i.MX Applications Processor Trust Architecture
AMF-IND-T0291

Rod Ziolkowski
Platform Security Architect
MCU Systems and Architecture Team
Introduction

i.MX-based products
• Rich, mobile, end-user, connected platforms
• Increasingly valuable assets: end-user data, licensed content, access credentials, intellectual property
• Increasingly threatened: malware, hacking, misuse

i.MX Trust Architecture
• Protects assets of multiple stakeholders
• Guards against sophisticated attacks
• Assures software measures
Agenda

- Introduction
- Why a Trust Architecture?
- Trust Architecture Features
- Trusted Architecture Deployment
- High Assurance Boot
  - Code Signing Tool
  - Manufacturing Tool
- Summary
Why a Trust Architecture?
Background

• i.MX product characteristics
  - Client / end-user (not server or fabric)
  - Mobile (physically vulnerable)
  - Connected (“internet of things”, remote threats)
  - Rich & open SW (large attack surface)

• Security trends
  - Percentage of breaches involving end-user devices **doubled** year-on-year (Verizon/US Secret Service)
  - Cybercriminals shifting focus from PC to mobile users (Cisco)
  - Major trojans continue to migrate to mobile devices (Security Week)
# Assets & Stakeholders

<table>
<thead>
<tr>
<th>Asset</th>
<th>Stakeholder</th>
<th>Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>Content owner</td>
<td>Piracy</td>
</tr>
<tr>
<td>- Media</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Applications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service access</td>
<td>Service provider</td>
<td>Fraud</td>
</tr>
<tr>
<td>- Network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Enterprise</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intellectual property</td>
<td>Manufacturer</td>
<td>Espionage</td>
</tr>
<tr>
<td>- Owned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Licensed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Personal data</td>
<td>End user</td>
<td>Privacy breach</td>
</tr>
<tr>
<td>- Identification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- Connections</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Threats

• Malware
  – Rootkits, trojans, viruses, worms, keyloggers, bots,…
  – Risk enhanced by rich & open OS
  – Countermeasures: trusted execution, high assurance boot

• Hacking
  – Reverse engineering, brute force
  – Countermeasures: secure storage, secure debug, encryption

• Physical attack
  – Bus snooping, glitching,
  – Countermeasures: secure storage, tamper detection
i.MX Trust Architecture Features & Deployment
i.MX Trust Architecture Features

**Trusted Execution**
- Isolates execution of critical SW from possible malware
- TrustZone Secure & Normal Worlds (processor modes)
- Hardware firewalls between CPU & DMA masters and memory & peripherals

**High Assurance Boot**
- Authenticated boot: prevents unauthorized SW execution
- Encrypted boot: protects SW confidentiality
- Digital signature checks embedded in on-chip boot ROM
- Run every time processor is reset

**HW Cryptographic Accelerators**
- i.MX family dependent
- Symmetric: AES-128, AES-256, 3DES, ARC4
- Message Digest & HMAC: SHA-1, SHA-256, MD-5
Secure Storage
- Protects data confidentiality and integrity
- Off-chip: cryptographic protection including device binding
- On-chip: self-clearing Secure RAM
- HW-only keys: no SW access

HW Random Number Generation
- Ensures strong keys and protects against protocol replay
- On-chip entropy generation
- Cryptographically secure deterministic RNG

Secure Clock
- Provides reliable time source
- On-chip, separately-powered real-time clock
- Protection from SW tampering
i.MX Trust Architecture Features (continued)

Secure Debug:
- Protects against HW debug (JTAG) exploitation for:
  - Security circumvention
  - Reverse engineering
- Three security levels + complete JTAG disable

Tamper Detection
- Protects against run-time tampering
- Monitoring of various alarm sources
  - Debug activation
  - External alarm (e.g. cover seal)
  - SW integrity checks
  - SW alarm flags
- HW and SW tamper response
- Support varies by i.MX family
i.MX Trust Architecture – Overview

