PACSign_Demo]$ openssl ecparam -name secp256r1 -genkey -noout \
-out key_pr_root_private_key.pem
```

Output:

```
using curve name prime256v1 instead of secp256r1
```

2. Create the root public key:

```
[PACSign_Demo]$ openssl ec -in key_pr_root_private_key.pem -pubout \
-out key_pr_root_public_key.pem
```

Output:

```
read EC key
writing EC key
```

3. Create private CSK1:

```
[PACSign_Demo]$ openssl ecparam -name secp256r1 -genkey -noout \
-out key_pr_csk1_private_key.pem
```

Output:

```
using curve name prime256v1 instead of secp256r1
```

4. Create public CSK1:

```
[PACSign_Demo]$ openssl ec -in key_pr_csk1_private_key.pem -pubout \
-out key_pr_csk1_public_key.pem
```

Output:

```
read EC key
writing EC key
```

5. Create private CSK2:

```
[PACSign_Demo]$ openssl ecparam -name secp256r1 -genkey -noout \
-out key_pr_csk2_private_key.pem
```

Output:

```
using curve name prime256v1 instead of secp256r1
```

6. Create public CSK2:

```
[PACSign_Demo]$ openssl ec -in key_pr_csk2_private_key.pem -pubout \
-out key_pr_csk2_public_key.pem
```

Output:

```
read EC key
writing EC key
```

3.5.2. HSM Key Creation

If you are using an HSM, you need one token to create and store the root and code signing keys. The following example initializes a token using SoftHSM, with separate security officer and user PINs.

```
[PACSign_Demo]$ softhsm2-util --init-token --label pac-hsm --so-pin hsm-owner \
--pin pac-afu-signer --free
```

Output:

```
Slot 0 has a free/uninitialized token.
The token has been initialized and is reassigned to slot 1441483598
```

After you create a token, you can create keys in that token. The following example initializes a root and two code signing keys in the token created above, similarly using pkcs11-tool to interact with SoftHSM. The HSM, not PACSign, uses the key ID provided in this example. PACSign uses CSK IDs from a configuration `*.json` file in PKCS11 mode. You must manage consistency across ID values in the HSM and those used by PACSign. See the *PACSign PKCS11 Manager *.json Reference* topic for more information on the `*.json` file format.

1. Initialize the root key:

```
[PACSign_Demo]$ pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token-label pac-hsm --login --pin pac-afu-signer --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp256r1 \
--usage-sign --label root_key --id 0
```

   Output:

```
Key pair generated:
Private Key Object; EC
label: root_key
ID: 00
Usage: decrypt, sign, unwrap
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
0441043d3756347e6c257dac085574cc1cd984cdeee2c1059a0f035dabc3ad6e1950c8717dc7
ac8451a90c2471e95f4a69d6517f02f678830280f90a479c76a3e95d64
EC_PARAMS: 06082a8648ce3d030107
label: root_key
ID: 00
Usage: encrypt, verify, wrap
```

2. Initialize the CSK1:

```
[PACSign_Demo]$ pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token-label pac-hsm --login --pin pac-afu-signer --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp256r1 \
--usage-sign --label csk_1 --id 1
```

   Output:

```
Key pair generated:
Private Key Object; EC
label: csk_1
ID: 01
Usage: decrypt, sign, unwrap
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
0441041a827c903b5da8478c81fe652208704f0621b984190cd961ee154ac5c3ba772d1caa26
964a189262ee31b8e5d77898f293c0589b350103037b664d31adf68924
EC_PARAMS: 06082a8648ce3d030107
label: csk_1
ID: 01
Usage: encrypt, verify, wrap
```

3. Initialize CSK2:

```
[PACSign_Demo]$ pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token-label pac-hsm --login --pin pac-afu-signer --keypairgen \
--mechanism ECDSA-KEY-PAIR-GEN --key-type EC:secp256r1 \
--usage-sign --label csk_2 --id 2
```

   Output:

```
Key pair generated:
Private Key Object; EC
label: csk_2
ID: 02
```

```
Usage: decrypt, sign, unwrap
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
04410495f7556912d8753cf873be7a54e7d88c28bca672496abd90d9b44cc95cf50df9169b7a
d043a7340003a2bf96cb461e0575319b541ceb5d873d06334b30d208cc
EC_PARAMS: 06082a8648ce3d030107
label: csk_2
ID: 02
Usage: encrypt, verify, wrap
```

4. After keys are created in your token, it may be useful to inspect the token to verify the expected keys, labels, and IDs are present.

```
[PACSign_Demo]$ pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so \
--token-label pac-hsm --login --pin pac-afu-signer -O
```

Output:

```
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
04410495f7556912d8753cf873be7a54e7d88c28bca672496abd90d9b44cc95cf50df9169b7a
d043a7340003a2bf96cb461e0575319b541ceb5d873d06334b30d208cc
EC_PARAMS: 06082a8648ce3d030107
label: csk_2
ID: 02
Usage: encrypt, verify, wrap
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
0441043d3756347e6c257dac085574cc1cd984cdeee2c1059a0f035dabc3ad6e1950c8717dc7
ac8451a90c2471e95f4a69d6517f02f678830280f90a479c76a3e95d64
EC_PARAMS: 06082a8648ce3d030107
label: root_key
ID: 00
Usage: encrypt, verify, wrap
Private Key Object; EC
label: root_key
ID: 00
Usage: decrypt, sign, unwrap
Private Key Object; EC
label: csk_2
ID: 02
Usage: decrypt, sign, unwrap
Private Key Object; EC
label: csk_1
ID: 01
Usage: decrypt, sign, unwrap
Public Key Object; EC EC_POINT 256 bits
EC_POINT:
0441041a827c903b5da8478c81fe652208704f0621b984190cd961ee154ac5c3ba772d1caa26
964a189262ee31b8e5d77898f293c0589b350103037b664d31adf68924
EC_PARAMS: 06082a8648ce3d030107
label: csk_1
ID: 01
Usage: encrypt, verify, wrap
```

Related Information

PACSign PKCS11 Manager *.json Reference on page 29

3.6. Root Entry Hash Bitstream Creation

In order to program the root entry hash to an Intel FPGA PAC, you must use PACSign to create a root entry hash bitstream.

1. In your PACSign command, specify the type RK_256 and select the appropriate HSM manager and configuration.

- To create a root entry hash bitstream using OpenSSL and the key generated in the *OpenSSL Key Creation* topic, type:

```
[PACSign_Demo]$ PACSign AFU -t RK_256 -H openssl_manager \
-r key_pr_root_public_key.pem -o root_public_program_ssl.gbs
```

- To create a root entry hash bitstream using a SoftHSM and the root key generated in the *HSM Key Creation* topic, type:

```
[PACSign_Demo]$ PACSign AFU -t RK_256 -H pkcs11_manager \
-C softhsm.json -r root_key -o root_public_program_hsm.gbs
```

Note: PACSign requires an HSM configuration `*.json` file to request the correct key from the HSM. For more information about the structure and contents of the `*.json` file, refer to the *PACSign PKCS11 Manager .json Reference* topic.

2. After creating the root entry hash bitstream, program the bitstream to an Intel FPGA PAC using the `fpgasupdate` command as follows:

```
$ sudo fpgasupdate root_public_program_ssl.gbs 05:00.0
```

This operation is permanent and irreversible. After an AFU root entry hash bitstream is programmed, the Intel FPGA PAC validates an AFU signature prior to loading. For more details on key management, see the *Key Management* topic. For more information on how to use `fpgasupdate`, refer to the Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA.

Related Information

- PACSign PKCS11 Manager *.json Reference on page 29
- HSM Key Creation on page 19
- Key Management on page 10
- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA

3.7. Signing Images

After the root and code signing keys have been created, you may sign your AFU. Use the PR bitstream type with the UPDATE identifier to perform this operation, and specify the HSM configuration, root key, code signing key, and image input and output file names.

The following example demonstrates image signing using OpenSSL and the root and code signing keys generated in *OpenSSL Key Creation* topic.

```
[PACSign_Demo]$ PACSign PR -t UPDATE -H openssl_manager \
-r key_pr_root_public_key.pem -k key_pr_csk1_public_key.pem -i hello_afu.gbs \
-o hello_afu_signed_ssl.gbs
```

The following example demonstrates image signing using SoftHSM PKCS11 and the root and code signing keys generated in *HSM Key Creation* topic. Refer to the *PACSign PKCS11 Manager .json Reference* topic for more information on the `*.json` file used.

```
[PACSign_Demo]$ PACSign PR -t UPDATE -H pkcs11_manager -C softhsm.json \
-r root_key -k csk_1 -i hello_afu.gbs -o hello_afu_signed_hsm.gbs
```

You can program signed bitstreams on your Intel FPGA PAC by using the `fpgasupdate` tool and performing a remote system update. An Intel FPGA PAC only authenticates signed bitstreams after a root entry hash bitstream has been programmed. An Intel FPGA PAC that has not been programmed with a root entry hash bitstream accepts a signed bitstream and ignores the contents of the signature chain.

If you sign your image with a canceled CSK and attempt to program the Intel FPGA PAC, the BMC recognizes the bitstream as corrupted, reports an error and you must power cycle the Intel FPGA PAC to recover the card.

Related Information

- Root Entry Hash Bitstream Creation on page 21
- HSM Key Creation on page 19
- PACSign PKCS11 Manager *.json Reference on page 29

3.7.1. Creating OpenCL* Bitstreams

Creating signed or unsigned OpenCL* bitstreams requires some additional steps, because the AFU is embedded in the FPGA hardware configuration (`.aocx`) file, which is derived from an OpenCL compile.

The `sign_aocx.sh` script (distributed in `$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/`) creates the OpenCL bitstream for you. It performs the following steps automatically:

1. Extracts the AFU from the `.aocx` file
2. Signs the AFU (if desired) and applies security metadata.
3. Packs the AFU back into the `.aocx` file.

You can create unsigned bitstreams (with security metadata only) or signed `.aocx` file using the script. `sign_aocx.sh` calls PACSign to create the signature bitstreams.

To create the OpenCL bitstream, follow this workflow:

1. Decide which HSM manager to use: OpenSSL manager or PKCS11manager
2. Decide whether to create a signed or unsigned image
3. Source the `init_env.sh` script: Sourcing the init_env.sh Script on page 23
4. Generate the desired image: Creating the OpenCL Bitstream on page 24
5. Program the image to the board: Programming the Image File on page 28

3.7.1.1. Sourcing the `init_env.sh` Script

Source the `init_env.sh` script to initialize the environment for the Acceleration Stack and OpenCL.

```
source <DEV install path>/init_env.sh
```

After you have sourced the required environment with this command, you can use the `sign_aocx.sh` script to create a signed or unsigned bitstream.

Type the following command to see help documentation for the script:

```
$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -h
```

The command above produces the following help output:

```
The script assumes the PACsign and Intel Acceleration Stack environment is
setup. If not run the command : <stack_installation_path>/init_env.sh

