Code-Signing Tool

User's Guide

Rev. 3.4.0
12/2023

Contents

About This Book

This manual, the *Code-Signing Tool User's Guide*, provides the details necessary to install, configure, and run the code-signing tool (CST).

Audience

This document provides installation instructions and describes the use of the code signing tools for administrators and engineers performing codes signing for the NXP High Assurance Boot (HAB) and Advanced High Assurance Boot (AHAB) feature.

Scope

This document focuses on the use of the CST to generate keys, certificates, HAB4/AHAB SRK tables, SRK hash values and generating data which include digital signatures. The use of the NXP Manufacturing tool to load images and to burn e-fuses are beyond the scope of this document.

Organization

The remainder of this manual is divided into sections according to the main HAB Code Signing Tool user tasks:

- Section 1, "Introduction," describes the background of the code-signing tool and the goals of the procedures in later sections.
- Section 2, "Installation" describes the steps to install the Code-Signing Tool (CST) program files.
- Section 3, "Key and Certificate Generation" details the steps to generate signing keys and certificates for the HAB Version 4 and AHAB.
- Section 4, "CST Usage" describes how to use the CST client command line interface.
- Section 5, "CSF Description Language" provides CST description language details required to create a CSF description file.

One appendix also included:

- Appendix A, "CST Architecture" presents details about CST implementation and customization options.

Revision History

Version	Date	Change Description
1.0	11/15/2011	- Initial Version
2.0	11/09/2012	- Bug fixes and other updates
2.1	4/15/2013	- Add Support for HAB4 fast authentication
2.2	10/14//2014	- Add note on Linux RNG dependency - Add Appendix B containing details on replacing the CST Back End - Corrected CA flag documentation
2.3	3/30/2015	- Bug fixes related to encrypted images
2.3.1	7/1/2015	- Fix for 64-bit version of srktable
2.3.2	3/15/2016	- Added support for manufacturing protection - Changed input from STDIN to command line argument - Made RNG unlock automatic only for CAAM
2.3.3	11/14/2017	- Added support for MS Windows - Removed support for several commands: Write Data Clear Mask Set Mask Check Clear/Set Set MID
3.0.0	04/04/2018	- Added support for AHAB
3.0.1	05/11/2018	- Bug fixes related to Windows support
3.1.0	08/2018	- Added OpenSSL 1.1.0 support - Added ECDSA support for HAB4 (Only available in HAB 4.5.0) - Fixed encrypted boot support - Added convlb.exe to work-around line break limitation in Windows - Added HSM backend - Added HAB4 log parser tool - Added HAB4 CSF parser tool - Added HAB4 SRKTool script
3.2.0	04/2019	- Removed HAB3 support - Added encrypted boot support for AHAB - Added CST source code

3.3.0	12/2019	- Added support for MacOS - Added AHAB signature block parser tool
3.3.1	07/2020	- CST tool binaries built using OpenSSL 1.1.1 - CST tool binaries support Encrypted Boot by default - Added HAB4 log parser tool for Windows and MacOS
3.3.2	04/2023	- Added new backend supporting HAB and AHAB signing using production keys stored on Hardware Token - Removed HSM backend - Updated srktool to support sha256 fuse array
3.4.0	12/2023	- Transitioned to OpenSSL 3, following OpenSSL 1.1.1 End Of Life in Sept 2023. No more public security fixes for 1.1.1 post that date Compiled with OpenSSL 3.2.0 for enhanced security and features - Simplified the PKCS#11 backend, eliminating redundant code - Introduced Dockerfile.hsm for experimenting with the PKCS#11 backend. - PKCS#11 generation scripts now located in the 'keys' folder - Removed repetitive PIN requests in PKI generation scripts - The backend now supports RSA PSS - Improved build system for efficiency and ease of use - Updated Dockerfile; relocated to top folder alongside Makefile for easier access and build management - Added support for 32bit Linux in the hab_log_parser tool - Removed OSX binaries; however, the build system and sources still support it - Added BUILD.md with instructions for CST building using Docker - Code-Signing Tool User's Guide updated with instructions on using the PKCS#11 backend and notes about i.MX9x devices

Conventions

Use this section to name, describe, and define any conventions used in the book. This document uses the following notational conventions:

- `Courier monospaced type` indicates commands, command parameters, code examples, expressions, datatypes, and directives.

- *Italic type* indicates replaceable command parameters.
- All source code examples are in C.

Definitions, Acronyms, and Abbreviations

The following list defines the acronyms and abbreviations used in this document.

AES	Advanced Encryption Standard
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
CA	Certificate Authority
CCM	Counter with CBC-MAC
CSF	Command Sequence File
CMS	Cryptographic Message Syntax
CST	Code-Signing Tool
DEK	Data Encryption Key
DER	ASN.1 Distinguished Encoding Rules
ELE	EdgeLock secure Enclave
HAB	High Assurance Boot
HAB4	High Assurance Boot Version 4
AHAB	Advanced High Assurance Boot
HSM	Hardware Security Module
MMU	Memory Management Unit
OS	Operating System
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastructure
PKCS	Public Key Cryptography Standards
RVT	ROM Vector Table
RSA	Public key encryption algorithm created by Rivest, Shamir and Adleman
SA	Signature Authority
SHA	Secure Hash Algorithm
SoC	System on Chip
SRK	Super Root Key
SW	Software
UID	Unique ID — a field in the processor and CSF identifying a device or group of devices

References

The following sources were referenced to produce this book:

1. *Open Secure Socket Layer (OpenSSL)*, http://www.openssl.org.
2. *RFC 3369: Cryptographic Message Syntax (CMS),* http://www.ietf.org/rfc/rfc3852.txt
3. *RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, http://www.ietf.org/rfc/rfc5280.txt
4. *RSA Private-Key Cryptography Standard #8 (PKCS #8) - Private-Key Information Syntax Standard*, version 1.2, RSA Laboratories, http://www.rsa.com/rsalabs.
5. *WAP Certificate and CRL Profiles (WAP-211-WAPCert)*, 22-May-2001, http://www.openmobilealliance.org
6. *RFC 3610: Counter with CBC-MAC (CCM)*, http://www.ietf.org/rfc/rfc3610.txt
7. *PKCS#11 wrapper library, 12/2023*. https://github.com/OpenSC/libp11
8. *The p11-kit web pages, 12/2023*. http://p11-glue.freedesktop.org/p11-kit.html
9. *The PKCS #11 URI Scheme, 12/2023*. https://tools.ietf.org/html/rfc7512

Additional Documents

The following documents provide additional information on secure boot with NXP processors

8. *High Assurance Boot Version 4 Application Programming Interface Reference Manual (HAB4_API)*. Included as part of the NXP Reference CST release.
9. *AN4547: Secure Boot on i.MX25, i.MX35, and i.MX51 using HABv3,* http://www.nxp.com
10. *AN4555: Secure Boot with i.MX28 HAB v4,* http://www.nxp.com
11. *AN4581: i.MX Secure Boot on HABv4 Supported Devices ,* http://www.nxp.com
12. *AN12312: Secure Boot on AHAB Supported Devices* http://www.nxp.com
13. *i.MX 6 DQ/DQP/SDL/SX/UL/ULL/ULZ, i.MX 7 SD/ULP, i.MX 8 MQ/MM/MN/MP/ULP/QXP/DXP/QM/DM/DXL, i.MX 9x Security Reference Manual* http://www.nxp.com

1 Introduction

The Code Signing Tool provides support to sign and encrypt images for use with HAB and AHAB enabled NXP processors.

1.1 Code Signing Components

The secure boot feature using HAB or AHAB included in many NXP processors is based on Public Key Infrastructure. The secure systems consist of two main components:

- The HAB library sub-component of NXP Processor Boot ROMs or the AHAB secure sub-system including a dedicated ARM core, ROM and FW.
- The CST

1.1.1 Secure components

The HAB library is a sub-component of the boot ROM and the AHAB component is a complete sub-system on select NXP processors. They are responsible for verifying the digital signatures included as part of the product software and ensures that, when the processor is configured as a secure device, no unauthenticated code is allowed to run. On NXP processors supporting the feature, encrypted boot may also be used to provide image cloning protection and, depending on the use case, image confidentiality. The secure components cannot only be used to authenticate the first stage of the boot chain, but the other components of the boot chain as well. The use of HAB or AHAB is bootloader and OS agnostic. An example is shown in Figure 1 and 2 for a generic boot chain.

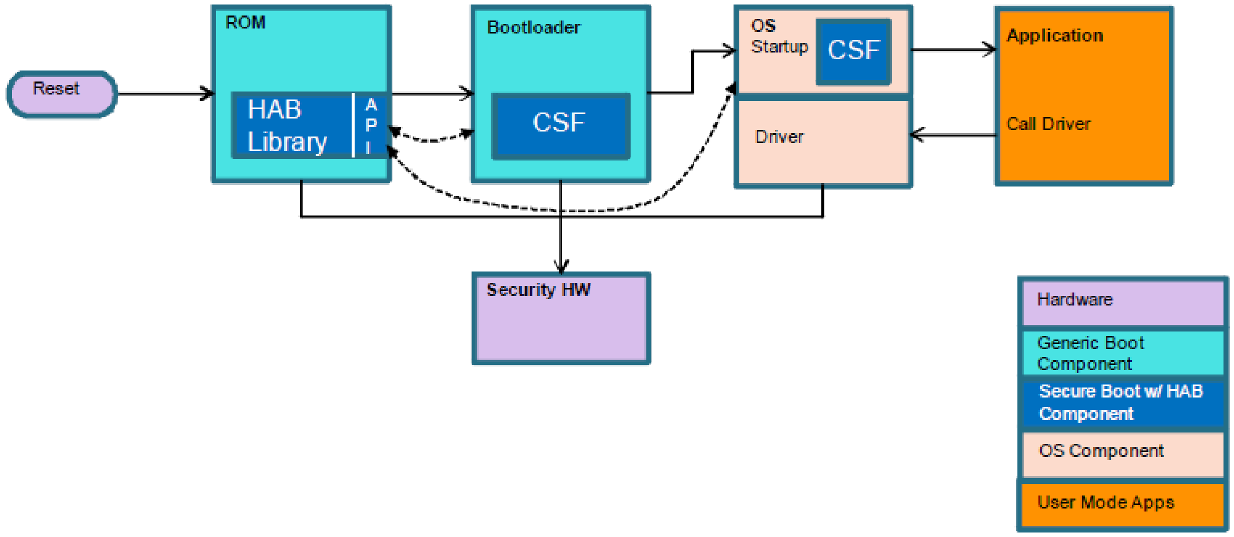

Figure 1. Generic Boot Flow using HAB

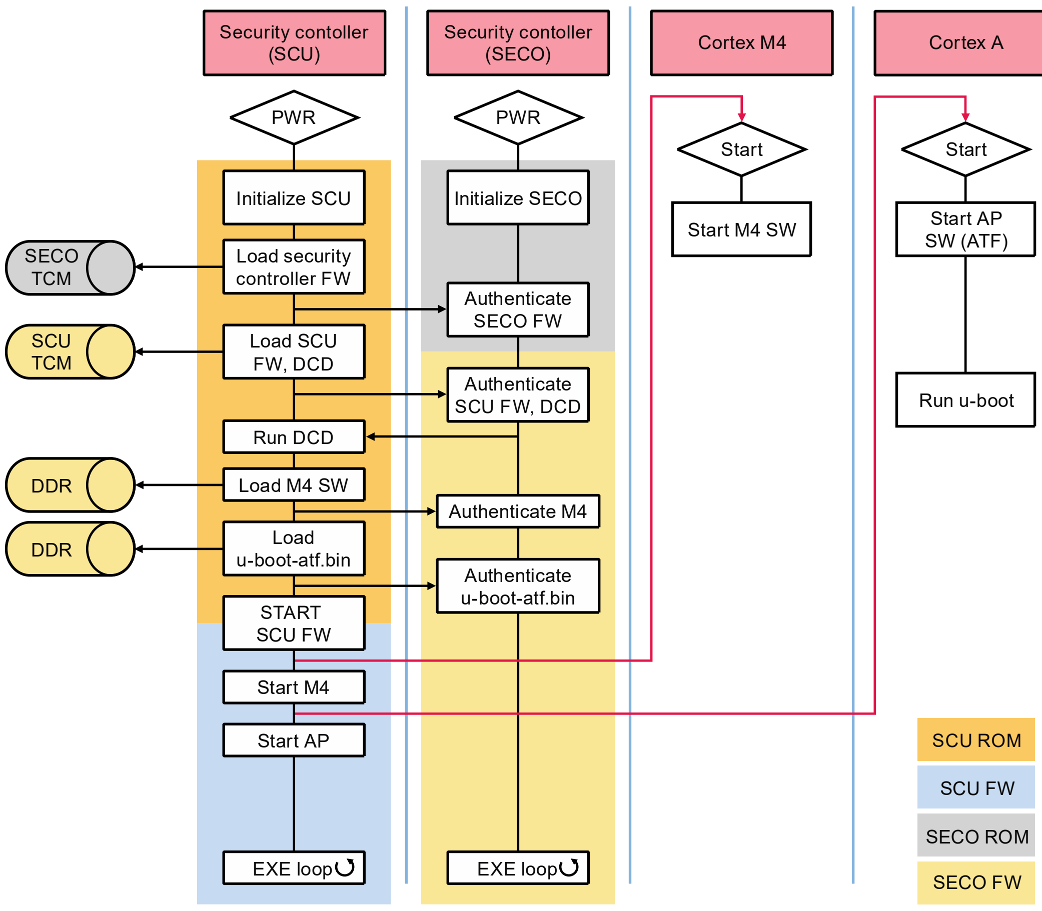

Figure 2. Generic Boot Flow using AHAB (SECO ROM & FW)

The secure boot process starts with the ROM reading eFuses to determine the security configuration of the SoC and the type of the boot device. The ROM then loads the images to memory. For HAB, the bootloader image contains both the bootloader itself in addition to: commands that the HAB uses to verify the image, digital signature data and public key certificate data which are collectively called Command Sequence File (CSF) data. The CSF data is generated off-line using the Code-Signing Tool (CST) which is introduced in the next section. For AHAB, the boot image contains the user-provided images (for the user programmable cores) in addition to a container header and a signature block that the AHAB uses to verify the images, the digital signatures and public key certificate data. The container header is generated off-line, using the mkimage tool that is not described as part of this document. The signature block is generated off-line by the Code Signing Tool (CST) which is introduced in the next section. Once ROM has completed loading the images, execution is then passed to the secure components which will verify the signatures. If signature verification fails, execution is not allowed to leave the ROM for securely configured SoCs. The exact behavior on signature verification failure at the ROM stage is SoC dependent. If all signatures, including image decryption, are successful then execution is passed to the next images which can perform similar steps to verify the next boot stage by calling back into the secure API.

NOTE

The ROM, HAB and AHAB cannot be changed so they can be considered as trusted software components. This allows the use of ROM, HAB and AHAB to establish a secure boot chain.