- **Tamper Detect**
  - GPIO alarm
  - SW alarm
  - Debug alarm

- **Secure RAM**
  - HW Firewall
  - Erase

- **Secure Clock**
  - HW Firewall

- **Peripheral Slave**
  - HW Firewall

- **Accelerator (Cipher, Hash, RNG)**
  - HW Firewall

- **Electrical Fuses (keys, security levels)**

- **Secure Debug**
  - JTAG

- **Secure Clock**
  - Debug alarm

- **Internal Memory**
  - External Memory
  - HW Firewall

- **ARM CPU**
  - DMA Master

- **ROM (High Assurance Boot)**

- **HW Firewall**
# i.MX Trust Architecture Deployment

<table>
<thead>
<tr>
<th>Feature</th>
<th>i.MX 258</th>
<th>i.MX 27L</th>
<th>i.MX 28x</th>
<th>i.MX 35x</th>
<th>i.MX 508</th>
<th>i.MX 51x</th>
<th>i.MX 53x</th>
<th>i.MX 6x²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>High Assurance Boot</td>
<td>V3</td>
<td>V4</td>
<td>V3</td>
<td>V4</td>
<td>V3</td>
<td>V4</td>
<td>V4</td>
<td>V4</td>
</tr>
<tr>
<td>Secure Storage</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hardware RNG</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secure Clock</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Secure Debug</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tamper Detection</td>
<td>✓</td>
<td>✓¹</td>
<td>✓¹</td>
<td></td>
<td>✓¹</td>
<td>✓¹</td>
<td>✓¹</td>
<td>✓¹</td>
</tr>
</tbody>
</table>

1 External Digital Tamper only monitored when main power is supplied
2 Trust architecture is the same across the i.MX6 family with the exception of i.MX6 SL
## HW Comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>i.MX53</th>
<th>i.MX 6 D/Q &amp; D/L</th>
<th>TI OMAP</th>
<th>NVIDIA Tegra</th>
<th>QCOM QSD</th>
<th>MARVELL ARMADA</th>
<th>Samsung Exynos 5</th>
<th>Intel Atom</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Execution</td>
<td>✔️</td>
<td>✔️</td>
<td>M-shield</td>
<td>Limited</td>
<td>Limited</td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>Secure Boot</td>
<td>✔️</td>
<td>✔️ (including encrypted boot)</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>? (16x)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Secure Storage</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td>HW key protection</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Cryptographic Accelerators</td>
<td>Symmetric Hash RNG</td>
<td>Symmetric Hash RNG</td>
<td>Symmetric Asymmetric Hash RNG</td>
<td>?</td>
<td>?</td>
<td>Symmetric Hash</td>
<td>✔️?</td>
<td>✗</td>
</tr>
<tr>
<td>Secure Real Time Clock</td>
<td>✔️</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>HW Firewalls</td>
<td>CSU</td>
<td>CSU</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✗</td>
</tr>
<tr>
<td>Content Protection</td>
<td>✗</td>
<td>HDCP DTCP</td>
<td>OMA HDCP</td>
<td>HDCP?</td>
<td>SecureMSM</td>
<td>?</td>
<td>HDCP?</td>
<td>✗</td>
</tr>
<tr>
<td>Secure Debug</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✗</td>
</tr>
<tr>
<td>Tamper Detection</td>
<td>✔️</td>
<td>✔️</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
i.MX Trust Architecture Features & Deployment
High Assurance Boot – Purpose

High Assurance Boot ensures the boot sequence:
- Uses authentic SW
- Remains confidential (if required)
- Establishes a “known-good” system state

High Assurance Boot protects against:
- Platform re-purposing
- Rootkits and similar unauthorized SW designed to
  - harvest secrets
  - circumvent access controls
- Offline SW reverse engineering (if required)
High Assurance Boot – Operation

- **Code Signing Using Private Key**
  - SW Image
  - Hash
  - Sign (RSA)
  - SW Image + Signature

- **Secured Environment**

- **Authentication Using Public Key**
  - SW Image + Signature
  - Compare
  - Run OS
  - Reload Image

- **Hash**
- **Private Key**
- **Public Key**
- **Fuse SRK**
- **Device Boot**
- **Verify (RSA)**

- **Generated Hash**
- **Generate**
- **Reload**
- **Reference Hash**
High Assurance Boot – Encrypted

Already supported in the Soc. Reference tools planned to enable this feature.
i.MX High Assurance Boot – Enablement & Tools
High Assurance Boot – Tools