******USAGE******:
1) For creating signed aocx run command :
./sign_aocx.sh [[[-H hsm_manager] [-i input_file ] [-r rootpublickey][-k
cskkey] [-o output_file]]]| [-h]]
2) For creating unsigned images run command :
./sign_aocx.sh [[[-H hsm_manager] [-i file ] [-r NULL] [-k NULL] [-o
output_file]
*****************
```

Command arguments:

- `-H` specifies the name of the HSM. Intel provides the `pkcs11_manager` and `openssl_manager` HSMs. You can also specify a custom HSM.
- `-i` specifies the input `.aocx` file or the path to the input `.aocx` file.
- `-r` specifies the root public key or the path to it.
- `-k` specifies the code signing key or the path to it.
- `-o` specifies the output filename you would like to create.
- `-h` displays the help text above.

If you would like to create an unsigned `.aocx` file, specify NULL as the root key (`-r`) and code signing key (`-k`) arguments.

You can run the script from any location by providing its path as shown above.

3.7.1.2. Creating the OpenCL Bitstream

Follow one of the four examples below, depending on which HSM you are using and whether the image is to be signed.

3.7.1.2.1. Example: Creating a Signed `.aocx` File Using OpenSSL Manager

Command syntax:

```
$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H openssl_manager \
-i <path_to_input_file/input_filename.aocx> -r <rootpublickey.pem> \
 -k <cskkey.pem> -o <path_to_output_file/output_filename.aocx>
```

Example output, signing `vector_add.aocx`:

```
$ $AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H openssl_manager \
-i vector_add.aocx -r ../openssl_keys_1_2_1/key_pr_root_public_key.pem \
-k ../openssl_keys_1_2_1/key_pr_csk2_public_key.pem -o signed_06_vector.aocx

The script assumes the PACsign and Intel Acceleration Stack environment is
setup. If not run the command : <stack_installation_path>/init_env.sh
hsm_manager=openssl_manager
aocx filename/path=vector_add.aocx
root_public_key=../openssl_keys_1_2_1/key_pr_root_public_key.pem
csk_public_key=../openssl_keys_1_2_1/key_pr_csk2_public_key.pem
output filename/path=signed_06_vector.aocx
```

```
openssl hsm_manager_options=openssl_manager
input path =.
input filename =vector_add.aocx
output path =.
output filename =signed_06_vector.aocx
Extracted the filename as signed_06_vector
1. Extracted the bin from the aocx
2. Extracted the gzip compressed GBS file from the .bin
3. Uncompressed .gz it to get the GBS file
Initiating PACSign tool to sign the GBS. This process will take a couple of
minutes...
Creating signed aocx file by signing the provided keys
2020-01-06 16:08:20,125 - PACSign.log - WARNING - Bitstream is already signed -
removing signature blocks
4. Signed the GBS
5. Compressed the gbs file
6. Added the signed gzip file to fpga.bin
7. Added the fpga.bin file back into aocx file
The signed file signed_06_vector.aocx has been generated. Use the command aocl
program <device_name> <filename>.aocx to program it on the FPGA card
```

The following message indicates that your output signed bitstream is successfully created:

```
The signed file <output_filename>.aocx has been generated. Use the command aocl
program <device_name> <filename>.aocx to program it on the FPGA card
```

3.7.1.2.2. Example: Creating an Unsigned .aocx File Using OpenSSL Manager

Command syntax:

```
$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H openssl_manager \
-i <path_to_input_file/input_filename.aocx> -r NULL -k NULL \
-o <path_to_output_file/output_filename.aocx>
```

Because no root key or code signing key is provided, the script asks if you would like to create unsigned bitstream, as shown below. Type Y to accept an unsigned bitstream.

```
No root key specified.  Generate unsigned bitstream? Y = yes, N = no: Y
No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: Y
```

Example output:

```
$ $AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H openssl_manager \
-i vector_add.aocx -r NULL -k NULL -o unsigned_vector_add.aocx

The script assumes the PACsign and Intel Acceleration Stack environment is
setup. If not run the command : <stack_installation_path>/init_env.sh
hsm_manager=openssl_manager
aocx filename/path=vector_add.aocx
root_public_key=NULL
csk_public_key=NULL
output filename/path=unsigned_vector_add.aocx
null=1
openssl hsm_manager_options=openssl_manager
input path =.
input filename =vector_add.aocx
output path =.
output filename =unsigned_vector_add.aocx
Extracted the filename as unsigned_vector_add
1. Extracted the bin from the aocx
2. Extracted the gzip compressed GBS file from the .bin
3. Uncompressed .gz it to get the GBS file
Initiating PACSign tool to sign the GBS. This process will take a couple of
minutes...
```

```
Creating unsigned aocx file by signing a NULL key
No root key specified.  Generate unsigned bitstream? Y = yes, N = no: Y
No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: Y
2020-01-13 17:57:17,052 - PACSign.log - WARNING - Bitstream is already signed -
removing signature blocks
4. Signed the GBS
5. Compressed the gbs file
6. Added the signed gzip file to fpga.bin
7. Added the fpga.bin file back into aocx file
The signed file unsigned_vector_add.aocx has been generated. Use the command
aocl program <device_name> <filename>.aocx to program it on the FPGA card
```

3.7.1.2.3. Example: Creating a Signed `.aocx` File Using PKCS11 Manager

Command syntax:

```
$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H pkcs11_manager \
-i <path_to_input_file/input_filename.aocx> -r <rootpublickey_name> \
-k <csk_name> -o <path_to_output_file/output_filename.aocx>
```

PKCS11 Manager gets the keys information from a `.json` file. If you follow the instructions in *HSM Key Creation*, your file is named `softhsm.json`.

Provide the `.json` file path and name when the script prompts you as follows:

```
For using pkcs11_manager please give the .json filename with the path:
```

Example output:

```
$ $AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H pkcs11_manager \
-i vector_add.aocx -r root_key -k csk_1 -o pkcs_vector.aocx

The script assumes the PACsign and Intel Acceleration Stack environment is
setup. If not run the command : <stack_installation_path>/init_env.sh
hsm_manager=pkcs11_manager
aocx filename/path=vector_add.aocx
root_public_key=root_key
csk_public_key=csk_1
output filename/path=pkcs_vector.aocx

For using pkcs11_manager please give the .json filename with the path:

<filepath>/softhsm.json

pkcs hsm_manager_options=pkcs11_manager -C softhsm.json
input path =.
input filename =vector_add.aocx
output path =.
output filename =pkcs_vector.aocx
Extracted the filename as pkcs_vector
1. Extracted the bin from the aocx
2. Extracted the gzip compressed GBS file from the .bin
3. Uncompressed .gz it to get the GBS file
Initiating PACSign tool to sign the GBS. This process will take a couple of
minutes...
Creating signed aocx file by signing the provided keys
2020-01-07 13:09:41,460 - PACSign.log - WARNING - Bitstream is already signed -
removing signature blocks
4. Signed the GBS
5. Compressed the gbs file
6. Added the signed gzip file to fpga.bin
7. Added the fpga.bin file back into aocx file
The signed file pkcs_vector.aocx has been generated. Use the command aocl
program <device_name> <filename>.aocx to program it on the FPGA card
```

Related Information

HSM Key Creation on page 19

3.7.1.2.4. Example: Creating an Unsigned .aocx File Using PKCS11 Manager

Command syntax:

```
$AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H pkcs11_manager \
-i <path_to_input_file/input_filename.aocx> -r NULL -k NULL \
-o <path_to_output_file/output_filename.aocx>
```

PKCS11 Manager gets the keys information from a `.json` file. If you follow the instructions in *HSM Key Creation*, your file is named `softhsm.json`.

Provide the `.json` file path and name when the script prompts you as follows:

```
For using pkcs11_manager please give the .json filename with the path:
```

Because no root key or code signing key is provided, the script asks if you would like to create unsigned bitstream, as shown below. Type Y to accept an unsigned bitstream.

```
No root key specified.  Generate unsigned bitstream? Y = yes, N = no: Y
No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: Y
```

Example output:

```
$  $AOCL_BOARD_PACKAGE_ROOT/linux64/libexec/sign_aocx.sh -H pkcs11_manager \
-i vector_add.aocx -r NULL -k NULL -o pkcs_vector.aocx
The script assumes the PACsign and Intel Acceleration Stack environment is
setup. If not run the command : <stack_installation_path>/init_env.sh
hsm_manager=pkcs11_manager
aocx filename/path=vector_add.aocx
root_public_key=NULL
csk_public_key=NULL
output filename/path=pkcs_vector.aocx
null=1

For using pkcs11_manager please give the .json filename with the path:

<filepath>/softhsm.json

pkcs hsm_manager_options=pkcs11_manager -C softhsm.json
input path =.
input filename =vector_add.aocx
output path =.
output filename =pkcs_vector.aocx
Extracted the filename as pkcs_vector
1. Extracted the bin from the aocx
2. Extracted the gzip compressed GBS file from the .bin
gzip: temp_pkcs_vector.gbs already exists; do you wish to overwrite (y or n)? y
3. Uncompressed .gz it to get the GBS file
Initiating PACSign tool to sign the GBS. This process will take a couple of
minutes...
Creating unsigned aocx file by signing a NULL key

No root key specified.  Generate unsigned bitstream? Y = yes, N = no: y
No CSK specified.  Generate unsigned bitstream? Y = yes, N = no: y

2020-01-07 15:59:16,726 - PACSign.log - WARNING - Bitstream is already signed -
removing signature blocks
4. Signed the GBS
gzip: signed_pkcs_vector.gbs.gz already exists; do you wish to overwrite (y or
n)? y
5. Compressed the gbs file
```

```
6. Added the signed gzip file to fpga.bin
7. Added the fpga.bin file back into aocx file
The signed file pkcs_vector.aocx has been generated. Use the command aocl
program <device_name> <filename>.aocx to program it on the FPGA card
```

Related Information

HSM Key Creation on page 19

3.7.1.3. Programming the Image File

After you have generated your signed or unsigned `.aocx` file, program it to the board by using the following command.

```
aocl program <device_name> <filename>.aocx
```

Note: You can find the device name by running aocl diagnose command.

For more information refer to the *Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA Quick Start User Guide* or the *OpenCL on Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA Quick Start User Guide*.

Related Information

- Intel Acceleration Stack Quick Start Guide for Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA
- OpenCL on Intel Programmable Acceleration Card with Intel Arria 10 GX FPGA Quick Start User Guide

3.8. Creating a CSK ID Cancellation Bitstream

To cancel a CSK ID on an Intel FPGA PAC, you must use PACSign to create a CSK ID cancellation bitstream. To do this, you must specify the type CANCEL, select the appropriate HSM manager and root key, and provide the key ID number to cancel. For OpenSSL, the key ID used during image signing is derived from the CSK filename. For PKCS11, the key ID used during image signing is extracted from the configuration .json.

1. Create a cancellation bitstream.

   Using OpenSSL:

   ```
   [PACSign_Demo]$ PACSign AFU -t CANCEL -H openssl_manager \
   -r key_pr_root_public_key.pem -d 1 -o ssl_csk1_cancel.gbs
   ```

   Using PKCS11:

   ```
   [PACSign_Demo]$ PACSign AFU -t CANCEL -H pkcs11_manager -C softhsm.json \
   -r root_key -d 1 -o hsm_csk1_cancel.gbs
   ```

2. Program the CSK ID cancellation on the Intel FPGA PAC using the `fpgasupdate` tool.

   ```
   $ sudo fpgasupdate ssl_csk1_cancel.gbs b2:00.0
   ```

   CSK ID cancellation bitstreams are only valid on Intel FPGA PACs that have been programmed with the corresponding root entry hash bitstream. After you program a CSK ID cancellation bitstream, you must power cycle the Intel FPGA PAC.

3.9. PACSign PKCS11 Manager *.json Reference

The PACSign PKCS11 Manager uses a `*.json` file that stores information on how to interact with your HSM.

It contains information specific to your HSM, as well as a description of the token and keys that you created for use with PACSign. The PKCS11 examples in this chapter use `softhsm.json`, which contains the following:

```
{
    "cryptoki_version": [2, 40],
    "library_version": [2, 5],
    "platform-name" : "DCP",
    "lib_path" : “/usr/local/lib/softhsm/libsofthsm2.so”,
    "curve": "secp256r1",
    "token": {
        "label": "pac-hsm",
        "user_password": "pac-afu-signer",
        "keys":
            [
                {
                    "label": "root_key",
                    "key_id": "0",
                    "type": "PR",
                    "permissions": "0xFFFFFFFF",
                    "csk_id": "0xFFFFFFFF",
                    "is_root": true
                },
                {
                    "label": "csk_1",
                    "key_id": "1",
                    "type": "PR",
                    "permissions": "0x4",
                    "csk_id": "0x1",
                    "is_root": false
                },
                {
                    "label": "csk_2",
                    "key_id": "2",
                    "type": "PR",
                    "permissions": "0x4",
                    "csk_id": "0x2",
                    "is_root": false
                }
            ]
        }
}
```

The `cryptoki_version` and `library_version` information is determined by your HSM and can be reported by pkcs11-tool:

```
[PACSign_Demo]$ pkcs11-tool --module=/usr/local/lib/softhsm/libsofthsm2.so -I
```

Output:

```
Cryptoki version 2.40
Manufacturer SoftHSM
Library Implementation of PKCS11 (ver 2.5)
Using slot 0 with a present token (0x55eb4b4e)
```

- `platform-name`: Always set to DCP.
- `lib_path`: Your HSM software library installation determines this path.
- `curve`: Always set to `secp256r1` because this is the only elliptic curve currently supported by the BMC.
- The `token` entry contains:
  - `label`: determined when you initialize the token in your HSM
  - `user_password`: determined when you initialize the token in your HSM
  - `keys`: lists the keys in the token available for use by PACSign
- Within the `key` field are:
  - `label`: determined when you initialize the token in your HSM
  - `key_id`: determined when you initialize the token in your HSM

    Note: Each `label` and `key_id` must match what you used when you created the key.
  - `type`: Either PR or SR for partial reconfiguration or static region, respectively.
  - `permissions`: Set to 0x1 for static region signing; 0x2 for BMC signing; 0x4 for partial reconfiguration region signing.
  - `csk_id`: What PACSign uses when signing an AFU; does not need to match the `key_id` field. Valid values are `0xFFFFFFFF` for root keys and `0x0-0x7F` for Intel FPGA PAC D5005 code signing keys.
  - `is_root`: Allows you to designate to PACSign the intended use of the key as a root key or code signing key.

3.10. Creating a Custom HSM Manager

PACSign is a Python tool that uses a plugin architecture for the HSM interface. PACSign is distributed with managers for both OpenSSL and PKCS #11. This section describes the functionality required by PACSign from the HSM interface and shows how to construct a plugin.

The distribution of PACSign uses the following directory structure:

```
hsm_managers
├── openssl_manager
│   └── library
├── pkcs11_manager
source
```

The top level contains `PACSign.py` with the generic signing code in source. The HSM managers reside each in their own subdirectory under `hsm_managers` as packages. The directory name is what is given to PACSign's `--HSM_MANAGER` command-line option. If the specific manager requires additional information, you can provide it using the optional `--HSM_config` command-line option. For example, the PKCS #11 plugin requires a `*.json` file describing the tokens and keys available on the HSM.

You must place each plugin that is to be supported in a subdirectory of the `hsm_managers` directory. Use a descriptive name for the directory that clearly describes the supported HSM. This subdirectory may have an `__init__.py` file whose contents import the modules needed by the plugin. The names of the plugin modules are not important to the proper functioning of PACSign.

The newly-created plugin must be able to export one attribute named HSM_MANAGER that is invoked by PACSign with an optional configuration file name provided on the command-line. Invocation of HSM_MANAGER(`config_file`) returns a class with certain methods exposed, which are described in later sections.

Current implementations of HSM_MANAGER define it as a Python class object. The initialization function of the class reads and parses the configuration file (if present) and performs HSM initialization. For the PKCS #11 implementation, the class looks like this:

```
class HSM_MANAGER(object):
    def __init__(self, cfg_file = None):
        common_util.assert_in_error(cfg_file, \
            PKCS11 HSM manager requires a configuration file")
        self.session = None
        with open(cfg_file, "r") as read_file:
            self.j_data = json.load(read_file)
        j_data = self.j_data

        lib = pkcs11.lib(j_data['lib_path'])
        token = lib.get_token(token_label=j_data['token']['label'])
        self.session = token.open(user_pin=j_data['token']['user_password'])
        self.curve = j_data['curve']

        self.ecparams = self.session.create_domain_parameters( \
        pkcs11.KeyType.EC, {pkcs11.Attribute: \
        pkcs11.util.ec.encode_named_curve_parameters(self.curve)}, \
        local=True)
```

Error handling code has been omitted for clarity. This code does the following:

- Opens and parses the `*.json` configuration file.
- Loads the vendor-supplied PKCS #11 library into the program.
- Sets up a session with the correct token.
- Retrieves the proper elliptic curve parameters for the curve you select.

The following sections describe the required exported methods of this class.

3.10.1. HSM_MANAGER.get_public_key(public_key)

This method returns an instance of a public key that is described by 'public_key', which was provided via a command-line option (`--root_key` or `--code_signing_key`). The HSM manager must know how to properly identify the key on the HSM given this string.

The public key instance is required to supply the public methods described in the sections that follow. The PKCS #11 implementation of this function, `get_public_key`, is below:

```
def get_public_key(self, public_key):
    try:
        key_, local_key = self.get_key(public_key, ObjectClass.PUBLIC_KEY)
        key_ = key_[Attribute.EC_POINT]
```