HAB and AHAB require the use of physical addresses, so if an MMU and a Level 2 cache are enabled within the bootloader stage then the address translation must be idempotent. This ensures that all boot components provide HAB or AHAB with physical addresses. Once all boot components have been verified, HAB and AHAB are no longer needed and the MMU and Layer 2 cache may be re-configured as required by the Operating System (OS).

The ROM/HAB/AHAB library integration also provides access to the APIs that boot components outside the ROM may call for image verification. The exact implementation of API is SoC dependent so please refer to the Reference Manual for the NXP processor you are using for specific details.

There are two major versions of the secure components that exist on NXP processors: HAB Version 4 (HAB4) and AHAB version. HAB version supports the flow shown in Figure 1. AHAB version supports the flow shown in Figure 2. HAB4 and AHAB use public key signature verification to ensure that product code is authentic. There some differences between these versions which are highlighted in Table 1 below. Please see the reference manual for the NXP processor you are using to determine which version of HAB or AHAB is supported.

Table 1

Feature	HAB4	AHAB
Image Authentication	Yes	Yes
Super Root Key	Multiple, revocable, fused hash	Multiple, revocable, fused hash
Public Key Type	ECC-P256, ECC-P384, ECC-P521, RSA-1024, RSA-2048, RSA-3072, RSA-4096 (ECC in i.MX 8 MP only)	ECC-P256, ECC-P384, ECC-P521, RSA-2048, RSA-3072, RSA-4096
Certificate Format	X.509	NXP proprietary
Signature Format	CMS (PKCS#1) CMS (ECDSA) (HAB 4.5 and later)	ECDSA (raw format, no DER encoding), PKCS#1
Hash Algorithm	SHA-256	SHA-256, SHA-384, SHA-512
Image Decryption	Yes (HAB4.1 and later)	Yes (SECO FW v2.3.0 and later, ELE 0.0.10 and later)
Image Decryption Algorithm	AES-CCM	AES-CBC
Image Decryption Key Blob Algorithm	NXP proprietary	NXP proprietary

Wrapped Key Format	CAAM Blob - Secret keys stored in CAAM secure RAM partition	CAAM Blob – Secret keys stored in CAAM secure RAM partition
Secret Key Type	AES-128/192/256	AES-128/192/256
Decryption Algorithm	AES-CCM - authenticated decryption	AES-CCM – authenticated decryption
Unlock Commands	Field Return Fuse Revocation Fuses Secure JTAG etc.	Not applicable

1.1.1.1 Secure components API

In order to allow boot components outside the ROM to continue the secure boot chain it must be possible for these components to call back into the HAB or AHAB. There are two versions of the API, one for HAB4 and one for AHAB.

Information on the HAB4 API can be found in the HAB4 API Reference Manual.

Information on the AHAB API can be found in the i.MX8 QXP/QM SECO API or the i.MX8ULP / i.MX9 ELE API document.

1.1.2 CST

There are several participants involved when performing cryptographic signatures as illustrated in Figure 2. These include:

- A Certificate Authority (CA). The CA is responsible for protecting the top-level CA key and for certifying lower level code signing keys.
- A Signature Authority (SA). The SA is responsible for performing the act of code signing.
- A Manufacturer. The Manufacturer is responsible for requesting digital signatures across product software.

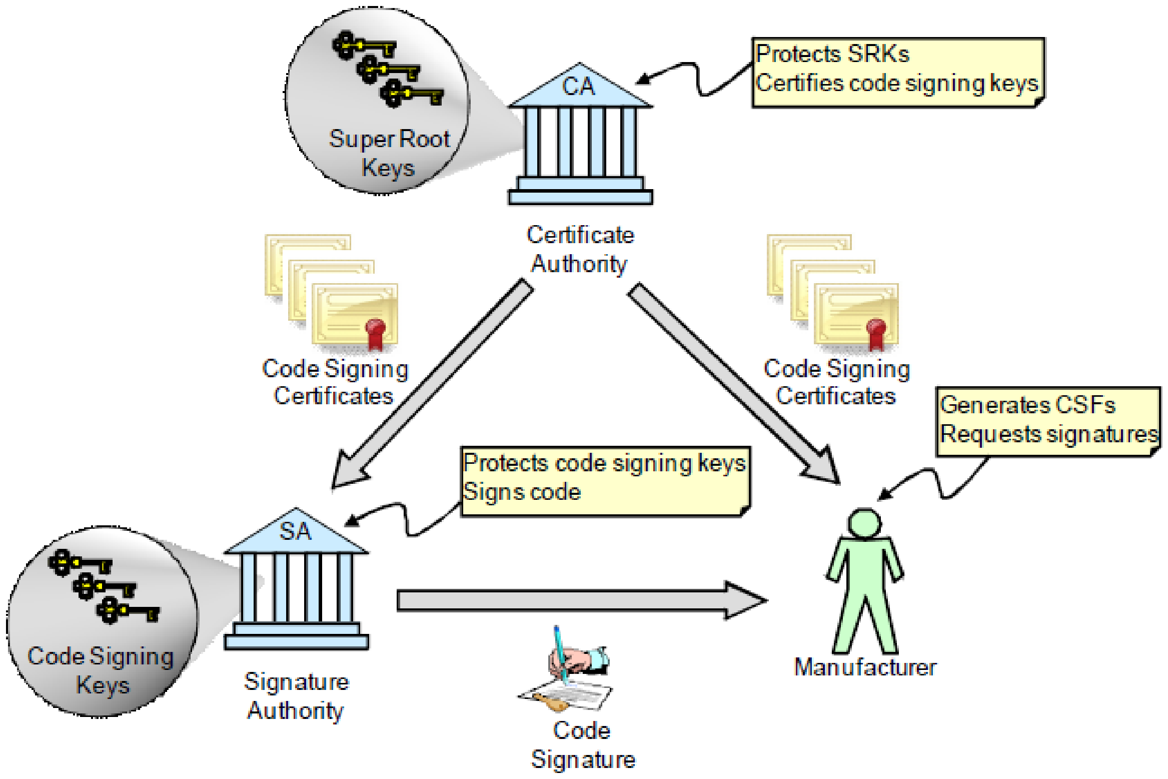

Figure 2. Generic Code Signing Participants

The CST is a set of command line tools residing on a host computer which serves as both the Certificate Authority (CA) and Signature Authority (SA) allowing manufacturers to control all aspects of the signing process.

The CST can establish a PKI tree of keys and certificates (CA function) needed for code signing in addition to generating digital signatures across data provided by a user (SA Function). The signatures generated by the CST can then be included as part of the end-product software image. The signatures are then verified by the secure components on the NXP processor at boot time.

Figure 3 shows how the CST is used to generate data which includes signatures, certificates and CSF commands (HAB only) the secure components in ROM will use to validate the product software. The CST takes two main inputs:

- A binary image or image(s) of the product software to be signed.
- A Command Sequence File (CSF). The CSF description file provides the instructions to the CST on what areas of the binary image need to be signed, which keys to sign the image with, etc.

The CST takes these inputs and generates binary data, which includes signatures, certificates and CSF commands (HAB only) that can then be attached to the product software to create a signed image. This User Guide focuses on the details of how to generate the key, certificates, CSF description files and how to run the CST executable.

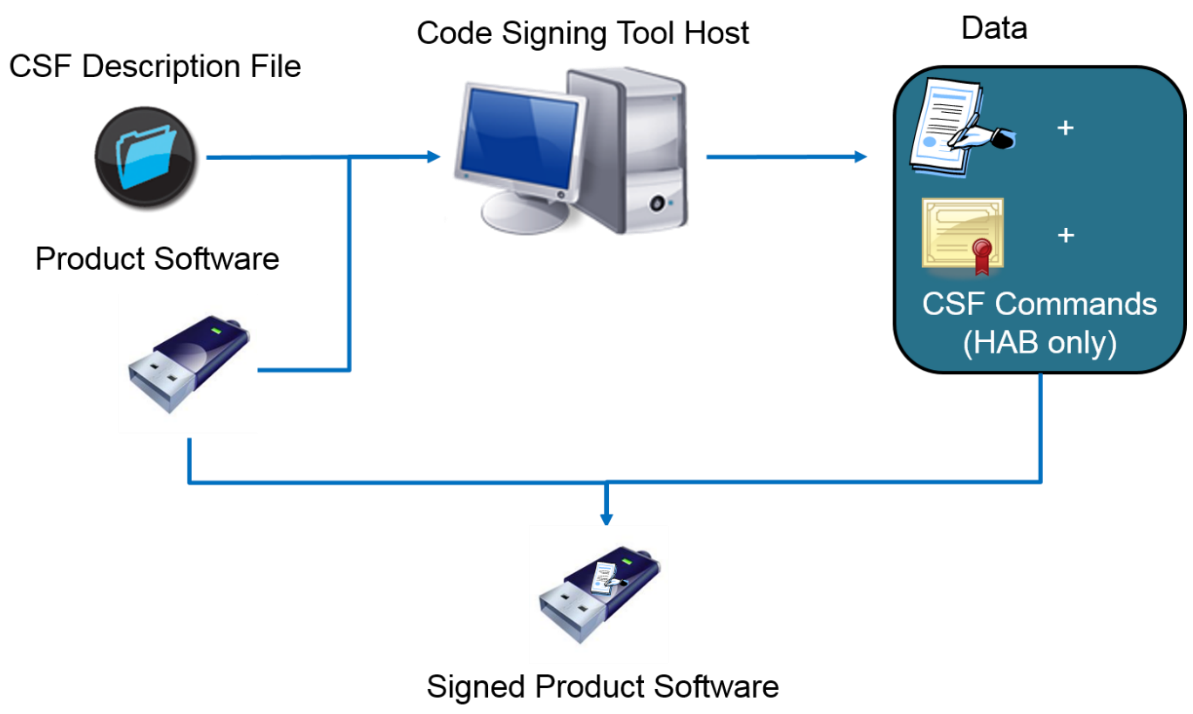

Figure 3. Code-Signing Tool - Digital Signatures

On certain NXP processors supporting HAB4 or AHAB, encrypted boot may also be used. Figure 4 shows the encrypted boot process with the CST. The encrypted boot case is very similar to generating signed images, but there are two main differences. The first is that the binary image is both decrypted and authenticated using a symmetric key rather than signed using a private asymmetric key. The second is the CST generates a one-time AES Data Encryption Key (DEK) which is used to encrypt the image. Note that when performing an encrypted boot digital signatures are still required, see Section 5.3.4 for an example CSF description file. The DEK is independent of the public keys used for code signing. The DEK output from the CST is protected but is not in the final form required for an encrypted boot on NXP processors.

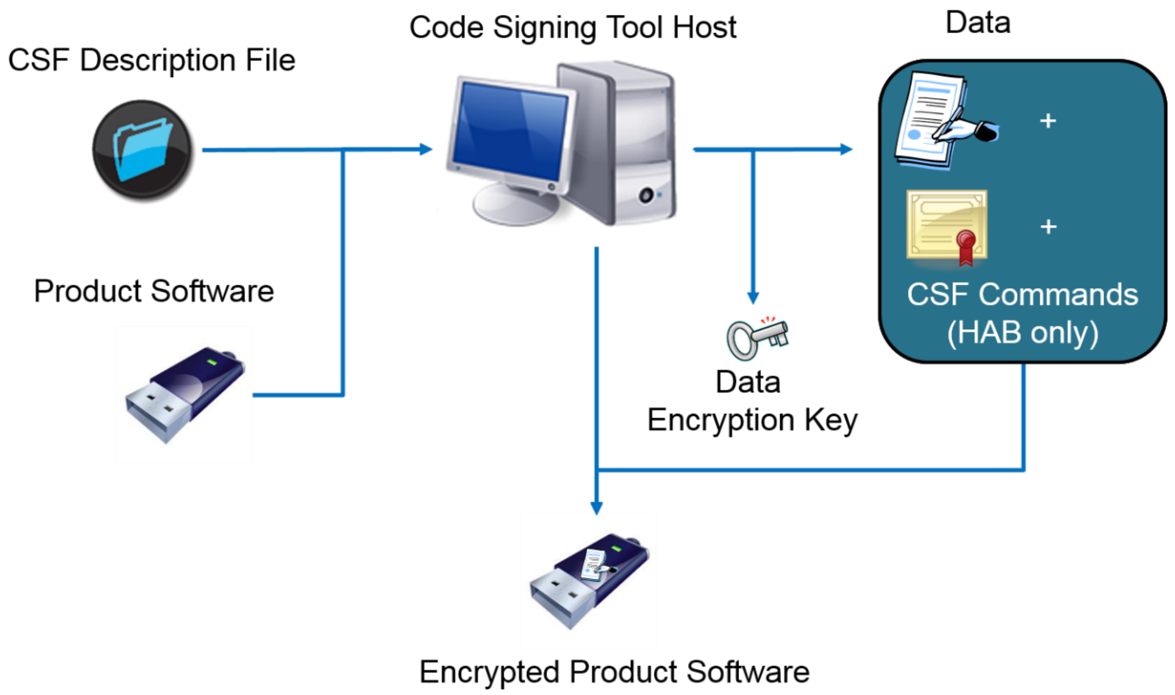

Figure 4. Code Signing Tool - Encrypted Boot

A cryptographic blob of the DEK must be created during the OEM manufacturing stages on each processor and then attached to the image on the boot device. The reason for this is the DEK blob is created using the device unique key embedded into the NXP processor which is only readable by the on-chip encryption engine. The DEK is common to all ICs using the same encrypted image but the DEK blob is unique per IC. Figure 5 provides an overview of DEK blob creation. The remaining details on DEK blob creation are beyond the scope of the CST and this document.

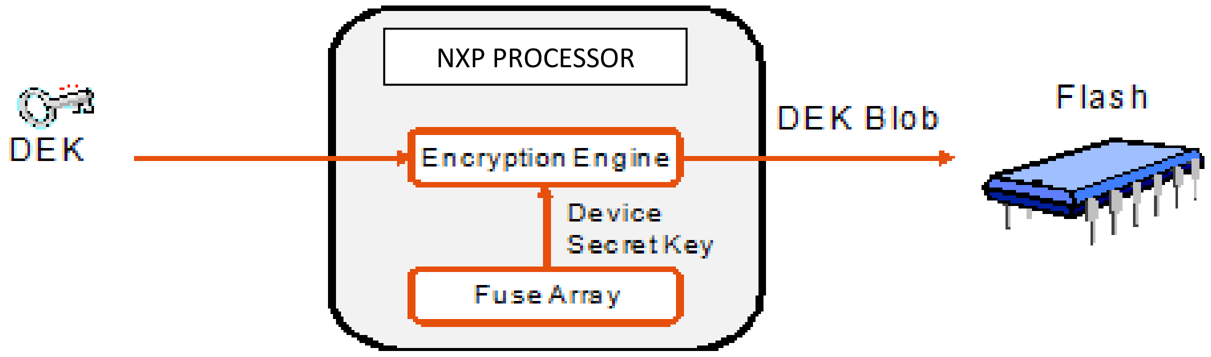

Figure 5. DEK Blob Creation

2 Installation

This section describes the installation of the CST code-signing client files.