Freescale Reference Code Signing Tool (CST):
- Offline process of creating digital signatures
- Signing Keys and signatures generated by device manufacturers
- Supports code signing for: i.MX258, i.MX28, i.MX35x, i.MX508, i.MX51x, i.MX53x and i.MX6x

Manufacturing Tool:
- Platform software provisioning
- One-Time Programmable e-fuse burning

- Latest releases of both tools can be downloaded from: http://www.freescale.com/webapp/sps/site/overview.jsp?code=IMX_DESIGN
Additional Resources

- Code Signing Tool Users Guide – included in CST release
- HAB 4 API Reference Manual – included in CST release
- Code signing for i.MX application notes. Ties in device configuration, code signing, fusing together in a single document:
  - AN4547: Secure Boot on i.MX25, i.MX35 and i.MX51 using HAB Version 3
  - AN4555: Secure Boot with i.MX28 HAB Version 4
  - AN4581: Secure Boot on i.MX50, i.MX53 and i.MX6 Series using HAB Version 4
  - AN4586: Configuring Secure JTAG for the i.MX 6 Series Family of Applications Processors
- Secure boot example included in i.MX6 Linux BSP releases
  - Authentication of u-boot and Linux kernel images
Planned Updates

• Support for i.MX 6 family encrypted boot
  – CST support to generation of encryption keys and the ability to encrypt images
  – Manufacturing tool support to create cryptographic blobs of encryption keys.

• Support for Manufacturing Tool to download HAB events using ROM Serial Download Protocol
  – Useful for debugging secure boot with HAB on SoCs in the Closed (Secure Configuration).
  – Avoids having to connect with JTAG.
    ▪ May not be possible if JTAG is disabled via fuses.
Code Signing for High Assurance Boot
Generic Code-signing Participants

- **Super Root Keys**
- **Code Signing Certificates**
- **Code Signing Authority (SA)**
- **Certificate Authority (CA)**
- **Signature Authority (SA)**
- **Code Signing Certificate**
- **Protects code signing keys**
- **Signs code**
- **Protects SRKs**
- **Certifies code signing keys**
- **Generates CSFs**
- **Requests signatures**

Manufacturers use Code Signing Certificates to sign code, which is then certified by the Signature Authority. The Certificate Authority certifies the code signing keys, and the Super Root Keys are protected by the Signature Authority.
Reference CST Features

• Reference CST supports:
  – CA functionality: key and certificate generation
  – SA functionality: signature generation
  – Freescale specific functions: HAB Command Sequence File (CSF) generation

• Fully self contained application that runs on a Linux PC
  – Cryptographic algorithm support provided by OpenSSL but can be replaced.

• Private keys are pass-phrase protected in an industry standard format (PKCS#8)
Freescale Reference Code Signing Tool

CSF Description File

Product Software

Freescale Reference Code Signing Tool Host

HAB Data

+ 

CSF Commands

Performs the operations of both a Certificate Authority and a Signature Authority

Signed Product Software
Code signing tools - reference
HAB PKI tree – CA Functionality (HAB v4)
Simple Secure Boot Chain on i.MX

Bootloader
CSF

Reset

HAB Library
i.MX Boot ROM
Boot Device Driver

Bootloader
Boot Device Driver

OS
CSF

Subsystems
Security
Device Driver

Boot Stages
First
Second
Third
CSF Commands

- Defines the actions that HAB will perform
  - Install a public key
  - Verify a digital signature over a block of data
  - And others

- CSF commands are executed sequentially

- As long as the required areas are covered by a signature a CSF is valid
  - CSF author is responsible for ensuring all vital area are covered by a signature
Example CSF File

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

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

[Install CSFK]
  File = "../crt/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
  Verification index = 0
  Target index = 2
  File = "../crt/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

# Sign padded u-boot starting at the IVT through to the end with
#   length = 0x2F000 (padded u-boot length) - 0x400 (IVT offset) = 0x2EC00
# This covers the essential parts: IVT, boot data and DCD.