```
    except pkcs11.NoSuchKey:
        pass # No key found
    except pkcs11.MultipleObjectsReturned:
        pass # Multiple keys found
    return _PUBLIC_KEY(key_[3:], local_key)
```

3.10.1.1. PUBLIC_KEY.get_X_Y()

This function returns a `common_util.BYTE_ARRAY()` that contains the elliptic curve point associated with the key. The returned value should be X concatenated with Y, each with the proper number of bytes. For our implementation, each of X and Y are 32 bytes (256 bits) because secp256r1 curve parameters are required.

3.10.1.2. PUBLIC_KEY.get_permission()

Intel FPGA PAC keys have associated permissions. This function returns an integer that corresponds to the assigned key permissions. For Intel FPGA PACs, all root key permissions must be the constant `0xFFFFFFFF`. For code signing keys, the permissions are described below.

Table 5. Key Permissions

Value	Name	Permission
1	SIGN_SR	Sign the FIM or Static Region
4	SIGN_PR	Sign the PR Region or AFU

3.10.1.3. PUBLIC_KEY.get_ID()

Intel FPGA PACs have a laddering key mechanism that allows for cancellation of code signing keys. This method returns the integer key ID of the specified key. The root key ID must be the constant `0xFFFFFFFF`. Root keys cannot be canceled.

Intel PAC with Intel Arria 10 GX FPGA AFU code signing key IDs must be in the range 0 to 127 (7-bit unsigned).

3.10.1.4. PUBLIC_KEY.get_content_type()

Code signing keys and root keys can be restricted to signing only certain types of content. For instance, there are separate root keys for PR, SR, and BMC bitstreams as well as corresponding code signing keys. This method should return the bitstream type associated with this key, and must be one of `{FIM, SR, BBS, BMC, BMC_FW, AFU, PR, or GBS}`.

3.10.2. HSM_MANAGER.sign(data, key)

This method uses the key provided to generate an ECDSA signature over the provided data.

The return value of this method is a `common_util.BYTE_ARRAY()` containing the R and S values of the signature concatenated. PACSign only signs hashes, so the length of the data to be signed will be a fixed-length 32 byte array.

3.10.3. Signing Operation Flow

A PACSign command that invokes the PKCS #11 manager plugin initializes it with the configuration file name.

PACSign performs insertion of authentication blocks into the bitstream, signed by the root and code signing keys. The resultant signed bitstream is written to the specified output file.

PACSign requests that the HSM manager retrieve the public key X and Y values for the root key and the code signing key. The HSM manager returns the R and S signature over PACSign-provided 256-bit hash values using the root key and code signing key. The following code snippet demonstrates how PACSign utilizes the HSM manager.

```
self.pub_root_key_c = self.hsm_manager.get_public_key(args.root_key)
common_util.assert_in_error(self.pub_root_key_c, \
    "Cannot retrieve root public key")
        self.pub_root_key = self.pub_root_key_c.get_X_Y()
        self.pub_root_key_perm = self.pub_root_key_c.get_permission()
        self.pub_root_key_id = self.pub_root_key_c.get_ID()
        self.pub_root_key_type = self.pub_root_key_c.get_content_type()

self.pub_CSK_c = self.hsm_manager.get_public_key(args.code_signing_key)
common_util.assert_in_error(self.pub_CSK_c != None, \
    "Cannot retrieve public CSK")
        self.pub_CSK = self.pub_CSK_c.get_X_Y()
        self.pub_CSK_perm = self.pub_CSK_c.get_permission()
        self.pub_CSK_id = self.pub_CSK_c.get_ID()
        self.pub_CSK_type = self.pub_CSK_c.get_content_type()

sha = sha256(block0.data).digest()
rs = self.hsm_manager.sign(sha, args.code_signing_key)
sha = sha256(csk_body.data).digest()
rs = self.hsm_manager.sign(sha, args.root_key)
```

3.11. PACSign Man Page

PACSign man page is reproduced here for convenience.

```
SYNOPSIS
python PACSign.py [-h] {FIM,SR,BBS,BMC,BMC_FW,AFU,PR,GBS} ...
python PACSign.py <CMD> [-h] -t {UPDATE,CANCEL,RK_256,RK_384} -H HSM_MANAGER [-
C HSM_CONFIG] [-r ROOT_KEY] [-k CODE_SIGNING_KEY] [-d CSK_ID] [-i INPUT_FILE] [-
o OUTPUT_FILE] [-y] [-v]

DESCRIPTION
PACSign is a utility designed to insert proper authentication markers on
bitstreams targeted for the PACs. To accomplish this, it uses a root key and an
optional code signing key to digitally sign the bitstreams to validate their
origin. The PACs will not accept loading bitstreams without proper
authentication.
The current PACs only support elliptical curve keys with the curve type
secp256r1 or prime256v1. PACSign is distributed with managers for both OpenSSL
and PKCS #11.

BITSTREAM TYPES
The first required argument to PACSign is the bitstream type identifier.

{SR,FIM,BBS,BMC,BMC_FW,PR,AFU,GBS}

Allowable image types. FIM and BBS are aliases for SR, BMC_FW is an alias for
BMC, and AFU and GBS are aliases for PR.

SR (FIM, BBS)
```

```
Static FPGA image

BMC(BMC_FW)

BMC image, including firmware for some PACs

PR (AFU, GBS)

Reconfigurable FPGA image

REQUIRED OPTIONS
All bitstream types are required to include an action to be performed by
PACSign and the name and optional parameter file for a key signing module.

-t, --cert_type <type>

Values must be one of UPDATE, CANCEL, RK_256, or RK_384[^1].
`UPDATE` - add authentication data to the bitstream.
`CANCEL` - create a code signing key cancellation bitstream.
`RK_256` - create a bitstream to program a 256-bit root key to the device.
`RK_384` - create a bitstream to program a 384-bit root key to the device.
[^1]:Current PACs do not support 384-bit root keys.

-H, --HSM_manager <module>

The module name for a manager that is used to interface to an HSM. PACSign
supplies both openssl_manager and pkcs11_manager to handle keys and signing
operations.