2.1 CST Package Contents and Installation

The CST is delivered in an archive file, which contains a version for Linux and a version for Windows. For MacOS, the CST is no longer released with supported binaries, but the build system support for the MacOS target is retained in the source code. The archive contains a Software_Content_Register_CST.txt file that lists the entire contents of the archive.

2.1.1 Linux System Requirements

The following checklist can be used to ensure appropriate software is available for the Linux CST. Check with your system administrator if any components are missing.

Table 2.a. Linux — CST System Requirements Checklist

Required Component	
❑	A Linux distribution: Ubuntu 18.04 through 22.04 known to work although other distributions should also work but have not been formally tested. • Check by viewing information shown on Linux login screen
❑	OpenSSL 3.2.0 is recommended. Versions preceding it are also verified to be compatible with this build. Required for included scripts to generate public key infrastructure (PKI). • Check by running "`openssl version`" • Available at http://www.openssl.org/

NOTE

The NXP Reference CST uses the Linux OS to generate random numbers for use as keys for encrypted boot. Given this, the Linux host on which the reference CST is installed MUST have good sources of entropy. Generally, this requires multiple entropy sources such as keyboard input, mouse input, network packet arrival times etc. Running the CST without these sources of entropy will cause lengthy delays in seeding the Linux random number generator.

2.1.2 Windows System Requirements

The following checklist can be used to ensure appropriate software is available for the MS Windows CST. Check with your system administrator if any components are missing.

Table 2.b. Windows — CST System Requirements Checklist

Required Component

❑	Windows 7 32bit and Windows 10 64bit are known to work. • Check by viewing information shown on system panel
❑	OpenSSL 3.2.0 is recommended. Versions preceding it are also verified to be compatible with this build. Required for included scripts to generate public key infrastructure (PKI). • Check by running "`openssl version`" Available at http://www.openssl.org/ Note: OpenSSL 1.1.1 can be used with the restriction to convert the password file ("key_pass.txt") to Unix format. The small tool `convlb.exe` that can be found within the *keys* directory does this conversion when using the script for generating the keys. Note: it may happen that the OpenSSL Windows installer does not set the PATH environment variable. Please make sure this variable is set to the OpenSSL bin directory.

NOTE

The NXP Reference CST uses the Windows OS to generate random numbers for use as keys for encrypted boot. Given this, the Windows host on which the reference CST is installed MUST have good sources of entropy. Generally, this requires multiple entropy sources such as keyboard input, mouse input, network packet arrival times etc. Running the CST without these sources of entropy will cause lengthy delays in seeding the Windows random number generator.

2.1.3 Unpacking the Files

Unpack the CST archive to the desired installation point. The following is an example for Linux and assumes that the client archive was saved in a directory named /home/<*username*>/cst:

```
$ cd /home/<username>/cst/
$ tar -zxvf <release package name>.tgz
```

This creates the following directories:

ca/

Contains the OpenSSL configuration files. These configuration files are used when generating signing keys and certificates with the OpenSSL command line tool.

code/

The `/ahab_signature_block_parser` directory contains the sources and headers to build a parser for evaluating AHAB images and provide signature details. More detailed information can be found in the README file located in the directory.

The `/cst` directory contains the sources and headers to build CST. More detailed information can be found in the Release Notes.

The `/hab_csf_parser` directory contains the sources and headers necessary to build a parser of the HAB CSF binaries generated by CST. More detailed information can be found in the README file located in the directory.

The `/hab_srktool_scripts` directory contains scripts that mimic the SRKTOOL executable behavior. More detailed information can be found in the README file located in the directory.

crts/

Contains the public key certificates used for signing. Initially this directory is empty.

docs/

Contains the CST user guide and HAB4 API guide.

keys/

Contains the private key files used for signing. Initially this directory contains scripts to generate the PKI tree:

`hab4_pki_tree.sh` — Use to generate a series of keys and certificates on a Linux or MacOS machine for use with an NXP processor supporting HAB4.

`hab4_pki_tree.bat` — Use to generate a series of keys and certificates on a Windows machine for use with an NXP processor supporting HAB4.

`ahab_pki_tree.sh` — Use to generate a series of keys and certificates on a Linux or MacOS machine for use with an NXP processor supporting AHAB.

`ahab_pki_tree.bat` — Use to generate a series of keys and certificates on a Windows machine for use with an NXP processor supporting AHAB.`add_key.sh` — Use to add new keys to an existing HAB4 PKI tree.

`convlb.exe` — Use to convert the line breaks of Windows text files to Unix format. Use to work around OpenSSL 1.0.2 limitations with file handling.

linux32/

Contains the CST executables for 32-bit Linux OS:

`bin/cst` — The CST executable used to sign code `bin/srktool`— Generate SRK table and e-fuse files for HAB4 or AHAB.

`bin/hab_log_parser` — Parse HAB persistent memory dumps and print out the HAB events.

linux64/

Contains the CST executables for 64-bit Linux OS:

`bin/cst` — The CST executable used to sign code

`bin/srktool`— Generate SRK table and e-fuse files for HAB4 or AHAB.

`bin/hab_log_parser` — Parse HAB persistent memory dumps and print out the HAB events.

mingw32/

Contains the CST executables for MS Windows:

`bin/cst.exe` — The CST executable used to sign code

`bin/srktool.exe`— Generate SRK table and e-fuse files for HAB4 or AHAB.

`/lib` — Contains library files needed for replacing the CST backend implementation.

Once the archive is unpacked, there are no additional installation steps required in order to use the CST.

3 Key and Certificate Generation

Once the CST installation is complete. The first step in signing code is generating private keys and certificates. The CST is not delivered with keys or certificates since these will be different for each manufacturer and perhaps even each product line.

The NXP reference CST generates keys by making use of the OpenSSL command line tool and a set of shell scripts for Linux. This makes OpenSSL the CA component shown in Figure 2. The provided scripts illustrate how to generate a PKI tree of keys and certificates. There are three sets of scripts generating an initial PKI tree. One for HAB4 and one for AHAB. The reason for this is that the PKI tree structure is different for each version as well as the final public key certificate format. HAB4 and AHAB requires X.509 [3] format certificates. The provided key and certificate generation scripts are for reference to illustrate how they should be generated with OpenSSL. Users may update these scripts or replace these scripts with something equivalent if required.

CAUTION

The NXP reference CST requires a one-to-one correspondence between the key names in the /keys directory and the certificates in /crts directory. The convention is <keyname>_key.<ext> for keys and <keyname>_crt.<ext> for certificates. For example, a key named keys/SRK1_sha256_2048_65537_v3_ca_key.der must have a corresponding certificate crts/SRK1_sha256_2048_65537_v3_ca_crt.der.

3.1 Generating HAB4 Keys and Certificates

This section covers only key and certificate generation for HAB4. Note that when making use of the encrypted boot feature digital signatures are still required. Data structures required by ROM and HAB cannot be encrypted but still must be covered by a valid digital signature. Also, a new symmetric key is dynamically generated by the CST for each Install Secret Key/Decrypt Data command pair. These symmetric keys are an output of the CST and encrypted with a supplied public key. See Section 5.3.4 for an example encrypted boot CSF file.

3.1.1 HAB4 PKI Tree

The tree structure for HAB4 generated by the `hab4_pki_tree.sh` script for Linux. This script will generate a HAB4 PKI tree as shown in Figure 11 and is in the /keys directory of the NXP Reference CST.

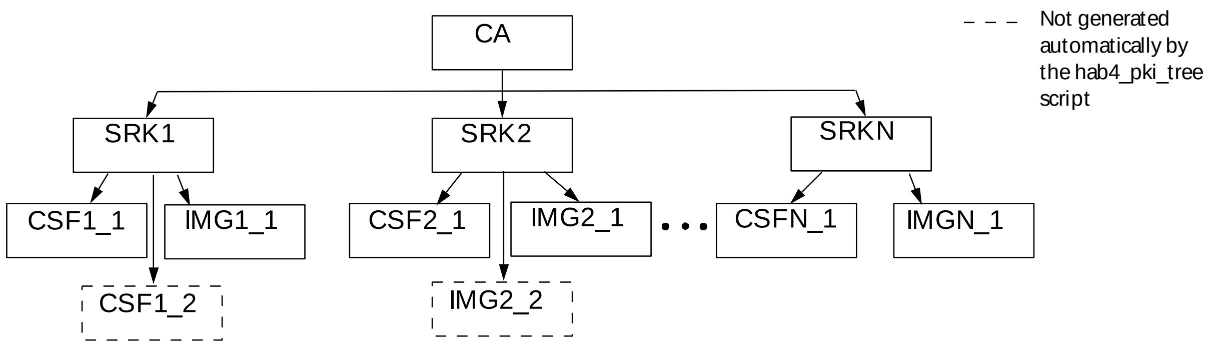

Figure 11. HAB4 PKI Tree

A HAB4 PKI tree consists of the following keys and certificates:

- CA key: is the top most key and is only used for signing SRK certificates.
- SRK: is the root key for HAB code signing keys. The cryptographic hash of a table of SRK is burned to one-time programmable efuses to establish a root of trust. Only one of the SRKs in the table may be selected for use on the NXP processor per reset cycle. The

selection of which SRK to use is a parameter within the Install Key CSF command (see Section 5.2.2, "Install SRK"). The SRK may only be used for signing certificate data of subordinate keys.

- CSF: is a subordinate key of the SRK and is used to verify the signature across CSF commands.
- IMG: is a subordinate key of the SRK key and is used to verify signatures across product software.
- NOTE: The CSF and IMG keys are not generated for a fast authentication PKI tree

The hab4_pki_tree script generates a basic tree in which up to a maximum of four SRKs may be generated. For each SRK a single CSF key and IMG key are also generated. Additional keys may be added to the tree later using a separate script. It is also possible to replace the OpenSSL and the hab4_pki_tree script with an alternative key generation solution, but this is beyond the scope of this document. If the key generation scheme described here is replaced a new scheme must follow these constraints:

- Keys must be in PKCS#8 format
- Certificates must be in X.509 format following the certificate profile specified by HAB4. Keys and Certificates must follow the file naming convention specified in Section 3, "HAB Key and Certificate Generation".

3.1.2 Running the hab4_pki_tree script Example

The hab4_pki_tree script can be run in two modes, a) Interactive, b) Command line interface (CLI).

The following are the common steps between interactive and CLI mode to generate a HAB4 PKI tree for Linux.

2. `cd <CST Installation Path>/keys`
3. Using your favorite text editor create a file called 'serial' in the /keys directory with contents 12345678. OpenSSL uses the contents of this file for the certificate serial numbers. You may choose to use another number for the initial certificate serial number.
4. Using your favorite text editor create a file called 'key_pass.txt' in the /keys directory. This file contains your pass phrase that will protect the HAB code signing private keys. The format of this file is the pass phase repeated on the first and second lines of the file. For example:

   my_pass_phrase

   my_pass_phrase

NOTE

Failure to generate the serial and key_pass.txt files prior to running the hab4_pki_tree script will result in OpenSSL errors and the script will fail to generate the requested tree.

CAUTION

It is up to the user how best to protect the pass phrase for the private keys. Loss of the pass phrase will result in not being able to sign code with the affected keys.

NOTE

Note that OpenSSL enforces that the pass phrase must be at least four characters long.

5. Prior to running the `hab4_pki_tree.sh` ensure that OpenSSL is included in your search path by running:

   ```
   > openssl version
   ```

3.1.2.1 Running the hab4_pki_tree script in interactive mode

Run the `hab4_pki_tree.sh` script in interactive mode. The script will ask a series of questions:

- Do you want to use an existing CA key (y/n)?
  - Choose no here unless you already have an existing CA key.
  - If you choose yes, the script will ask you to provide the filenames (including path information) to the location of the CA key and corresponding CA public key certificate.
- Do you want to use Elliptic Curve Cryptography (y/n)?:
  - If "n", Enter key length in bits for PKI tree:
    - This is the length in bit for the keys in the tree. For HAB4 1024, 2048, 3072 and 4096-bit RSA keys are supported. All keys in the tree are generated with the same length.
  - If "y", Enter length for elliptic curve to be used for PKI tree: Possible values p256, p384, p521:
    - This is the length in bit for the keys in the tree. For HAB4 P256, P384 and P521 EC keys are supported. All keys in the tree are generated with the same length.
- Enter PKI tree duration (years):
  - This defines the validity period of the corresponding certificates.
- How many Super Root Keys should be generated?
  - Up to four SRKs may be generated by this script. This allows for up to four SRKs to be included in a HAB4 SRK table. See Section 4.2, "SRK Tool" for further details.
- Do you want the SRK certificates to have the CA Flag set?

– Answer ‘y’ for a standard tree, ‘n’ for fast authentication tree.