# Blocks have the following definition:
# Image block start address on i.MX, Offset from start of image file, Length of block in bytes, image data file

[Authenticate Data]
  Verification index = 2
  Blocks = 0x7780400 0x400 0x2EC00 "u-boot-pad.bin"
Manufacturing Tool
Manufacturing Tool Features

• Features in the context of secure boot include:
  - Image provisioning to boot device, e.g. NAND Flash, SD/MMC etc.
  - Uses Serial Download Protocol of i.MX boot ROM
  - Support for fuse burning. Examples include:
    ▪ Security configuration
    ▪ Root key hash
    ▪ Root key revocation
    ▪ Secure JTAG response field
    ▪ and various fuse field lock bits
Interprets XML commands and sends them to i.MX to be executed.
Example Manufacturing Tool XML Script

<LIST name="MX6Q Sabre-lite-SPI_NOR" desc="Choose SPI-NOR as media">
  <!--
    boot dip settings for SPI-NOR boot:
    SW26: dip 1, 4, 5, 6 are on. Others are off
    SW28: dip 5 is on. Others are off
  -->
  <CMD type="find" body="Recovery" timeout="180"/>
  <CMD type="boot" body="u-boot-mx6q-sabrelite.bin" >Loading uboot.</CMD>
  <CMD type="load" file="ulimage" address="0x10800000"
    loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Kernel.</CMD>
  <CMD type="load" file="initramfs.cpio.gz.uboot" address="0x10C00000"
    loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE" >Doing Initramfs.</CMD>
  <CMD type="jump" > Jumping to OS image. </CMD>
  <CMD type="find" body="Updater" timeout="180"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_LOCK"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_CFG5"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_RESP0"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_RESP1"/>

  <CMD type="push" body="$ echo 0x87654321 > /sys/fsl_otp/HW_OCOTP_RESP0">Burn SJC_RESP0 field in OTP</CMD>
  <CMD type="push" body="$ echo 0x00edcba9 > /sys/fsl_otp/HW_OCOTP_HSJC_RESP1">Burn SJC_RESP1 field in OTP</CMD>
  <CMD type="push" body="$ echo 0x00000040 > /sys/fsl_otp/HW_OCOTP_LOCK">Burn SJC_RESP lock fuse in OTP</CMD>
  <CMD type="push" body="$ echo 0x00040000 > /sys/fsl_otp/HW_OCOTP_CFG5">Burn JTAG_SMODE = 01 in OTP</CMD>
</LIST>
</UCL>
Summary
Summary and Q&A

i.MX Trust Architecture:
- Protects assets of multiple stakeholders
- Guards against sophisticated attacks
- Assures software measures

You can now:
- Plan how to protect your products using the i.MX Trust Architecture
- Select i.MX security features
- Pursue more in-depth examination of features and tools
In Depth Study: Linux Secure Boot for i.MX6
Code Signing for i.MX6

- Covers the secure boot example that will be included in a future Linux BSP release.
- Already available in Freescale i.MX Linux BSP release
- Following slides cover
  - Generating signing keys with the FSL reference CST
    - Including SRK table generation
    - SRK fuse blowing
  - Signing U-boot
  - Signing the kernel image to extend the secure boot chain
Generating Code Signing Keys and Certs

This script is a part of the code signing tools for Freescale's High Assurance Boot. It generates a basic PKI tree. The PKI tree consists of one or more Super Root Keys (SKK), with each SKK having two subordinate keys:

- a Command Sequence File (CSF) key
- Image key.

Additional keys can be added to the PKI tree but a separate script is available for this. This script assumes openSSL is installed on your system and is included in your search path. Finally, the private keys generated are password protected with the password provided by the file key_pass.txt. The format of the file is the password repeated twice:

my_password
my_password

All private keys in the PKI tree are in PKCS #8 format will be protected by the same password.