-C, --HSM_config <cfg> (optional)

The argument to this option is passed verbatim to the specified HSM manager.
For pkcs11_manager, this option specifies a JSON file describing the PKCS #11
capable HSM's parameters.

OPTIONS
-r, --root_key <keyID>

The key identifier recognizable to the HSM manager that identifies the root key
to be used for the selected operation.

-k, --code_signing_key <keyID>

The key identifier recognizable to the HSM manager that identifies the code
signing key to be used for the selected operation.

-d, --csk_id <csk_num>

Only used for type CANCEL and is the key number of the code signing key to
cancel.

-i, --input_file <file>

Only used for UPDATE operations. Specifies the file name containing the data to
be signed.

-o, --output_file <file>

Specifies the name of the file to which the signed bitstream is to be written.

-y, --yes

Silently answer all queries from PACSign in the affirmative.

-v, --verbose

Can be specified multiple times. Increases the verbosity of PACSign. Once
enables non-fatal warnings to be displayed; twice enables progress information.
Three or more occurrences enables very verbose debugging information.
```

```
NOTES
Different certification types require different sets of options. The table
below describes which options are required based on certification type:

UPDATE
    root_key     code_signing_key     csk_id     input_file     output_file
SR   Optional[^2]  Optional[^2]           No          Yes           Yes
BMC  Optional[^2]  Optional[^2]           No          Yes            Yes
PR   Optional[^2]  Optional[^2]           No          Yes            Yes

CANCEL
    root_key     code_signing_key     csk_id     input_file     output_file
SR     Yes         No               Yes         No           Yes
BMC    Yes         No               Yes         No           Yes
PR     Yes         No               Yes         No           Yes

RK_256 / RK_384[^1]
    root_key     code_signing_key     csk_id     input_file     output_file
SR     Yes         No               No          No           Yes
BMC    Yes         No               No          No           Yes
PR     Yes         No               No          No           Yes
[^2]: For UPDATE type, both keys must be specified to produce an authenticated
bitstream. Omitting one key generates a valid, but unauthenticated bitstream
that can only be loaded on a PAC with no root key programmed for that type.

EXAMPLES
The following command will generate a root hash programming PR bitstream. The
generated file can be given to fpgasupdate to program the root hash for PR
operations into the device flash. Note that root hash programming can only be
done once on a PAC.

python PACSign.py PR -t RK_256 -o pr_rhp.bin -H openssl_manager -r
key_pr_root_public_256.pem

The following command will add authentication blocks to hello_afu.gbs signed by
both provided keys and write the result to s_hello_afu.gbs. If the input
bitstream were already signed, the old signature block is replaced with the
newly-generated block.

python PACSign.py PR -t update -H openssl_manager -i hello_afu.gbs -o
s_hello_afu.gbs -r key_pr_root_public_256.pem -k key_pr_csk0_public_256.pem

The following command will generate a code signing key cancellation bitstream
to cancel code signing key 4 for all BMC operations. CSK 4 bitstreams that
attempt to load BMC images will be rejected by the PAC.

python PACSign.py BMC -t cancel -H openssl_manager -o csk4_cancel.gbs -r
key_bmc_root_public_256.pem -d 4
```

4. Using fpgasupdate

Use the `fpgasupdate` command to securely update the following files in flash:

- BMC firmware
- FIM images
- AFU (partial reconfiguration) images

When you call `fpgasupdate` the TCM orchestrates the update.

- The TCM rejects an update request if another update is currently in progress. The TCM monitors flash write and update counts and delays an update 30 seconds if more than 1,000 updates have occurred, and 60 seconds if more than 2,000 updates have occurred.
- The TCM stops the currently running AFU and loads the BIP from on-board flash.
- The TCM grants access only to a staging area in the on-board DDR memory, and only for enough time for the host to write an update into the staging area.

  Note: Overwriting memory contents is harmless at this point, because the previous AFU is no longer present and the BIP has full control. The next AFU to be loaded does not make assumptions about the contents of memory.
- The TCM then restricts all write access to ensure the update image cannot be changed during or after the authentication process.
- If authentication is successful, the TCM copies the image from the staging area into the appropriate interface: the BMC flash for BMC updates, the on-board flash for FIM or AFU updates, or directly to the PR interface for an immediate execution of the new AFU.

To use the command type:

```
$ sudo fpgasupdate [--log-level=<level>] file [bdf]
```

where the following options are as follows:

Table 6. fpgasupdate Options

Parameters	Options	Notes
level	state, ioctl, debug, info, warning, error, critical. Default value is state.	
file	The secure update file that you program in the Intel FPGA PAC	
[bdf]	`[ssss:]bb:dd:f`, corresponding to PCIe segment, bus, device, function. The segment is optional; if omitted, a segment of `0000` is assumed.	If there is only one Intel FPGA PAC in the system, then `bdf` may be omitted. In this case, `fpgasupdate` determines the address automatically.

4.1. Troubleshooting

`fpgasupdate` provides descriptive errors when it cannot complete the requested operation.

When using `fpgasupdate` to program bitstreams created or signed with PACSign, the tool may reject the bitstream if, for example, there was an error in the signing process or if the signed bitstream is corrupted.

The TCM provides extended error codes to assist in troubleshooting an authentication failure. Error codes are logged in your system messaging log, such as `dmesg` or `/var/syslogs`. You may use the following table to decode the authentication status and associated errors.

Table 7. Authentication Status Register Values and Error Descriptions

Authentication Status Value	Error Name	Error Description	Corrective Action
32'h00000000	Block0 Magic value error	Bitstream Format Error: Block 0 bad magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000001	Block0 ConLen error	Bitstream Format Error: Block 0 content length error. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000002	Block0 ConType B[7:0] > 2	Bitstream Format Error: Block 0 content type error. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000003	Root Entry Magic Number error	Bitstream Format Error: Root entry bad magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000004	Root Entry Curve Magic value error	Bitstream Format Error: Root entry bad magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000005	Root Entry Permission error	Root entry bad permissions. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000006	Root Entry Key ID error	Bitstream Format Error: Root entry bad key ID. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000007	Root Entry hash mismatch	Bitstream Format Error: Root entry does not match root entry stored on card.	Ensure bitstream is properly signed with the correct keys.
32'h00000008	CSK Entry Magic value error	Bitstream Format Error: CSK bad magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h00000009	CSK Entry Curve Magic value error	Bitstream Format Error: CSK bad curve magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.

continued...

Authentication Status Value	Error Name	Error Description	Corrective Action
32'h0000000A	CSK Key Canceled	Authentication Error: CSK canceled. Indicates you are attempting to program an image with a canceled CSK.	Ensure bitstream is properly signed with the correct keys.
32'h0000000B	CSK Entry Permission error	Authentication Error: CSK bad permission. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h0000000C	CSK Entry verify ECDSA and SHA failed	Authentication Error: CSK signature invalid. Indicates CSK or root entry hash tampering.	Ensure bitstream is properly signed with the correct keys.
32'h0000000D	Block0 Entry Magic value error	Bitstream Format Error: Block 0 entry bad magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h0000000E	Block 0 Entry Curve Magic value error	Bitstream format error: Block 0 entry bad curve magic number. Indicates bitstream corruption.	Ensure bitstream is properly signed with the correct keys.
32'h0000000F	Block0 Entry verify ECDSA and SHA failed	Authentication Error: Block 0 entry signature invalid. May indicate image tampering.	Ensure bitstream is properly signed with the correct keys.
32'h00000010	Block1 Entry Magic Value Error	Bitstream Format Error: Block 1 entry bad magic number. Indicates bitstream corruption.	Program root entry hash bitstream.
32'h00000011	BIP PR error	TCM unable to load BIP.	Contact Intel support.
32'h00000012	BIP Block 0/1 error	Bitstream format error: BIP does not recognize Block0/ Block1 entry	Ensure bitstream is properly signed with the correct keys.
32'h00000013	BIP Start error	TCM unable to load and start BIP.	Contact Intel support.
32'h00000014	Host bitstream download timeout	Host loading of bitstream did not complete in time expected by TCM.	Retry update operation.
32'h00000015	Host canceled update	Update operation canceled by host.	Retry update operation.
32'h00000016	Root Entry Hash bitstream not programmed for RSU and Cancellation	You attempted to program a key cancellation bitstream without first programming a root entry hash bitstream.	Program a root entry hash before attempting to program a key cancellation bitstream.
8'h00000017	KEY hash has been programmed for KEY hash programming certificate	Authentication Error: Attempt to program root entry hash when the root entry hash bitstream has already been programmed.	You may only program root entry hash bitstream one time.
32'h00000018	Payload SHA Invalid	Authentication Error: Payload SHA mismatch. May indicate tampering of the root key.	Verify correctness of bitstream; may need to resign.
32'h00000019	Host BMC command blacklisted	TCM filtered a disallowed command from the host to the Intel FPGA PAC BMC.	Do not issue the failing command to the BMC.

continued...

Authentication Status Value	Error Name	Error Description	Corrective Action
32'h0000001A	Host response timeout on BMC command	The host did not respond in time after a BMC command.	Retry BMC command operation.
32'h0000001B	BMC timeout on request	The BMC did not respond in time to acknowledge a command.	Retry command.
32'h0000001C	BMC timeout sending response	The BMC did not respond in time after acknowledging a command.	Power-cycle Intel FPGA PAC and retry command.
32'h0000001d	Host BMC response timeout on response acknowledge	The Host BMC did not respond in time to the Intel FPGA PAC BMC.	Retry command.
32'h0000001e	Flash open error	The TCM experienced an error accessing on-board flash.	Contact Intel support.
32'h0000001f	Flash read error	The TCM experienced an error accessing on-board flash.	Contact Intel support.
32'h00000020	BIP load error	The BIP encountered an error during loading.	Retry update operation.
32'h00000021	PR reset timeout	The TCM encountered a timeout when attempting to reset the PR region.	Power cycle the PAC.
32'h00000022	BMC update prohibited after PR	The TCM blocks BMC updates after an AFU has been loaded.	Power cycle the PAC and attempt BMC update before loading an AFU.
32'h00000023	Flash update prohibited after PR	The TCM blocks a flash update after an AFU has been loaded.	Power cycle the PAC and attempt update before loading an AFU.
32'h00000024	BMC update delayed	The TCM refused the BMC update operation due to a high update count and a short time since the last update operation.	Retry update operation at a later time (up to 120s).
32'h00000025	Flash update delayed	The TCM refused the update operation due to a high update count and a short time since the last update operation.	Retry update operation at a later time (up to 120s).
32'h00000026	Bad BMC command length	The TCM filtered a command to the BMC due to an improper command length.	Format the BMC command correctly.
32'h00000027	Bad BMC response length	The TCM filtered a BMC response due to an improper length.	Format the BMC response correctly.
32'h00000028	PR prohibited from factory image	The TCM reverted to factory image and an attempt to load an AFU was made.	Update the Intel FPGA FIM image.
32'h00000029	CSK bad CSK ID	Authentication Error: CSK ID is an invalid value.	Verify correctness of bitstream; may need to resign.

continued...

Authentication Status Value	Error Name	Error Description	Corrective Action
32'hDD00xxxx	BIP internal error code	The BIP experienced an error.	Contact Intel support.
32'hEE00xxxx	PR IP internal error code	The AFU experienced an error.	Contact AFU vendor support.
32'hFFFFFFFF	No Error	No Error	

5. Document Revision History

Document Version	Changes
2020.03.06	Initial production release.