Figure 12 below illustrates the use of the hab4_pki_tree script.

```
$ ./hab4_pki_tree.sh
...
<snip>
...
Do you want to use an existing CA key (y/n)?: n

Key type options (confirm targeted device supports desired key type):
Select the key type (possible values: rsa, rsa-pss, ecc)?: rsa
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 1
Do you want the SRK certificates to have the CA flag set? (y/n)?: n
A default 'serial' file was created!
A default file 'key_pass.txt' was created with password = test!

+++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
+++++++++++++++++++++++++++++++++++++

Generating a RSA private key
...................................+++++
.....+++++
writing new private key to 'temp_ca.pem'
-----

++++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
++++++++++++++++++++++++++++++++++++++++

.............................................................................
..........................................................+++++
.....................................+++++
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'SRK1_sha256_2048_65537_v3_usr'
Certificate is to be certified until Dec 12 21:44:07 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

Figure 12. Example Usage of the hab4_pki_tree Script

3.1.2.2 Running the hab4_pki_tree script in CLI mode

The hab4_pki_tree script is run in CLI mode by providing the necessary inputs to the script at the time of execution. This interface can be helpful when automation is required. Following are the inputs related to the CLI:

./hab4_pki_tree:.sh -existing-ca <y/n> [-ca-key <CA key name> -ca-cert <CA cert name>] -use-ecc <y/n> -kl <ECC/RSA Key Length> -duration <years> -num-srk <1-4> -srk-ca <y/n>

Options:

- **-existing-ca:** Choose whether to use an existing CA key

  Valid inputs:

  "y" – Provide **-ca-key** with CA key filename and **-ca-cert** with CA public key certificate filename (including path information)

  "n" – Existing CA key is not selected
- **-use-ecc:** Choose whether to use Elliptic Curve Cryptography (ECC) or RSA

  Valid inputs:

  "y" – ECC keys will be generated for the PKI tree

  "n" – RSA keys will be generated for the PKI tree
- **-kl** – Enter key length for the key type selected
  - If **-use-ecc** is "y" then provide **-kl** with length for elliptic curve to be used for PKI tree: Possible values p256, p384, p521
  - If **-use-ecc** is "n" then provide **-kl** with length in bit for the RSA to be used for PKI tree: Possible values for HAB4 1024, 2048, 3072 and 4096-bit
- **-duration:** Enter PKI tree duration (in years)
- **-num-srk:** Enter up to four SRKs (1 – 4)
- **-srk-ca:** Choose whether the SRK certificates are to have the CA Flag set

  Valid inputs:

  "y" – A standard PKI tree will be created

  "n" – A fast authentication PKI tree will be created.

For more information on these options, please refer to the details in interactive mode section.

At this point the script will generate the SRK, CSF and IMG keys and certificates in the /keys and /crts directory. The generated keys will exist in PKCS#8 [4] format in both PEM and DER forms.

Certificates are in the /crts directory X.509 [3] format in both PEM and DER format. The `cst` will accept key and certificate files in either PEM or DER form.

NOTE

You may notice that there are several pem files such as 12345678.pem, serial.old, index.txt.attr and so on. These files are left over from the OpenSSL key and certificate generation process.

At this point all key and certificate information required for signing an image for HAB4 is now available.

3.1.3 Generating HAB4 SRK tables and Efuse Hash

The previous section discussed the steps to generate the keys and certificates for a HAB4 PKI tree. Now that they have been generated, the next step is to generate a HAB4 SRK table and corresponding hash value for burning to efuses on the SoC. In HAB4 it is possible to include up to four SRKs in a signed image, although only one may be used per reset cycle. By collecting SRKs in a table it is possible to select one of the SRKs at boot time. The Install SRK CSF command (see Section 5.2.2, "Install SRK") selects which SRK to use from the table to establish the root of trust. Any of the SRKs in the table may be selected without having to change the SRK_HASH value burned to efuses on the SoC.

This is useful on NXP processors where additional fuses are available for SRK revocation. That is, in the event one or more of the SRKs in the table are compromised, efuses corresponding to the compromised keys can be burned preventing those SRKs from ever being used again. This is enforced by the HAB library. The next SRK in the table can be used to sign new images. A minimum of one and maximum of four SRKs can be placed in an SRK table.

NOTE

Only the first three SRKs in a table can be revoked, so it recommended to use an SRK table with four keys in order to have one SRK to fall back on which cannot be revoked.

SRK tables are generated using the `srktool`. The following illustrates the generation of an SRK table from the /crts directory using the four SRKs created in the previous section.

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e
SRK_1_2_3_4_fuse.bin -d sha256 -c
./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_crt.pem
,./SRK3_sha256_2048_65537_v3_ca_crt.pem,./SRK4_sha256_2048_65537_v3_ca_crt.pe
m -f 1
Number of certificates    = 4
SRK table binary filename = SRK_1_2_3_4_table.bin
SRK Fuse binary filename  = SRK_1_2_3_4_fuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0x5B31CBE9
SRK HASH[1] = 0x6DE304C8
SRK HASH[2] = 0x99F821DE
SRK HASH[3] = 0x2803B237
SRK HASH[4] = 0xC8EF0FF8
SRK HASH[5] = 0x12F30689
SRK HASH[6] = 0xF38CE4A3
SRK HASH[7] = 0x39669C00
```

Figure 13. SRK Table and Efuse Generation Example

In this example:

- All four SRKs are included in the table
- The SHA-256 hash value is generated with 32 bit of fuse data per word. Some NXP processors require the hash value to be generated with 8 bits of fuse data per word. In that case use the '-f 0' option.
- The hash result in the resulting `SRK_1_2_3_4_fuse.bin` file is in little endian format. This means that the first byte in the file corresponds to SRK_HASH[255:248] and the last byte corresponds to SRK_HASH[7:0] in the efuse map. Similarly, when using the '-f 0' option the first non-zero byte in the file corresponds to SRK_HASH[255:248] and the last non-zero byte corresponds to SRK_HASH[7:0].

CAUTION

Do not enter spaces between the ',' when specifying the SRKs in the -c or --certs option. Doing so will cause all certificates specified after the first space ***not*** to be included in the table and resulting efuse hash.

3.1.4 Programming the SRK Hash Value to Efuses

The previous section provided the details on how to SRK tables and the corresponding efuse data. In this section the hash value is of interest. The value located in the efuse file is intended to be burned to the SRK_HASH efuse field on the SoC supporting HAB4 and is computed automatically by the hab4_pki_tree script using the `srktool`. The `SRK1_2_3_4_fuse.bin` file from the example in the previous section has the following contents:

```
e9cb315bc804e36dde21f89937b20328f80fefc88906f312a3e48cf3009c
```

This hash value must be burned to the SoC efuses in the following order:

```
SRK HASH[0] = 0x5B31CBE9
SRK HASH[1] = 0x6DE304C8
SRK HASH[2] = 0x99F821DE
SRK HASH[3] = 0x2803B237
SRK HASH[4] = 0xC8EF0FF8
SRK HASH[5] = 0x12F30689
SRK HASH[6] = 0xF38CE4A3
SRK HASH[7] = 0x39669C00
```

Figure 14. SRK Hash Value Assignment to SoC SRK_HASH Efuse Field for HAB4

Please refer to the fuse map for the NXP processor you are using for location details of the SRK_HASH field.

3.1.5 Adding a Key to a HAB4 PKI Tree

Adding to an existing HAB4 PKI tree can be done using the add_key script. The add_key script can be run in two modes, a) Interactive, b) Command line interface (CLI).

3.1.5.1 Running the add_key script in interactive mode

The following steps are used to add a new key by running add_key script in interactive mode:

1. Run the `add_key.sh` script for Linux. The script will prompt you with several questions:
   - Which version of HAB/AHAB do you want to generate the key for (4 = HAB4 / a = AHAB)
   - Enter new key name (e.g. SRK5):
     - This the name of the new key, such as SRK2, CSF1_2, etc.
   - Enter new key type:
     - This can be either ECC or RSA
   - Enter new key length in bits:
     - If the This is the length of the new key in bits. This should match the key length of the signing key.
   - Enter certificate duration (years):
     - This defines the validity period for the corresponding certificate generated
   - Is this an SRK key?
     - If you are generating a new SRK enter ‘y’, otherwise enter ‘n’
     - If you enter no , it implies that you are generating a CSF/IMG/SGK key.
   - Enter <key type> signing key name:
     - If you are generating a new SRK <key type> is CA. Enter the path and filename of the CA key in the /keys directory.
     - If you are generating a new CSF/IMG/SGK key <key type> is SRK. Enter the path and filename of the SRK in the /keys directory you wish to use to generate the CSF/IMG/SGK key.
   - Enter <cert type> signing certificate name:
     - If you are generating a new SRK certificate the <cert type> is CA. Enter the path and filename of the CA certificate in the /crts directory.
     - If you are generating a new CSF/IMG/SGK certificate <cert type> is SRK. Enter the path and filename of the SRK certificate in the /certs directory you wish to use to generate the CSF/IMG/SGK certificate.

Using the keys generated in Section 3.2.2, "Running the hab4_pki _tree script Example", Figure 15 below shows how to add a new SRK key to the PKI tree.

```
$ ./add_key.sh
Which version of HAB/AHAB do you want to generate the key for (4 = HAB4 / a =
AHAB)?: 4
Enter new key name (e.g. SRK5):
Enter new key type (ecc / rsa / rsa-pss): rsa
Enter new key length in bits: 2048
Enter certificate duration (years): 10
Is this an SRK key?: y
Do you want the SRK to have the CA flag set (y/n)?: y
Enter CA signing key name: CA1_sha256_2048_65537_v3_ca_key.pem
Enter CA signing certificate name:
../crts/CA1_sha256_2048_65537_v3_ca_crt.pem
.....................................................+++++
.................................+++++
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'_sha256_2048_65537_v3_ca'
Certificate is to be certified until Dec 12 22:06:16 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
```

Figure 15. Adding a New SRK to a HAB4 PKI Tree Example

3.1.5.2 Running the add_key script in CLI mode

The add_key script is run in CLI mode by providing the necessary inputs to the script at the time of execution. This interface can be helpful when automation is required. Following are the inputs related to the CLI:

./add_key.sh -ver <4/a> -key-name <new key name> -kt <ecc/rsa> -kl <key length> [-md <message digest>] -duration <years> -srk <y/n> [-srk-ca <y/n>] -signing-key <CA/SRK signing key> -signing-crt <CA/SRK signing cert>

Options:

— **-ver**: Enter the version of HAB/AHAB you want to generate the key for (4/a)

Valid inputs:

"4" – HABv4 is selected

"a" – AHAB is selected

- **-key-name**: Enter new key name
- **-kt:** Enter key type of the new key (rsa/ecc)

  Valid inputs:

  "rsa" – RSA key type is selected

  "ecc" – ECC key type is selected
- **-kl:** Enter length of the new key in bits
  - If **-kt** is "rsa", then supported key lengths are 2048, 3072, 4096-bit
  - If **-kt** is "ecc", then supported key lengths are p256, p384, p521
- **-md:** Enter hashing function
  - If **-ver** is "a" (AHAB) then supported message digests are sha256, sha384, sha512
  - If **-ver** is "4" (HAB) then message digest is fixed to sha256
- **-duration:** Enter certificate duration (in years)
- **-srk:** Choose whether this an SRK key

  Valid inputs:

  "y" – A new SRK key is generated

  "n" – A new CSF/IMG/SGK key is generated
- **-srk-ca:** Choose whether the SRK certificates has the CA Flag set

  Valid inputs:

  "y" – The SRK certificate has CA flag set

  "n" – The SRK certificate does not have CA flag set
- **-signing-key:** Enter signing key filename (including path information)
  - If this is an SRK key then enter CA key filename based on whether the SRK is a CA cert or user cert
  - If this is a CSF/IMG/SGK key then enter the signing SRK key filename
- **-signing-crt:** Enter signing certificate filename (including path information)
  - If this is an SRK key then enter CA certificate filename based on whether the SRK is a CA cert or user cert
  - If this is a CSF/IMG/SGK certificate then enter the signing SRK certificate filename

For more information on these options, please refer to the details in interactive mode section.

CAUTION

Do not run this script without first generating a HAB4 PKI tree. Failure to do so will result in errors.

3.2 Generating AHAB Keys and Certificates

This section covers only key and certificate generation for AHAB. Note that when making use of the encrypted boot feature digital signatures are still required. Data structures required by ROM and AHAB cannot be encrypted but still must be covered by a valid digital signature.

NOTE

This section provides examples specifically for SECO-enabled devices. If you are using ELE-enabled devices, please refer to the SRKTool command line help, for guidance on the number of supported fuses.

3.2.1 AHAB PKI Tree

The tree structure for AHAB generated by the `ahab_pki_tree.sh` script for Linux. This script will generate a AHAB PKI tree as shown in Figure 11 and is located in the /keys directory of the NXP Reference CST.

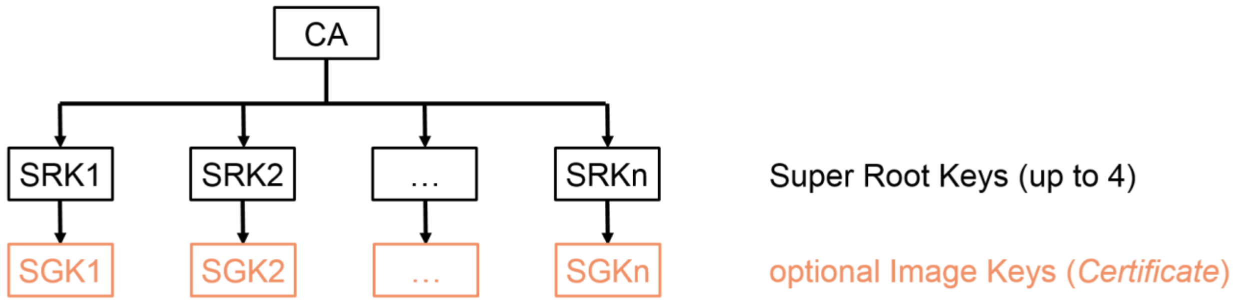
CA
SRK1
SRK2
…
SRKn
Super Root Keys (up to 4)
SGK1
SGK2
…
SGKn
optional Image Keys (*Certificate*)

Figure 11. AHAB PKI Tree

An AHAB PKI tree consists of the following keys and certificates:

- CA key: is the top most key and is only used for signing SRK certificates.
- SRK: is the root key for AHAB code signing keys. The cryptographic hash of a table of SRK is burned to one-time programmable efuses to establish a root of trust. Only one of the SRKs in the table may be selected for use on the NXP processor. The selection of which SRK to use is a parameter within the Install SRK CSF command (see Section 5.2.2, "Install SRK"). The SRK may only be used for signing certificate data of subordinate keys.
- SGK: is a subordinate key of the SRK key and is used to verify signatures across product software.
- NOTE: The SGK keys are not generated if the SRK keys do not have the CA flag set

The ahab_pki_tree script generates a basic tree in which four SRKs may be generated. For each SRK a single SGK key is also generated. Additional keys may be added to the tree later using a separate script. It is also possible to replace the OpenSSL and the ahab_pki_tree script with an alternative key generation solution, but this is beyond the scope of this document. If the key generation scheme described here is replaced a new scheme must follow these constraints:

- Keys must be in PKCS#8 format
- Certificates must be in X.509 format following the certificate profile specified by AHAB. Keys and Certificates must follow the file naming convention specified in Section 3, "Key and Certificate Generation".

3.2.2 Running the ahab_pki_tree script Example

The following are the common steps to generate an AHAB PKI tree in interactive or CLI mode.

1. `cd <CST Installation Path>/keys`
2. Using your favorite text editor create a file called 'serial' in the /keys directory with contents 12345678. OpenSSL uses the contents of this file for the certificate serial numbers. You may choose to use another number for the initial certificate serial number.
3. Using your favorite text editor create a file called 'key_pass.txt' in the /keys directory. This file contains your pass phrase that will protect the AHAB code signing private keys. The format of this file is the pass phase repeated on the first and second lines of the file. For example:

   my_pass_phrase

   my_pass_phrase

   NOTE

   Failure to generate the serial and key_pass.txt files prior to running the ahab_pki_tree script will result in OpenSSL errors and the script will fail to generate the requested tree.

   CAUTION

   It is up to the user how best to protect the pass phrase for the private keys. Loss of the pass phrase will result in not being able to sign code with the affected keys.

   NOTE

   Note that OpenSSL enforces that the pass phrase must be at least four characters long.

4. Prior to running the `ahab_pki_tree` script ensure that OpenSSL is included in your path by running:

   ```
   > openssl version
   ```

3.2.2.1 Running the ahab_pki_tree script in interactive mode

Run the `ahab_pki_tree` script in interactive mode. The script will ask a series of questions:

- Do you want to use an existing CA key (y/n)?
  - Choose no here unless you already have an existing CA key.
  - If you choose yes, the script will ask you to provide the filenames (including path information) to the location of the CA key and corresponding CA public key certificate.
- Do you want to use Elliptic Curve Cryptography (y/n)?
  - This is the type of the keys in the tree.
  - If you choose yes, the script will ask you to provide the Elliptic Curve to be used. For AHAB, P-256, P-384 and P-521curves are supported.
  - If you choose no, the script will ask you to enter the length in bit for the RSA keys in the tree. For AHAB 2048, 3072 and 4096-bit RSA keys are supported.
  - All keys in the tree are generated with the same length.
- Enter digest algorithm to use:
  - This is the digest algorithm used to create the keys
  - Allowed digest algorithms are: sha256, sha384 and sha512
- Enter PKI tree duration (years):
  - This defines the validity period of the corresponding certificates.
- Do you want the SRK certificates to have the CA Flag set?
  - Answer 'y' for a tree with *Certificates* as defined by the AHAB architecture.