Do you want to use an existing CA key (y/n)? n
Enter key length in bits for PKI tree: 2048
Enter PKI tree duration (years): 10
How many Super Root Keys should be generated? 4

Generating CA key and certificate

Generating a 2048 bit RSA private key

writing new private key to 'temp_ca.pem'

Generating SKK key and certificate

Generating RSA private key, 2048 bit long modulus

c is 65537 (0x10001)

Using configuration from ../ca/openSSL.cnf

Check that the request matches the signature
Two files are generated:

- SRK table: contains the SRK table contents which are included in the HAB data.
- SRK fuse file: contains SHA256 result to be burned to fuses
Burning SRK fuses with Mfg tool for MX6

- `hexdump -e '/4 "0x"' -e '/4 "%X"\n"' <fuses filename`
- This provides the fuse value in the correct byte order which is essential.
Example Mfg tool XML script to blow SRK fuses

<!-- boot dip settings for SPI-NOR boot: 
SW26: dip 1, 4, 5, 6 are on. Others are off 
SW28: dip 5 is on. Others are off -->

<CMD type="find" body="Recovery" timeout="180"/>
<CMD type="boot" body="Recovery" file="u-boot-mx6q-sabrelite.bin">Loading uboot.</CMD>
<CMD type="load" file="uImage" address="0x10800000" loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE">Doing Kernel.</CMD>
<CMD type="load" file="initramfs.cpio.gz.uboot" address="0x10C00000" loadSection="OTH" setSection="OTH" HasFlashHeader="FALSE">Doing Initramfs.</CMD>
<CMD type="jump"> Jumping to OS image. </CMD>
<CMD type="find" body="Updater" timeout="180"/>

<!-- ***** Caution - running this xml script with the fuse burning commands uncommented 
***** in the Mfg tool permanently burns fuses. Once completed this operation cannot 
***** be undone! -->

<CMD type="push" body="$ echo 0xE94B1F02 > /sys/fsl_otp/HW_OCOTP_SRK0">Burn Word 0 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0x067E7696 > /sys/fsl_otp/HW_OCOTP_SRK1">Burn Word 1 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0xBB70C24E > /sys/fsl_otp/HW_OCOTP_SRK2">Burn Word 2 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0xD874E6C8 > /sys/fsl_otp/HW_OCOTP_SRK3">Burn Word 3 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0x53D215CC > /sys/fsl_otp/HW_OCOTP_SRK4">Burn Word 4 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0xBE2D3E36 > /sys/fsl_otp/HW_OCOTP_SRK5">Burn Word 5 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0xBB5932AA > /sys/fsl_otp/HW_OCOTP_SRK6">Burn Word 6 of SRK hash field in OTP </CMD>
<CMD type="push" body="$ echo 0x01D69CA0 > /sys/fsl_otp/HW_OCOTP_SRK7">Burn Word 7 of SRK hash field in OTP </CMD>
Fuse Burning Hints and Warnings:

- Need to update XML script to match generated SRK fuse file contents
- Experiment with burning on non-essential first
  - Especially important for boards that do not have a CPU socket!
  - General Purpose fuse field is a good place to start. For example:

  <!-- **** The following is a simple example to burn bit 0 of the GP1 field. The
  **** results can also be verified by the u-boot command:
  **** "md.1 0x021bc600 1"
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_GP1"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_GP2"/>
  <CMD type="push" body="$ echo 0x00000001 > /sys/fsl_otp/HW_OCOTP_GP1">Burn bit0 of GP1 at OTP</CMD>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_GP1"/>
  <CMD type="push" body="$ cat /sys/fsl_otp/HW_OCOTP_GP2"/>

- MX6 does not check SRK hash when sec_config = OPEN
- Do Not blow sec_config field to CLOSED unless absolutely sure!
U-Boot Image on i.MX6 for SD Boot

- Destination
- Image Size
- Plugin Flag

Offset
- 0x0000
- 0x0400

Image Vector Table (IVT)

- Boot Data

Device Config. Data (DCD)

- U-Boot

HAB Data (CSF, Certs, Signatures)

Fill Data

Signed Data

Note: HAB requires that:
- the entire IVT
- the entire DCD
- initial byte of Boot Data
- initial word of U-Boot must to be signed

Offset
- 0x31000
- 0x2F000

Fill Data

HAB Data (CSF)

Entry

DCD

Boot Data

Plugin Flag
U-boot CSF Description file

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

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

[Install CSFK]
File = ".../crts/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
Verification index = 0
Target index = 2
File = ".../crts/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

# Sign padded u-boot starting at the IVT through to the end with
# length = 0x2F000 (padded u-boot length) - 0x400 (IVT offset) = 0x2EC00
# Note: 0x2F000 may be different depending on the size of U-Boot
# This covers the essential parts: IVT, boot data and DCD.
# Blocks have the following definition:
# Image block start address on i.MX, Offset from start of image file, 
# Length of block in bytes, image data file

[Authenticate Data]
Verification index = 2
Blocks = 0x77800400 0x400 0x2EC00 "u-boot-pad.bin"
u-Boot Signing Script

#!/bin/bash
echo "extend u-boot to 0x2F000..."
# Again the 0x2F000 may be different depending on the size of U-Boot.
objcopy -I binary -O binary --pad-to 0x2f000 --gap-fill=0xff u-boot.bin u-boot-pad.bin

echo "generate csf data..."
../linux/cst --o u-boot_csf.bin < u-boot.csf

echo "merge image and csf data..."
cat u-boot-pad.bin u-boot_csf.bin > u-boot-signed.bin

# This step is not strictly necessary - just padding image to a nice size
echo "extend final image to 0x31000..."
objcopy -I binary -O binary --pad-to 0x31000 --gap-fill=0xff u-boot-signed.bin \
u-boot-signed-pad.bin
echo "u-boot-signed-pad.bin is ready"
ulimage Memory Map for i.MX6

Offset
0x00000000
0x00000040
0x003FDEF0
0x003FE000
0x00400000
0x00400000

Header
ulimage Data
Fill Data (Padded to nearest 4kB)
Image Vector Table (IVT)
HAB Data (CSF, Certs, Signatures)
Fill Data

Signed Data

Varies based on ulimage size

Varies based on ulimage size

0x00000000
0x00000040
0x003FDEF0
0x003FE000
0x00400000
0x00400000
ulimage CSF Description file

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

[Install SRK]
File = "/crt5/SRK_1_2_3_4_table.bin"
Source index = 0

[Install CSFK]
File = "/crt5/CSF1_1_sha256_2048_65537_v3_usr_crt.pem"

[Authenticate CSF]

[Install Key]
Verification index = 0
Target index = 2
File = "../crt5/IMG1_1_sha256_2048_65537_v3_usr_crt.pem"

# Sign padded ulimage start at address 0x10800000
# length = 0x3FE0000
# Note: 0x3FE0000 may be different depending on the size of ulimage
# This covers the essential parts: original ulimage and the attached IVT
# Blocks have the following definition:
# Image block start address on i.MX, Offset from start of image file,
# Length of block in bytes, image data file [Authenticate Data]
[Authenticate Data]
Verification index = 2
Blocks = 0x10800000 0x0 0x003FE0000 "ulimage-pad-ivt.bin"
ulmage IVT Generation (genIVT)

#!/usr/bin/perl -w
use strict;
open(my $out, '>:raw', 'ivt.bin') or die "Unable to open: $!";
print $out pack("V", 0x412000D1); # IVT Header
print $out pack("V", 0x10801000); # Jump Location
print $out pack("V", 0x0); # Reserved
print $out pack("V", 0x0); # DCD pointer
print $out pack("V", 0x0); # Boot Data
print $out pack("V", 0x10BFDFE0); # Self Pointer
print $out pack("V", 0x10BFE000); # CSF Pointer
print $out pack("V", 0x0); # Reserved
close($out);
ulimage Signing Script

#!/bin/bash
# Again the 0x3FE000 may be different depending on the size of uImage.
echo "extend uImage to 0x3FDFE0..."
objcopy -I binary -O binary --pad-to 0x3fdfe0 --gap-fill=0xff uImage uImage-pad.bin

echo "generate IVT"
./genIVT
echo "attach IVT..."
cat uImage-pad.bin ivt.bin > uImage-pad-ivt.bin

echo "generate csf data..."
./linux/cst --o uImage_csf.bin < uImage.csf

echo "merge image and csf data..."
cat uImage-pad-ivt.bin uImage_csf.bin > uImage-signed.bin

echo "extend final image to 0x400000..."
objcopy -I binary -O binary --pad-to 0x400000 --gap-fill=0xff uImage-signed.bin \
uImage-signed-pad.bin

• Provision ulimage-signed-pad.bin to the SD card and boot the board