Figure 12 below illustrates the use of the ahab_pki_tree script.

```
$ ./ahab_pki_tree.sh
...
<snip>
...
Do you want to use an existing CA key (y/n)?: n

Key type options (confirm targeted device supports desired key type):
Select the key type (possible values: rsa, rsa-pss, ecc)?: ecc
Enter length for elliptic curve to be used for PKI tree:
Possible values p256, p384, p521:  p384
Enter the digest algorithm to use: sha384
Enter PKI tree duration (years): 10
Do you want the SRK certificates to have the CA flag set? (y/n)?: n
```

```
++++++++++++++++++++++++++++++++++++++
+ Generating CA key and certificate +
++++++++++++++++++++++++++++++++++++++

Generating an EC private key
writing new private key to 'temp_ca.pem'
-----

+++++++++++++++++++++++++++++++++++++++++
+ Generating SRK key and certificate 1 +
+++++++++++++++++++++++++++++++++++++++++

read EC key
writing EC key
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'SRK1_sha384_secp384r1_v3_usr'
Certificate is to be certified until Dec 12 22:09:07 2033 GMT (3650 days)

Write out database with 1 new entries
Data Base Updated
...
<snip>
...
```

Figure 12. Example Usage of the AHAB_pki_tree Script

3.2.2.2 Running the ahab_pki_tree script in CLI mode

The ahab_pki_tree script is run in CLI mode by providing the necessary inputs to the script at the time of execution. This interface can be helpful when automation is required. Following are the inputs related to the CLI:

./ahab4_pki_tree.sh -existing-ca <y/n> [-ca-key <CA key name> -ca-cert <CA cert name>] -use-ecc <y/n> -kl <ECC/RSA Key Length> -da <digest algorithm> -duration <years> -srk-ca <y/n>

Options:

— **-existing-ca:** Choose whether to use an existing CA key

Valid inputs:

"y" – Provide **-ca-key** with CA key filename and **-ca-cert** with CA public key certificate filename (including path information)

"n" – Existing CA key is not selected

— **-use-ecc:** Choose whether to use Elliptic Curve Cryptography (ECC) or RSA

Valid inputs:

"y" – ECC keys will be generated for the PKI tree

"n" – RSA keys will be generated for the PKI tree

— **-kl** – Enter key length for the key type selected

- If **-use-ecc** is "y" then provide **-kl** with length for elliptic curve to be used for PKI tree: Possible values p256, p384, p521
- If **-use-ecc** is "n" then provide **-kl** with length in bit for the RSA to be used for PKI tree: Possible values for AHAB 2048, 3072 and 4096-bit

— **-da:** Enter digest algorithm:

- Valid inputs include sha256, sha384 and sha512

— **-duration:** Enter PKI tree duration (in years)

— **-num-srk:** Enter up to four SRKs (1 – 4)

— **-srk-ca:** Choose whether the SRK certificates have the CA Flag set

Valid inputs:

"y" – A standard PKI tree will be created

"n" – A fast authentication PKI tree will be created.

For more information on these options, please refer to the details in interactive mode section.

At this point the script will generate the SRK and SGK keys and certificates in the /keys and /crts directory. The generated keys will exist in PKCS#8 [4] format in both PEM and DER forms. Certificates are in the /crts directory X.509 [3] format in both PEM and DER format. The `cst` will accept key and certificate files in either PEM or DER form.

NOTE

You may notice that there are several pem files such as 12345678.pem, serial.old, index.txt.attr and so on. These files are left over from the OpenSSL key and certificate generation process.

At this point all key and certificate information required for signing an image for AHAB is now available.

3.2.3 Generating AHAB SRK tables and Efuse Hash

The previous section discussed the steps to generate the keys and certificates for an AHAB PKI tree. Now that they have been generated, the next step is to generate an AHAB SRK table and corresponding hash value for burning to efuses on the SoC. In AHAB four SRKs are included in a signed image, although only one may be used. By collecting SRKs in a table it is possible to select one of the SRKs at boot time. The Install SRK CSF command (see Section 5.2.2, "Install SRK") selects which SRK to use from the table to establish the root of trust. Any of the SRKs in the table may be selected without having to change the SRK_HASH value burned to efuses on the SoC.

This is useful on NXP processors where additional fuses are available for SRK revocation. That is, in the event one or more of the SRKs in the table are compromised, efuses corresponding to the compromised keys can be burned preventing those SRKs from ever being used again. This is enforced by the AHAB code. The next SRK in the table can be used to sign new images. Four SRKs can be placed in an SRK table.

NOTE

The four SRKs in a table can be revoked.

SRK tables are generated using the `srktool`. The following illustrates the generation of an SRK table from the /crts directory using the four SRKs created in the previous section.

```
$ ../linux64/bin/srktool -a -s sha384 -t SRK_1_2_3_4_table.bin -e
SRK_1_2_3_4_fuse.bin -f 1 -c
./SRK1_sha384_secp384r1_v3_usr_crt.pem,./SRK2_sha384_secp384r1_v3_usr_crt.pem
,./SRK3_sha384_secp384r1_v3_usr_crt.pem,./SRK4_sha384_secp384r1_v3_usr_crt.pe
m
Number of certificates    = 4
SRK table binary filename = SRK_1_2_3_4_table.bin
SRK Fuse binary filename  = SRK_1_2_3_4_fuse.bin
SRK Fuse binary dump:
SRK HASH[0] = 0xA83170EF
SRK HASH[1] = 0xFA1D5EA8
SRK HASH[2] = 0xE3A2C737
SRK HASH[3] = 0xAD1D0241
SRK HASH[4] = 0x6246BE44
SRK HASH[5] = 0x75439F14
SRK HASH[6] = 0x9F65FC0B
SRK HASH[7] = 0x6DAC9B80
SRK HASH[8] = 0x9481C935
SRK HASH[9] = 0x8C6CC5EC
SRK HASH[10] = 0x9104B9E5
SRK HASH[11] = 0x5F97C971
SRK HASH[12] = 0x0DA8DDA5
SRK HASH[13] = 0xAC21273D
SRK HASH[14] = 0x0FCE73F7
SRK HASH[15] = 0x3FC9ACBA
```

Figure 13. SRK Table and Efuse Generation Example

In this example:

- All four SRKs are included in the table
- The signature hash algorithm that will be used for signing is SHA-384 (option "-s")
- The SHA-512 hash value to be fused is generated with 32 bit of fuse data per word. Some NXP processors require the hash value to be generated with 8 bits of fuse data per word. In that case use the '-f 0' option.
- The hash result is in the resulting `SRK_1_2_3_4_fuse.bin` file.

CAUTION

Do not enter spaces between the ',' when specifying the SRKs in the -c or --certs option. Doing so will cause all certificates specified after the first space ***not*** to be included in the table and causing an execution error of `srktool`.

3.2.4 Programming the SRK Hash Value to Efuses

The previous section provided the details on how to SRK tables and the corresponding efuse data. In this section the hash value is of interest. The value located in the efuse file is intended to be burned to the SRK_HASH efuse field on the SoC supporting AHAB and is computed automatically by the AHAB_pki_tree script using the `srktool`. The `SRK_1_2_3_4_fuse.bin` file from the example in the previous section has the following contents:

```
ef7031a8a85e1dfa37c7a2e341021dad44be4662149f43750bfc659f809bac6d35c98194ecc56
c8ce5b9049171c9975fa5dda80d3d2721acf773ce0fbaacc93f
```

Here is the corresponding hexadecimal dump of the fuse file.

```
$ hexdump -C SRK_1_2_3_4_fuse.bin
00000000  ef 70 31 a8 a8 5e 1d fa  37 c7 a2 e3 41 02 1d ad  |.p1..^..7...A...|
00000010  44 be 46 62 14 9f 43 75  0b fc 65 9f 80 9b ac 6d  |D.Fb..Cu..e....m|
00000020  35 c9 81 94 ec c5 6c 8c  e5 b9 04 91 71 c9 97 5f  |5.....l.....q.._|
00000030  a5 dd a8 0d 3d 27 21 ac  f7 73 ce 0f ba ac c9 3f  |....='!..s.....?|
```

This hash value must be burned to the SoC efuses in the following order (the first word to the first fuse row index):

```
$ hexdump -e '/4 "0x"' -e '/4 "%X""\n"' SRK_1_2_3_4_fuse.bin
0xA83170EF
0xFA1D5EA8
0xE3A2C737
0xAD1D0241
0x6246BE44
0x75439F14
0x9F65FC0B
0x6DAC9B80
0x9481C935
0x8C6CC5EC
0x9104B9E5
0x5F97C971
0xDA8DDA5
0xAC21273D
0xFCE73F7
0x3FC9ACBA
```

Please refer to the fuse map for the NXP processor you are using for location details of the SRK_HASH field.

3.2.5 Adding a Key to an AHAB PKI Tree

Adding to an existing AHAB PKI tree can be done using the add_key script. The following steps are used to add a new key:

1. Run the `add_key.sh` script for Linux. The script will prompt you with several questions:
   - Which version of HAB/AHAB do you want to generate the key for (3/4/a)?
     - Enter a here for AHAB
   - Enter new key name (e.g. SRK5):
     - This is the name of the new key, such as SRK2, SGK3, etc.
   - Enter new key type (ecc / rsa):
     - This is the type of the new key, either ECC or RSA.
   - Enter new key length in bits:
     - This is the length of the new key in bits. This should match the key length of the signing key.
   - Enter new message digest:
     - This is the digest of the key signature.
   - Enter certificate duration (years):
     - This defines the validity period for the corresponding certificate generated
   - Is this an SRK key?
     - If you are generating a new SRK enter 'y', otherwise enter 'n'

  - If you enter yes, you will be prompted with “Do you want the SRK to have the CA flag set?”. Enter yes if you are generating a SRK with the CA flag set.
  - If you enter no, you are generating a new SGK key.
- Enter <key type> signing key name:
  - If you are generating a new SRK <key type> is CA. Enter the path and filename of the CA key in the /keys directory.
  - If you are generating a new SGK key <key type> is SRK. Enter the path and filename of the SRK in the /keys directory you wish to use to generate the SGK key.
- Enter <cert type> signing certificate name:
  - If you are generating a new SRK certificate the <cert type> is CA. Enter the path and filename of the CA certificate in the /crts directory.
  - If you are generating a new SGK certificate <cert type> is SRK. Enter the path and filename of the SRK certificate in the /certs directory you wish to use to generate the SGK certificate.

Using the keys generated in Section 3.2.2, “Running the AHAB_pki _tree script Example”, Figure 15 below shows how to add a new SRK key to the PKI tree.

```
$ ./add_key.sh
Which version of HAB/AHAB do you want to generate the key for (4 = HAB4 / a =
AHAB)?: a
Enter new key name (e.g. SRK5):
Enter new key type (ecc / rsa / rsa-pss): ecc
Enter new key length (p256 / p384 / p521): p384
Enter new message digest (sha256, sha384, sha512): sha384
Enter certificate duration (years): 10
Is this an SRK key?: y
Do you want the SRK to have the CA flag set (y/n)?: n
Enter CA signing key name: CA1_sha384_secp384r1_v3_ca_key.pem
Enter CA signing certificate name: ../crts/CA1_sha384_secp384r1_v3_ca_crt.pem
read EC key
writing EC key
Using configuration from ../ca/openssl.cnf
Check that the request matches the signature
Signature ok
The Subject's Distinguished Name is as follows
commonName            :ASN.1 12:'_sha384_secp384r1_v3_usr'
Certificate is to be certified until Dec 12 22:25:42 2033 GMT (3650 days)
```

```
Write out database with 1 new entries
Data Base Updated
```

Figure 15. Adding a New SRK to an AHAB PKI Tree Example

CAUTION

Do not run this script without first generating an AHAB PKI tree. Failure to do so will result in errors.

4 CST Usage

This section describes how to use the CST and other tools in the release package.

4.1 CST (Code Signing Tool)

The `cst` tool in the release package is the main application used to generate binary CSF data using input CSF description files passed as standard input. The CST can be executed from any location provided the correct absolute or relative path is provided. The paths to certificate and image files inside CSF can be either relative to the current working directory location or as absolute paths.

CAUTION

Due to limitation in current cst implementation the cst must be run from a directory at the same level as <Installation path>/keys. For example, <Installation path>/product_code where the product code to be signed is located.

Usage:

cst --output <*binary*> [--cert <cert_file>] --input <input_csf>

[--license] [--help]

Description:

```
    -o, --output <binary csf>:
        Output binary CSF filename

    -i, --input <csf text file>:
        Input CSF text filename

    -c, --cert <public key certificate>:
        Optional, Input public key certificate to encrypt the dek

    -b, --backend <ssl or pkcs11>:
        Optional, Select backend. SSL backend is the default and
        uses keys stored in the local host filesystem. The PKCS11
        backend supplies an interface to PKCS11 supported keystore.
```

```
-g, --verbose:
    Optional, displays verbose information.  No additional
    arguments are required

-l, --license:
    Optional, displays program license information.  No additional
    arguments are required

-v, --version:
    Optional, displays the version of the tool.  No additional
    arguments are required

-h, --help:
    Optional, displays usage information.  No additional
    arguments are required
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `cst`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0	if the executable succeeded, or
>0	otherwise.

Cautions:

None.

Pre-Conditions/Assumptions:

Input CSF must be present at specified path.

Certificates must be in a directory called `crts`.

Keys must be in a directory called `keys`. The `keys` directory must be located at the same level as the `crts` directory.

Filenames for the keys and certificates must use the following convention <filename>_<type>.pem or <filename>_<type>.der

where: <filename> is the root of the key/certificate filename

<type> is key for keys and crt for certificates.

Example: keys/SRK1_sha256_2048_65537_v3_ca_key.der must have a corresponding certificate crts/SRK1_sha256_2048_65537_v3_ca_crt.der

Post Conditions:

None.

Examples:

1. To generate out.bin file from input example.csf, use

```
cst -o out.bin -i example.csf
```

2. To print program license information, use

```
cst --license
```

3. To print usage information, use

```
cst --help
```

4. To generate out.bin from input hab4.csf and public key certificate to encrypt symmetric key(s)

```
cst -o out.bin --cert dek_protection_crt.pem -i example.csf
```

4.2 SRK Tool

The SRK tool is used to generate super root key table data and its hash (for efuses).

4.2.1 SRK Tool Usage for HAB4

This section describes usage of SRK tool for HAB4.

Usage:

```
srktool --hab_ver <version> --table <tablefile> --efuses <efusefile>

        --digest <digestalg> --certs <srk>,%<srk>,...

        [--fuse_format <format>] [--license]
```

Description:

```
  -h, --hab_ver <version>:
      HAB Version - set to 4 for HAB4 SRK table generation

  -t, --table <tablefile>:
      Filename for output SRK table binary file

  -e, --efuses <efusefile>:
      Filename for the output SRK efuse binary file containing the SRK
table
      hash

  -d, --digest <digestalg>:
      Message Digest algorithm. Only sha256 is supported

  -c, --certs <srk1>,<srk2>,...,<srk4>:
      X.509v3 certificate filenames.
        - Certificates may be either DER or PEM encoded format
        - Certificate filenames must be separated by a ','with no spaces
        - A maximum of 4 certificate filenames may be provided. Additional
          certificate names are ignored
        - Placing a % in front of a filename replaces the public
          key data in the SRK table with a corresponding hash digest

  -f, --fuse_format <format>:
      Optional, Data format of the SRK efuse binary file.  The
      format may be selected by setting <format> to either:
        - 0: 8 fuses per word, ex: 00 00 00 0a 00 00 00 01 ...
        - 1 (default): 32 fuses per word, ex: 0a 01 ff 8e

   -l, --license:
      Optional, displays program license information.  No additional
      arguments are required.

   -v, --version:
      Optional, displays the version of the tool.  No additional
      arguments are required.

   -b, --verbose:
      Optional, displays a verbose output.
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `srktool`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0	if the executable succeeded, or
>0	otherwise.

NOTE

Using the % prefix in the -c option does not change the SRL fuse pattern generated but does reduce the overall size of the SRK Table. However, an SRK prefixed with % cannot be selected in the Install SRK command using that SRK Table.

Cautions:

None.

Pre-Conditions/Assumptions:

None.

Post Conditions:

None.

Examples:

1. To generate an SRK table and corresponding fuse pattern from 3 certificates
   - using PEM encoded certificate files
   - using full key for first two certificates and hash digest for the third
   - using the default 32 fuse bits per word for the efuse file

```
srktool --hab_ver 4 --table table.bin --efuses fuses.bin \
                   --digest sha256 \
                   --certs srk1_crt.pem,srk2_crt.pem,srk3_crt.pem
```

2. To generate an alternative SRK Table with the same fuse pattern as in example 1 and with SRK3 selectable:

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
                   --digest sha256 \
                   --certs srk1_crt.pem,srk2_crt.pem,srk3_crt.pem
```

3. To generate an SRK table and corresponding fuse pattern from 2 certificates
   - using DER encoded certificate files
   - using the optional 8 fuse bits per word for the efuse file

```
srktool --hab_ver 4 --table table.bin  --efuses fuses.bin \
                   --digest sha256 \
                   --certs srk1_crt.der,srk2_crt.der\
                        --fuse_format 1
```

4.2.2 SRK Tool Usage for AHAB

This section describes usage of SRK tool for AHAB.

Usage:

```
srktool --ahab_ver --table <tablefile> --efuses <efusefile>
```

```
                --sign_digest <digestalg> --certs <srk>,<srk>,...
                [--fuse_format <format>] [--license]
```

Description:

```
srktool --ahab_ver --table <tablefile> --efuses <efusefile>
        --sign_digest <digestalg> --certs <srk>,<srk>,...
        [--fuse_format <format>] [--license]

  -a, --ahab_ver:
      AHAB Version - set for AHAB SRK table generation

  -t, --table <tablefile>:
      Filename for output SRK table binary file

  -e, --efuses <efusefile>:
      Filename for the output SRK efuse binary file containing the SRK
table
      hash

  -d, --digest <digestalg>:
      Message Digest algorithm.
          - sha512 (default): Supported in 8/8x devices
          - sha256: Supported in 8ULP

  -s, --sign_digest <digestalg>:
      Signature Digest algorithm. Either sha256, sha384 or sha512

  -c, --certs <srk1>,<srk2>,...,<srk4>:
      X.509v3 certificate filenames.
        - Certificates may be either DER or PEM encoded format
        - Certificate filenames must be separated by a ','with no spaces
        - A maximum of 4 certificate filenames may be provided. Additional
          certificate names are ignored
  -f, --fuse_format <format>:
      Optional, Data format of the SRK efuse binary file.  The
      format may be selected by setting <format> to either:
        - 0: 8 fuses per word, ex: 00 00 00 0a 00 00 00 01 ...
        - 1 (default): 32 fuses per word, ex: 0a 01 ff 8e

   -l, --license:
      Optional, displays program license information.  No additional
      arguments are required.
```

```
-v, --version:
   Optional, displays the version of the tool.  No additional
   arguments are required.

-b, --verbose:
   Optional, displays a verbose output.
```

Command line arguments that specify a file or directory can contain spaces if they are quoted. File names with leading and trailing spaces are not supported.

If an error occurs during the operation of `srktool`, an error message will be printed to the standard output stream and the executable will exit with a non-zero status.

Exit Status:

0	if the executable succeeded, or
>0	otherwise.

Cautions:

None.

Pre-Conditions/Assumptions:

None.

Post Conditions:

None.

Examples:

1. To generate an SRK table and corresponding fuse pattern
    - using PEM encoded certificate files
    - using the default 32 fuse bits per word for the efuse file

```
srktool --ahab_ver --table table.bin --efuses fuses.bin \
                 --sign_digest sha384 \
                 --certs
      srk1_crt.pem,srk2_crt.pem,srk3_crt.pem,srk4_crt.pem
```

2. To generate an SRK table and corresponding fuse pattern
    - using DER encoded certificate files
    - using the optional 8 fuse bits per word for the efuse file

```
srktool --ahab_ver --table table.bin --efuses fuses.bin \
            --sign_digest sha256 \
            --certs
srk1_crt.der,srk2_crt.der,srk3_crt.der,srk4_crt.der \
                 --fuse_format 1
```

5 CSF Description Language

This section describes the CSF description language. A CSF description file is written in the CSF description language, which is parsed and processed by the CST application and generates a binary file containing the CSF commands (valid only for HAB), certificates, and signatures, which are interpreted by the secure element on the end-product device.

5.1 Overview

The following are the general properties of CSF description files:

- The CSF description file is a text file containing statements, one per line.
- A backslash character '\' at the end of a line (ignoring white space or comments) continues the statement to the next line.
- Blank lines are ignored.
- Comments beginning with the # character on any line are ignored.
- Multiple white space characters are equivalent to a single space. Except where noted, keywords and parameters are separated by white space. White space at the beginning or end of a line is ignored.
- Except for file names, all keywords and parameters are case-insensitive.
- All certificate file parameters are relative to current folder from where CST application is being executed.
- All byte parameters are specified as integers in the range 0...255. They can be specified in hexadecimal or decimal.
- All parameters that specify a file name must be double quoted. A quoted file name can contain spaces. The following file names are not supported:
  - File name with leading or trailing spaces.
  - File name that contains a double quote (") as part of the file name.
- Ordering of commands within the CSF description is significant only to the following extent:
  - The Header command must precede any other command. Valid for HAB and AHAB. The next statements are valid only for HAB.
  - The Install SRK command must precede the Install CSFK command.
  - The Install CSFK must precede the Authenticate CSF command.

— Install SRK, Install CSFK and Authenticate CSF commands must appear exactly once in a CSF description file.

— A verification index in an Authenticate Data command must appear as the target index in a previous Install Key command.

— Commands in the binary CSF follow the order in which they appear in the CSF description.

5.2 CSF Commands

This section describes each CSF command in detail.

5.2.1 Header

The Header command contains data used in the CSF header as well as default values used by the CST for other commands throughout the remaining CSF.

There must be exactly one Header command and it must appear first in the CSF.

Table 3 below lists the Header command arguments.

Table 3. Header arguments

Argument name	Description	Valid values	HAB4	AHAB
Target	Targeted secure element. If not specified, HAB will be assumed.	HAB, AHAB	O	M
Version	Version of HAB	4.x, where x=0,1,...	M	M
Mode	Mode of CST execution (to be specified only for HSM handling)	HSM	O	O
Hash Algorithm	Default hash algorithm	SHA256	O	X
Engine	Default engine.	ANY, SAHARA, RTIC, DCP, CAAM and SW	O	X
Engine Configuration	Default engine configuration	See Table 4	O	X
Certificate Format	Default certificate format)	X509	O	X
Signature Format	Default signature format	PKCS1, CMS	O	X

M = mandatory, O = optional, D = use default from Header if absent and X = not present

Table 4 below lists valid engine configuration values for each engine type

Table 4. Valid Engine configuration values

Engine name	Valid engine configuration values
ANY	0
SAHARA	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 DSC BE816 DSC BE832
DCP	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP32 OUT SWAP8 OUT SWAP32
CAAM	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 OUT SWAP8 OUT SWAP16 DSC SWAP8 DSC SWAP16
RTIC	One or more of these, separated by '|': 0 IN SWAP8 IN SWAP16 OUT SWAP8 KEEP
SW	0

5.2.1.1 Header Examples

```
[Header]
    Version = 4.1 # HAB4 example
    Hash Algorithm = SHA256
    Engine = Any
```

```
        Engine Configuration = 0
        Certificate Format = X509
        Signature Format = CMS

    [Header]
        Target = AHAB # AHAB example
        Version = 1.0
```

5.2.2 Install SRK

The Install SRK command authenticates and installs the root public key for use in subsequent Install CSFK (HAB only) or Install Key (HAB4 only) commands.

HAB or AHAB authenticates the SRK using the SRK hash (SRK_HASH) fuses. HAB4 or AHAB allows revocation of individual keys within the SRK table using the SRK revocation (SRK_REVOKE) fuses.

HAB installs the SRK in slot 0 of its internal public key store.

There must be exactly one Install SRK command in a CSF, and it must occur before the Install CSFK (HAB only) command. Table 5 lists the Install SRK command arguments.

Table 5. Install SRK arguments

Argument name	Description	Valid values	HAB4	AHAB
File	SRK table	Valid file path	M	M
Source Index	SRK index within SRK table. Installation fails if the SRK revocation fuse with this index is burned.	0..3	M	M
Source	SRK certificate corresponding to the specified SRK index	Valid file path	X	M
Source Set	Origin of the SRK table	NXP, OEM (NXP is reserved for NXP deliverables)	X	M
Revocations	Revoked SRKs (Note that this field may trigger a fusing procedure)	4-bit bitmask	X	M
Hash Algorithm	SRK table hash algorithm	SHA256	D	X

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.2.1 Install SRK Examples

```
[Install SRK] # HAB4 example
    File = "../crts/srk_table.bin"
    Source Index = 0
    Hash Algorithm = sha256

[Install SRK] # AHAB example
    File = "../crts/srk_table.bin"
    Source = "../crts/srk3_crt.pem"
    Source index = 2
    Source set = OEM
    Revocations = 0x0
```

5.2.3 Install CSFK (HAB only)

The Install CSFK command authenticates and installs a public key for use in subsequent Authenticate CSF commands.

HAB authenticates the CSFK from the CSFK certificate using the SRK.

HAB installs the CSFK in slot 1 of its internal public key store.

There must be exactly one Install CSFK command in a CSF, and it must occur before the Authenticate CSF command. Table 6 lists the Install CSFK command arguments.

Table 6. Install CSFK arguments

Argument name	Description	Valid values	HAB4
File	CSFK certificate	Valid file path	M
Certificate Format	CSFK certificate format	X509	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.3.1 Install CSFK Examples

```
[Install CSFK] # HAB4 example
    File = "../crts/csf.pem"
    Certificate Format = X509
```

5.2.4 Install NOCAK (HAB4 only)

The Install NOCAK command authenticates and installs a public key for use with the fast authentication mechanism (HAB 4.1.2 and later only). With this mechanism, one key is used for all signatures.

HAB installs the no-CA key in slot 1 of its internal public key store.

There must be exactly one Install NOCAK command in a CSF, and it must occur before the Authenticate CSF command and there must be no Install Key commands. Table 7 lists the install NOCAK command arguments.

Table 7. Install CSFK arguments

Argument name	Description	Valid values	HAB4
File	CSFK certificate	Valid file path	M
Certificate Format	CSFK certificate format	X509	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.4.1 Install NOCAK Examples

```
[Install NOCAK] # HAB4 example
    File = "../crts/csf.pem"
    Certificate Format = X509
```

5.2.5 Authenticate CSF (HAB only)

The Authenticate CSF command authenticates the CSF from which it is executed.

HAB authenticates the CSF using the CSFK public key, from a digital signature generated automatically by the CST.

There must be exactly one Authenticate CSF command in a CSF file, and it must occur after the Install CSFK command. Most other CSF commands are allowed only after the Authenticate CSF command. Table 8 lists the Authenticate CSF command arguments.

Table 8. **Authenticate CSF arguments**

Argument name	Description	Valid values	HAB4
Engine	CSF signature hash engine	ANY, SAHARA, RTIC, DCP, CAAM and SW	D
Engine Configuration	Configuration flags for the hash engine. Note that the hash is computed over an internal RAM copy of the CSF.	see Table 4	D
Signature Format	CSF signature format	CMS	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.5.1 Authenticate CSF Examples

```
[Authenticate CSF] # HAB4 example using all default arguments

[Authenticate CSF] # HAB4 example
    Engine = DCP
    Engine Configuration = 0
    Signature Format = CMS
```

5.2.6 Install Key (HAB only)

The Install Key command authenticates and installs a public key for use in subsequent Install Key or Authenticate Data commands.

HAB authenticates a public key from a public key certificate using a previously installed verifying key and a hash of the public key certificate.

HAB installs the authenticated public key in an internal public key store with a zero-based array of key slots.

The CSF author is responsible for managing the key slots in the internal public key store to establish the desired public key hierarchy and determine the keys used in authentication operations. Overwriting occupied key slots is not allowed, although a repeat command to re-install the same public key occupying the target slot will be skipped and not generate an error.

Multiple Install Key commands are allowed in a CSF. An Install Key command must precede any command which uses the installed key, and all Install Key commands must come after the Authenticate CSF command. Table 9 lists the Install Key command arguments.

Table 9. Install Key arguments

Argument name	Description	Valid values	HAB4
File	Public key certificate	Valid file path	M
Verification Index	Verification key index in key store.	0, 2, ..., 4 CSFK not supported	M
Target Index	Target key index in key store.	2, ..., 4 SRK, CSFK slots reserved.	M
Certificate Format	Public key certificate format.	X509	D
Hash Algorithm	Hash algorithm for certificate binding. If present, a hash of the certificate specified in the File argument is included in the command to prevent installation from other sharing the same verification key.	SHA256	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.6.1 Install Key Examples

```
[Install Key] # HAB4 example
    Key = "../crts/imgk.pem"
    Verification Index = 0
    Target Index = 2
    Certificate Format = X509
```

5.2.7 Authenticate Data

The Authenticate Data command verifies the authenticity of pre-loaded data in memory. The data may include executable SW instructions and may be spread across multiple non-contiguous address ranges drawn from multiple object files.

HAB authenticates the pre-loaded data using a previously installed public key from a digital signature generated automatically by the CST.

The security configuration is taken from the Header command. Table 10 lists the Authenticate Data command arguments.

Table 10. Authenticate Data arguments

Argument name	Description	Valid values	HAB4	AHAB
Blocks	List of one or more data blocks. Each block is specified by four parameters: • source file (must be ***binary***), • starting load address in memory • starting offset within the source file • length (in bytes) **Note:** Please be aware that a maximum of 8 statement blocks are allowed.	*file address offset length* with *file*: valid pathname *address*: 32-bit unsigned integer *offset*: 0, ..., size of *file* *length*: 0, ..., size of *file* - *offset* Block parameters separated by spaces. Multiple blocks separated by commas.	M	X
Verification Index	Verification key index in key store.	2, ..., 4 (HAB4) SRK, CSFK not supported NOTE: For HAB4 Fast Authentication, this must be 0	M	X
Engine	Data signature hash engine.	ANY, SAHARA, RTIC, DCP, CAAM and SW	D	X
Engine Configuration	Configuration flags for the engine.	See Table 4	D	X
Signature Format	Data signature format	CMS	D	X
File	Binary to be signed	Valid file path	X	M
Offsets	List of 2 offsets. Meaningful information for CST into the binary to be signed (this information is printed out by mkimage)	*container_header_offset* *signature_block_offset* Offset parameters separated by spaces Unsigned integers	X	M

Signature	Binary file containing the signature of the container. This field has been added for the HSM support.	Valid file path	X	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.7.1 Authenticate Data Examples

```
[Authenticate Data] # HAB4 example
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
               0xf801000 0x0 0x1000 "xyz.bin"
    Verification Index = 2
    Engine = DCP
    Engine Configuration = 0
    Signature Format = CMS

[Authenticate Data] # AHAB example
    File = "flash.bin"
    Offsets = 0x400 0x610
```

5.2.8 Install Secret Key

This command is applicable from HAB 4.1 onwards and only on processors which include CAAM and SNVS. Each instance of this command generates a CSF command to install a secret key in CAAM's secret key store. A key blob as described in Section 1.1.2 is unwrapped using a master key encryption key (KEK) supplied by SNVS. A random key is generated and protected by the CST back end and encrypted using a public key passed with --cert command line option to CST and saved in a file under the name passed in the Key argument. This file is intended for later use by provisioning software to create the blob. Table 11 lists the Install Secret Key command arguments. Each execution of the CST will generate a different secret key, overwriting any previous secret key in the given file.

Table 11. Install Secret Key arguments

Argument name	Description	Valid values	HAB4	AHAB
Key	Output filename for CST to create the encrypted data encryption key	Valid pathname	M	M
Key length	Key length in bits	128, 192 and 256	M	M

Verification Index	Master KEK index	0 or 1: OTPMK from fuses 2: ZMK from SNVS 3: CMK from SNVS	D	X
Target Index	Target secret key store index	0, 1, 2 or 3 of secret key store	M	X
Blob Address	Absolute memory address where blob will be loaded	Internal or external DDR address	M	X
Key Identifier	Identifier that must match the value provided during the blob generation.	32-bit value (default value is 0)	X	O
Image Indexes	List of images that will be encrypted.	Mask of bits (by default, all images are encrypted)	X	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.8.1 Install Secret Key Examples

```
[Install Secret Key] # HAB4 - Example using OTPMK (Default)
    Key = "data_encryption.key"
    Target Index = 0 /* Secret key store index */
    Blob Address = 0x0090a000 /* internal ram address */

[Install Secret Key] # HAB4 - Example using ZMK
    Key = "data_encryption.key"
    Verification Index = 2 /* ZMK */
    Target Index = 0 /* Secret key store index */
    Blob Address = 0x0090a000 /* internal ram address */

[Install Secret Key] # AHAB - Example using default values
    Key = "data_encryption.key"
    Key Length = 128

[Install Secret Key] # AHAB - Example using optional values
    Key = "data_encryption.key"
    Key Length = 128
    Key Identifier = 0x4a534d21
    Image Indexes = 0xFFFFFFFE /* Image index 0 not encrypted */
```

5.2.9 Decrypt Data (HAB only)

This command is applicable from HAB4.1 onwards. Each instance generates a CSF command to decrypt and authenticate a list of code/data blocks using secret key stored in the secret key store.

CST will generate a corresponding AUT_DAT command. CST will encrypt the data blocks in-place in the given files using a secret key and generate MAC data which is appended to the CSF. Table 12 lists the Decrypt Data command arguments. The secret key index must have been the target key index in a preceding Install Secret Key command. The same secret key must never be used more than once. The secret key used is removed from the secret key store by the Decrypt Data command. A separate Install Secret Key command (which generates a fresh secret key) is required for another Decrypt Data command.

Table 12. Decrypt Data arguments

Argument name	Description	Valid values	HAB (> 4.0)
Blocks	List of one or more data blocks. Each block is specified by four parameters: • source file (must be ***binary***), • starting load address in memory • starting offset within the source file • length (in bytes)	*file address offset length* with *file*: valid pathname *address*: 32-bit unsigned integer *offset*: 0, ..., size of *file* *length*: 0, ..., size of *file* - *offset* Block parameters separated by spaces. Multiple blocks separated by commas.	M
Verification Index	Secret key index in Secret key store	0, 1, 2 or 3 from secret key store	M
Engine	MAC engine	CAAM (Default)	D
Engine Configuration	Configuration flags for the engine.	See Table 4 Default from header command	D
MAC Bytes	Size of MAC in bytes.	Even value between 4 and 16 (Default 16)	D

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.9.1 Decrypt Data Examples

```
[Decrypt Data]
    Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
    0xf8010000 0x0 0x1000 "xyz.bin"
    Verification Index = 0


[Decrypt Data]
Blocks = 0xf8000000 0x0 0x10000 "flash.bin", \
                0xf8010000 0x0 0x1000 "xyz.bin", \
                0xf8012000 0x2000 0x4000 "xyz.bin", \
                0xf8018000 0x8000 0x1000 "xyz.bin"
    Verification Index = 3
    Engine = CAAM
    Engine Configuration = 0
```

5.2.10 NOP (HAB only)

The NOP command has no effect.

Multiple NOP commands may appear in a CSF after the Authenticate CSF command. For HAB4, NOP commands may also appear between the Header and Authenticate CSF commands.

The NOP command has no arguments.

5.2.10.1 NOP Example

```
[NOP]
```

5.2.11 Set Engine (HAB only)

The Set Engine command selects the default engine and engine configuration for a given algorithm.

Some CSF commands allow the CSF author to select the engine used for an algorithm by specifying an argument other than ANY. However, if the engine argument is ANY, then HAB selects the engine to use based on internal criteria. The Set Engine command overrides the HAB internal criteria and selects the engine and configuration to use when ANY is specified.

Some algorithm types do not have an associated engine argument in the CSF commands (e.g. the signature algorithm in Authenticate Data commands). By default, HAB selects the engine to use for such algorithms based on internal criteria. The Set Engine command overrides the HAB internal criteria in such cases as well.

Multiple Set Engine commands may appear anywhere in a CSF after the Header command. Subsequent commands use the engine selected by the most recent Set Engine command. Table 21 lists the Set Engine command arguments.

Table 21. Set Engine arguments

Argument name	Description	Valid values	HAB4
Hash Algorithm	Hash algorithm	SHA256	M

Engine	Engine Use ANY to restore the HAB internal criteria.	ANY, SAHARA, RTIC, DCP, CAAM and SW	M
Engine Configuration	Configuration flags for the engine.	See Table 4	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.11.1 Set Engine Example

```
[Set Engine]
    Hash Algorithm = SHA256
    Engine = DCP
    Engine Configuration = 0
```

5.2.12 Init (HAB only)

The Init command initializes specified engine features when exiting the internal boot ROM.

Multiple Init commands may appear after the Authenticate CSF command. A feature will be initialized if specified in one or more Init commands. Table 22 lists the Init command arguments.

Table 22. Init arguments

Argument name	Description	Valid values	HAB4
Engine	Engine to initialize	SRTC	M
Features	Comma-separated list of features to initialize	RNG (CAAM). See Table 24	O

M = mandatory, O = optional, D = use default from Header if absent and X = not present

NOTE

Please refer to AN4581 regarding Init RNG feature before using it.

5.2.12.1 Init Example

```
[Init]
    Engine = SRTC

[Init]
    Engine = CAAM
    Features = RNG
```

5.2.13 Unlock (HAB only)

The Unlock command prevents specified engine features from being locked when exiting the internal boot ROM.

Multiple Unlock commands may appear after the Authenticate CSF command. A feature will be unlocked if specified in one or more Unlock commands. Table 23 lists the Unlock command arguments.

Table 23. Unlock arguments

Argument name	Description	Valid values	HAB4
Engine	Engine to unlock	SRTC, CAAM, SNVS and OCOTP	M
Features	Comma-separated list of features to unlock	See Table 24	O
UID	Device specific 64-bit UID Required to unlock certain features, must be absent for others (see Table 24).	*U0,U1,... U7* with *Ui*=0..255 UID bytes separated by commas	M/X

M = mandatory, O = optional, D = use default from Header if absent and X = not present

Table 24 shows valid Features values available in Init/Unlock commands for each Engine argument.

Table 24. Valid feature values

Engine	Features	UID	Init/Unlock command effect
SRTC		X	The Init command clears any failure status flags and clears the low-power counters and timers if the SRTC is in Init state. The Unlock command prevents the secure timer and monotonic counter being locked if the SRTC is in Valid state
CAAM	MID	X	Leaves Job Ring and DECO master ID registers unlocked.
	RNG	X	Leaves RNG state handle 0 uninstantiated, does not generate descriptor keys, does not set the AES DPA mask, and does not block state handle 0 test instantiation.

	MFG	X	Keep manufacturing protection private key in CAAM internal memory.
SNVS	LP SWR	X	Leaves LP SW reset unlocked.
	ZMK WRITE	X	Leaves Zeroisable Master Key write unlocked.
OCOTP	FIELD RETURN	M	Leave Field Return activation unlocked.
	SRK REVOKE	X	Leave SRK revocation unlocked.
	SCS	M	Leave SCS register unlocked.
	JTAG	M	Unlock JTAG using SCS HAB_JDE bit.

M = mandatory, O = optional, D = use default from Header if absent and X = not present

5.2.13.1 Unlock Examples

```
[Unlock]
    Engine = SRTC

[Unlock]
    Engine = CAAM
    Features = RNG

[Unlock]
    Engine = OCOTP
    Features = JTAG, SRK REVOKE
    UID = 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0xcd, 0xef
```

5.2.14 Install Certificate (AHAB only)

The Install Certificate command is optional.

The Install Certificate command converts a public key into the NXP format.

AHAB authenticates a Certificate from a previously installed verifying SRK and a hash of the public key certificate.

There must be up to one Install Certificate command in a CSF. Table 9 lists the Install Certificate command arguments.

Table 9. Install Certificate arguments

Argument name	Description	Valid values	AHAB

File	Public key certificate	Valid file path	M
Permissions	Please refer to the AHAB architecture specification for setting this value correctly	8-bit bitmask	M
Signature	Binary file containing the signature of the NXP-format public key certificate. This field has been added for the HSM support.	Valid file path	O

M = mandatory, O = optional

5.2.14.1 Install Certificate Examples

```
[Install Certificate]
    File = "../crts/sgk1_crt.pem"
    Permissions = 0x1
```

5.3 CSF Examples

This section provides some examples for HAB4 and AHAB CSF.

5.3.1 HAB4 CSF Example

Figure 18 is an example of a HAB4 CSF description. This example CSF description:

- Defines a version 4 CSF description.
- Overrides default engine ANY with DCP in Authenticate Data command
- Lists three blocks from image for signing.

```
[Header]
    Version = 4.0
    Security Configuration = Open
    Hash Algorithm = sha256
    Engine = DCP
        Engine Configuration = 0
        Certificate Format = X509
        Signature Format = CMS

[Install SRK]
        File = "TBL_1_sha256_tbl.bin"
        Source index = 0

[Install CSFK]
        File = "CSF1_1_pkcs1_pkcs1_sha256_1024_3_v3_usr_crt.bin"

[Authenticate CSF]

[Install Key]
        Verification index = 0
        Target index = 2
        File = "IMG1_1_pkcs1_pkcs1_sha256_1024_3_v3_usr_crt.bin"

# whole line comment

[Authenticate Data]   # part line comment
        Verification index = 2
        Engine = DCP
        Blocks = 0xf8009400 0x400 0x40 "MCUROM-OCRAM-ENG_img.bin", \
                 0xf8009440 0x440 0x40 "MCUROM-OCRAM-ENG_img.bin", \
                 0xf800a000 0x1000 0x8000 "MCUROM-OCRAM-ENG_img.bin"
```

Figure 18. Example HAB4 CSF Description File

5.3.2 HAB4 CSF Fast Authentication Example

Figure 19 is an example of a HAB4 CSF description for fast authentication. This example CSF description:

- Defines a version 4 CSF description.
- Tells HAB to use fast authentication mechanism
- Lists single block from image for signing

```
#Illustrative Command Sequence File Description
[Header]
    Version = 4.1
    Hash Algorithm = sha256
    Engine = ANY
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS

[Install SRK]
    File = "../crts/TBL_1_sha256+tbl.bin"
    Source index = 0

[Install NOCAK]
    File = "../crts/SRK1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]
#whole line comment

[Authenticate Data]    # part line comment
```

```
    Verification index = 0
    Blocks = 0x877fb000 0x000 0x48000 "signed-uboot.bin"
```

Figure 19. Example HAB4 CSF Description File

5.3.3 HAB4 CSF Example for Encrypted Boot

Figure 20 is an example of a HAB version 4.1 CSF description demonstrating on how to use Install Secret Key and Decrypt Data commands. This example CSF description:

- Defines a version 4.1 CSF description.
- Necessary blocks from image for signing.
- Install Secret Key command
- Blocks for encryption by CST and decryption by ROM/HAB

```
[Header]
    Version = 4.1
    Hash Algorithm = SHA256
    Engine Configuration = 0
    Certificate Format = X509
    Signature Format = CMS
    Engine = CAAM
    Engine Configuration = 0

[Install SRK]
    File = "../crts/SRK_1_2_3_4_table.bin"
    Source index = 0

[Install CSFK]
    File = "../crts/CSF1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate CSF]

[Install Key]
    Verification index = 0
    Target index = 2
    File = "../crts/IMG1_1_sha256_4096_65537_v3_usr_crt.der"

[Authenticate data]
    Verification index = 2
    Blocks = 0x27800400 0x400 800 "u-boot-mx6q-arm2_padded.bin"

[Install Secret Key]
    Verification index = 0
    Target index = 0
    Key = "dek.bin"
    Key Length = 128
    Blob address = 0x27831000
```

```
[Decrypt Data]
    Verification index = 0
    Mac Bytes = 16
    Blocks = 0x27800720 0x720 0x2E8E0 "u-boot-mx6q-arm2_padded.bin"
```

Figure 20. Example HAB4 CSF Description File with Decrypt Data Command

5.3.4 AHAB CSF Example

Figure 18 is an example of an AHAB CSF description. This example CSF description:

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk1_crt.pem"
    # Index of SRK in SRK table
    Source index = 0
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x0

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x490
```

Figure 18. Example AHAB CSF Description File

5.3.5 AHAB CSF with Certificate Example

Figure 19 is an example of an AHAB CSF description with the Certificate. This example CSF description:

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk3_crt.pem"
    # Index of SRK in SRK table
    Source index = 2
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x1
[Install Certificate]
    # Public key certificate in PEM or DER format
    File = "…/crts/sgk3_crt.pem"
    Permissions = 0x1

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x710
```

Figure 19. Example AHAB CSF Description File

5.3.6 AHAB CSF Example for encrypted boot

Figure 18 is an example of an AHAB CSF description for encrypted boot. This example CSF description:

```
[Header]
    Target = AHAB
    Version = 1.0

[Install SRK]
    # Output of srktool
    File = "…/crts/srk_table.bin"
    # Public key certificate in PEM or DER format
    Source = "…/crts/srk1_crt.pem"
    # Index of SRK in SRK table
    Source index = 0
    # Origin of SRK table
    Source set = OEM
    # Revoked SRKs
    Revocations = 0x0

[Authenticate Data]
    # Output of mkimage
    File = "flash.bin"
    # Offsets = Container header Signature block (printed out by mkimage)
    Offsets   = 0x400             0x490

[Install Secret Key]
    Key = "data_encryption.key"
    Key Length = 128
    Key Identifier = 0x4a534d21
    # Image index 0 not encrypted
    Image Indexes = 0xFFFFFFFE
```

Figure 18. Example AHAB CSF Description File

Appendix A CST Architecture

The NXP CST is a reference implementation and is sufficient for most use cases. The tool has a Front End supporting the NXP proprietary operations. There are two reference Back End implementations that provide the cryptographic services to the Front End through the Adaption Layer. The first Back End uses the OpenSSL library [1] for performing all cryptographic operations related to digital signature generation and encryption and accesses key material directly in the filesystem. The second Back End uses an OpenSSL engine supporting a PKCS#11 interface. The key material is referenced with token identifiers in the CSF description files. Please refer to the A1.3 section Using Code-Signing Tool with Hardware Security Module for an example using the PKC#11 backend with a provider.

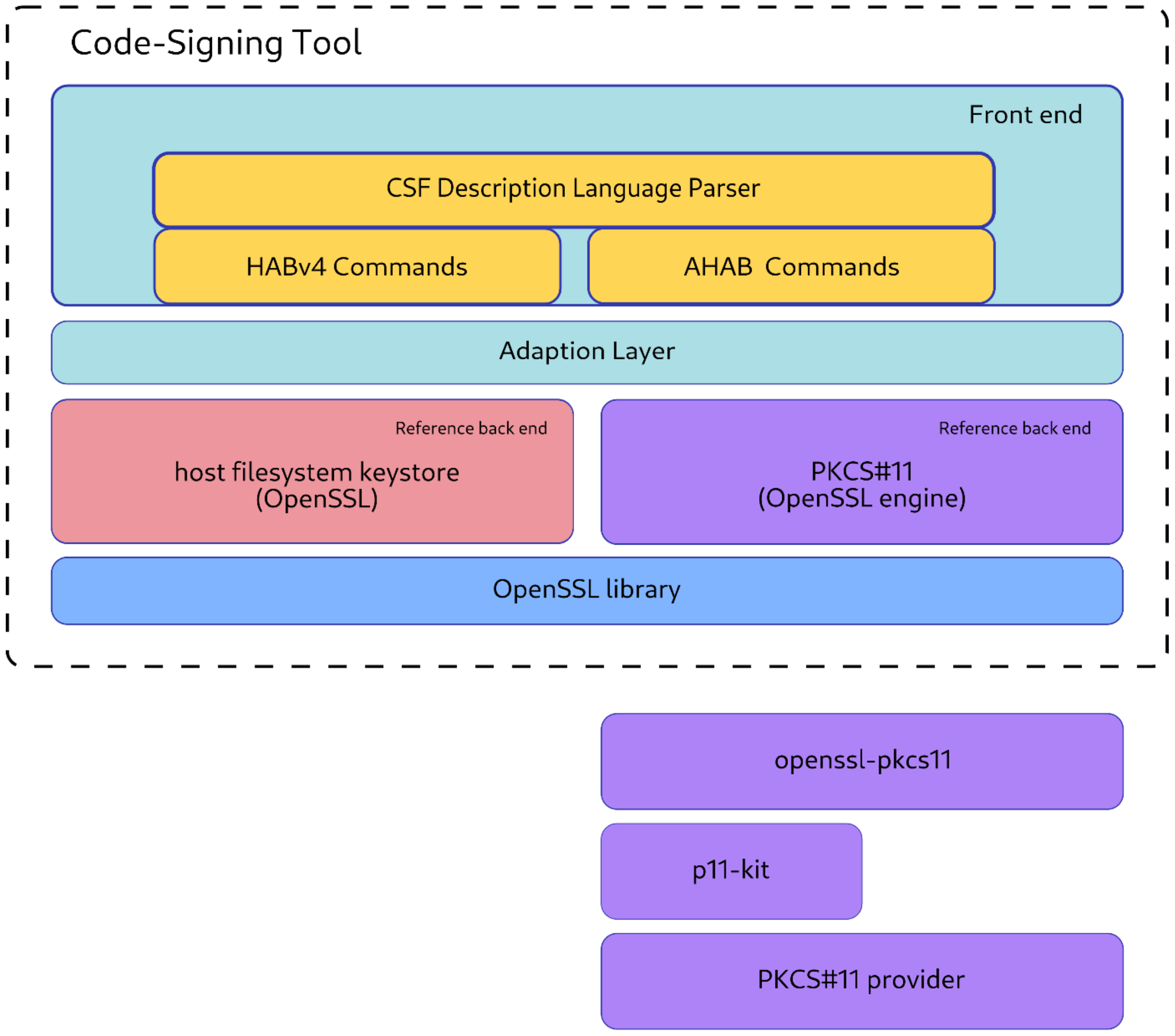

Figure B-1. Overview of Reference CST components

A.1 Customizing the Back End

There may be instances where an alternate back-end interface is required to provide cryptographic services. In such cases it is possible to adapt CST reference implementation to the alternate services. To accommodate this NXP has architected the CST in two parts a Front End and a Back End. The Front End contains all the NXP proprietary operations of the CST with the Back End containing all standard cryptographic operations.

In addition to the CST executables, the package includes the source code. It is located in the package's `code/cst` directory. A Dockerfile is included to create a build environment or as a reference for a build host's dependency requirements.

A.1.1 Back End API

The Back End must implement three API's used by the Front End.

A.1.1.1 gen_sig_data()

```
int32_t gen_sig_data(const char* in_file,
                     const char* cert_file,
                     hash_alg_t hash_alg,
                     sig_fmt_t sig_fmt,
                     uint8_t* sig_buf,
                     size_t *sig_buf_bytes,
                     func_mode_t mode);
```

The CST Front End uses this API to generate signature data.

`in_file`	Input data
`cert_file`	Signer certificate file
`hash_alg`	Hash algorithm
`sig_fmt`	Signature format
`sig_buf`	Signature buffer
`sig_buf_bytes`	Size of signature buffer
`Mode`	Custom mode

A.1.1.2 gen_auth_encrypted_data()

```
int32_t gen_auth_encrypted_data(const char* in_file,
                                const char* out_file,
                                aead_alg_t aead_alg,
                                uint8_t *aad,
                                size_t aad_bytes,
                                uint8_t *nonce,
                                size_t nonce_bytes,
                                uint8_t *mac,
                                size_t mac_bytes,
                                size_t key_bytes,
                                const char* cert_file,
                                const char* key_file,
                                int reuse_dek);
```

The CST Front End uses this generate authenticated encrypted data. It generates an encryption key and uses it to encrypt plaintext input data. Optionally, it can use an input key instead of generating a key.

A generate encryption key can be optionally encrypted using a certificate for secure transport.

`in_file`	Plaintext input data
`out_file`	Output cipher text
`aead_alg`	AES_CCM or AES_CCB
`Aad`	Additional authenticated data
`aad_bytes`	Size of aad
`Nonce`	Nonce bytes to return
`nonce_bytes`	Size of nonce
`Mac`	Output MAC
`mac_bytes`	Size of MAC
`key_bytes`	Size of symmetric key
`cert_file`	Certificate file for DEK encryption
`key_file`	Input key file
`reuse_dek`	Use existing input key

A.1.1.3 read_certificate()

```
X509* read_certificate(const char* reference);
```

Reads X.509 certificate data from the provided certificate file.

`Reference`	Certificate file

- Any new Back End implementation must follow implement these APIs in an equivalent adaptation layer corresponding to the new cryptographic services.
- For reference the source code and header files for the NXP reference implementation.
- To use a new method for public key generation, replace the key generation scripts with the new implementation.

NOTE

Although the Back End may replace OpenSSL for code signing, SA and CA support, the CST Front End still makes use of OpenSSL for some non-code signing operations. This means that when linking library components together to generate a CST executable an OpenSSL library must also be included. CST uses OpenSSL 3.2.0 which is available at [1].

A.2 Front End References to Code Signing Keys

When replacing the CST Back-End it is important to keep in mind that he CST Front End refers to code signing keys and certificates using file names. These are the key filenames that correspond to the RSA public key certificate and private key files generated by the CST PKI scripts. However, filenames may not be the native method for referencing keys in a new replacement Back End service. If this is the case, then the new Adaptation Layer is responsible for converting to and from file name references.

This is also true for Data Encryption Keys that the CST generates for encrypting images.

A.3 Using Code-Signing Tool with Hardware Security Module

This section delivers essential information for users seeking effective utilization of CST with an HSM.

The CST's default backend uses OpenSSL to perform HAB and AHAB signature generation and data encryption. OpenSSL exposes an Engine API which makes it possible to plug in alternative implementations for its native cryptographic operations. Libp11 (openssl-pkcs11) serves as the PKCS#11 engine used by CST to access PKCS#11 enabled HSMs. Encompassing necessary functions for session and token management, certificate handling, signing, and hashing, it acts as a liaison between vendor PKCS#11 modules and the OpenSSL engine API. This engine can interface directly with a PKCS#11 provider or leverage the p11-kit proxy module for system-wide PKCS#11 module access. For additional information, consult [8].

Step 1 - Build and run Docker.hsm image

In the root folder of the package, you'll find a Dockerfile.hsm that facilitates the installation of dependencies, enabling an easy setup for trying out the CST with pkcs11 backend.

```
$ docker build -t cst:hsm --build-arg http_proxy=$http_proxy --build-
arg hostUserName=$USER --build-arg hostUID=$(id -u) --build-arg
hostGID=$(id -g) -f Dockerfile.hsm .
```

Launch a shell that can be used to run the commands as user `softhsm` for the subsequent steps:

```
$ docker run -v $(pwd):/home/$USER/cst -w /home/$USER/cst --rm -it -
-entrypoint bash cst:hsm
```

Step 2 – Creating a Token

The pkcs11-tool command is typically used for interacting with PKCS#11 tokens. To initialize a token, run the following commands:

```
$ pkcs11-tool --module $PKCS11_MODULE_PATH --init-token --init-pin --so-
pin=$SO_PIN --new-pin=$USR_PIN --label="CST-HSM-DEMO" --pin=$USR_PIN --
login

Using slot 0 with a present token (0x0)
Token successfully initialized
User PIN successfully initialized
```

PKCS11_MODULE_PATH should contain the path or name of the PKCS#11 module. In the example, it is set to the path of the p11-kit and can be changed to be set to the path of SoftHSM, depending on the desired PKCS#11 configuration. The token label *CST-HSM-DEMO* will be used later with CST to locate certificates and keys needed to sign images.

Step 3 – Generate a PKI tree

To generate PKI tree for AHAB or HABv4, run the hsm_ahab_pki_tree.sh or hsm_hab4_pki_tree.sh script accordingly in interactive mode. The script will prompt you with a series of questions, and it is expected that you provide answers to guide the generation process.

```
$ cd keys/
$ ./hsm_hab4_pki_tree.sh or ./hsm_ahab_pki_tree.sh
```

Example:

```
$./hsm_hab4_pki_tree.sh -existing-ca n -use-ecc n -kl 2048 -duration 10
-num-srk 4 -srk-ca y
```

You can list the objects stored on the token by using:

```
$ pkcs11-tool --module $PKCS11_MODULE_PATH -l --pin $USR_PIN --list-
objects
```

We anticipate finding both keys and certificates within the token.

Step 4 – Generate SRK table and fuse

Create SRK table and e-fuse files for HAB4 or AHAB, similar to using the default CST backend, by specifying paths to the SRK certificate:

```
$ cd ../crts/
$ ../linux64/bin/srktool ...
```

Example:

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_2_3_4_table.bin -e
SRK_1_2_3_4_fuse.bin -d sha256 -c
./SRK1_sha256_2048_65537_v3_ca_crt.pem,./SRK2_sha256_2048_65537_v3_ca_c
rt.pem,./SRK3_sha256_2048_65537_v3_ca_crt.pem,./SRK4_sha256_2048_65537_
v3_ca_crt.pem -f 1
```

Step 5 – Create Command Sequence File

CST with pkcs11 backend finds keys and certificates in a HSM using PKCS#11 URIs. This URI format is defined in the "PKCS #11 URI Scheme" specification, which is described in RFC 7512 [9]. In this example, the CSF instructs CST to access the SRK certificate with the identifier or label *SRK1_sha384_secp384r1_ca* in the PKCS#11 token/module named *CST-HSM-DEMO*, and it's providing the PIN value *12345678* as part of the authentication process.

```
[Install CSFK]
   File = "pkcs11:token=CST-HSM-
DEMO;object=CSF1_1_sha256_2048_65537_v3_usr;type=cert;pin-
value=${USR_PIN}"
```

URI can be broken down as follows:

- **Token: CST-HSM-DEMO**: This specifies the PKCS#11 module or token being used, in this case, it's named CST-HSM-DEMO.
- **Object: CSF1_1_sha256_2048_65537_v3_usr:** Indicates the specific object within the token. In this example, it's referring to an object with the identifier or label CSF1_1_sha256_2048_65537_v3_usr.
- **Type: cert:** Specifies that the object is a certificate.
- **Pin-value:** This is an optional parameter, representing the PIN needed to access the specified object.

Step 6 – Sign using pkcs11 back-end

The final step is to invoke CST with the **-b** switch, specifying the pkcs11 backend.

```
$ ../linux64/bin/cst -b pkcs11 -i example.csf -o signed-example.bin
CSF Processed successfully and signed data available in signed-
example.bin
```