Cyclone V SoC Secure Boot Example

27 Jan 2022 - 09:57 | Version 18 | KrisChaplin | ecdsa secure boot aes

Board:	DE10-Nano Development Board
State:	running
Miembros:	KrisChaplin

Introduction

This document describes mechanisms and implementation techniques to perform an authenticated software boot with encrypted hardware design on Intel Cyclone V SoC and Arria V SoC devices. The files for the hardware and software implementation are hardware-13-10-17.tar.gz and software-13-10-17b.tar.gz

Background

In order to securely boot a processor, it is required that each step of the boot process be trusted. This means that mechanisms are needed in order to ensure that from reset up to the running of the final application, only authorized software is able to be run.
Intel 28nm SoC FPGA devices do not have direct authentication support in the processor Boot ROM - this feature is available in later SoC FPGA families. However, with the implemetation of three features of the Cyclone V SoC and Arria V SoC devices, it is possible to control the boot process in such a way as to only allow authorized software to run.

Securing the boot flow

There are four steps that are required in Cyclone V SoC and Arria V SoC in order to harden the device against unauthorized software being run.

Step 1: Forcing FPGA images to be encrypted

A feature of the 28nm FPGA and SoC devices is that they can be programmed via one-time-programmable fuses to accept and decrypt FPGA configuration bitstreams using a shared secret. An additional fuse can restrict the FPGA to only accept encrypted bitstreams and fail to load plain text programming files. These mechanisms aim to allow the hardware developer to protect their design from being cloned into unauthorized hardware as well as optionally prevent unauthorized images from being run. Details on how to encrypt and program the fuses for 28nm FPGAs can be found in Application Note AN556.
By programming the FPGA to only accept encrypted bitstreams, the user can mandate that only programming files they have encrypted can used. By configuring the HPS to boot from the FPGA with the BSEL pins, the user can cause the Boot ROM to jump to preloader code within the FPGA on startup. Initialized FPGA memory connected to the HPS2FPGA bridge can only be executed if part of an FPGA image encrypted with the correct key, hence the preloader code is secured against plaintext programming files overriding the code to be executed. This preloader software, when protected forms the basis of trust of the running software.

Step 2: Authenticating next stage boot

In a standard SoC boot environment, the preloader has the task of enabling HPS I/O pins, calibrating external DDR memory and then loading the next stage payload into DDR. This payload is usually defined as the boot loader. In a secure boot system, the preloader has an additional task - namely to authenticate the next stage of boot, and ensure that only code that is deemed as authenticated is allowed to run. This authentication technique should only rely on keys, or key hashes stored and initialized within the FPGA, as only this region of code has been protected by the FPGA bitstream encryption.

Step 3: Authenticating and decrypting Operating System and applications

The final stage of the boot flow once a trusted, authenticated boot loader is run is to authenticate and optionally decrypt payload for the Operating system, device tree and ramdisk images. The flow for this is not specific to 28nm SoC FPGA devices, and so is not covered in this application note. The key or key hash for authentication and decryption algorithms as well as the algorithm used for the crypto functions must all be contained within the application payload that is authenticated by the preloader.

Step 4: Hardening the SoC FPGA from board-level tampering

It should be noted that it is not possible with 28nm SoC FPGAs to 100% safeguard against unauthorised software being run if physically attacked by a user with unlimited resources at their disposal. It is however possible to harden the device against two specific vectors that may be deemed as potential entry points.

Step 4a: Disabling access to the processor JTAG interface
It is possible for the preloader early in the boot process to physically disable the JTAG interface of the processor system. In doing so, it will not be possible for a user to attach a JTAG debugger to the target and use it to interfere with the program flow or bypass the authentication mechanisms. Disabling JTAG access can be achieved from the processor with a trivial register write to a configuration register. This mechanism can be reversed by trusted software at runtime if needed.

Step 4b: Force the processor to execute trusted code from FPGA on-chip memory
The Boot ROM software embedded into 28nm SOC FPGA devices will typically sample the BSEL external pins in order to determine which boot media to load the next stage boot from. This next stage boot software is

commonly called the "preloader" as it is code that is executed prior to the boot loader. The preloader code can be located in the following devices, which is selected using the BSEL pins:
FPGA - Via the HPS2FPGA Bridge
NAND Flash
SD/MMC
QSPI Flash

For this application, the BSEL pins would be set to "FPGA - Via the HPS2FPGA Bridge". However, the BSEL pins on the SoC FPGA could potentially be changed by a malicious user with physical board access. This could be achieved by removing resistors or overdriving the voltage of the BSEL pins on the PCB. This would allow the attacker to prevent the correct booting of the processor system, thus potentially allowing access to the device via JTAG. Mechanisms exist within the FPGA to override the BSEL external pins and for the processor to ignore them in favour of FPGA-driven signals. Once the FPGA is configured, the processor can be forced to boot from the FPGA on-chip memory, regardless of external PCB settings for the BSEL pins.

Example system – Extending the minimal preloader to authenticate next stage payload

In this example application, the Minimal Preloader (MPL) has been extended to include support for authentication using the ECDSA Algorithm. This is achieved by using libraries from mbed TLS, which are available under open source or commercial licenses.

Hardware Architecture

From a hardware perspective there are three IP blocks that need to be included in any Cyclone V SoC or Arria V SoC Qsys project in order to support an authenticated boot.
1. On-chip FPGA memory of at least 64KB
The basis of the secure boot flow is to trust the executable code that is running on the target, right from the first stages of the preloader. As determined earlier in this document, the mechanism to achieve this is to initialize FPGA RAM contents with the preloader code and force the FPGA to only accept correctly encrypted bit streams. The on-chip memory should be placed at the base address of the HPS2FPGA bridge, giving an effective processor base address of 0xC0000000.
2. Memory controller for access to off-chip non-volatile memory
The payload for the next stage of boot will likely be held off-chip. A memory controller is required in order for the preloader with authentication to copy the data into DDR memory for authentication and boot. In the example application, an Altera Serial Flash Controller IP is used to interface to an EPCS or EPCQ flash configuration device.
3. Powerup reset IP
On configuration of the FPGA, it is important to reset the processor system, to ensure that the forced BSEL values that are configured are sampled by the Boot ROM. This is achieved with an IP that provides a power up reset. This IP is available in the example system, and is called "Powerup delayed reset".

Additional considerations

At time of publishing, it is not possible to force the BSEL signals internally within the Qsys tool. The Qsys tool generates a top level HDL wrapper file, and it is this file that needs to be changed in order to force the BSEL values. In the example project, this is automated using a TCL script file that is run prior to implementation.

Generation of the hardware project

This example has been built for the Terasic de10-nano board, and was generated in the Quartus 16.1.2 Build 203 tool. It has been tested on Quartus Prime 16.1.2 Build 203 and Quartus Prime 17.0.2.Build 602

1. Extract the hardware zip file hardware-13-10-17.tar.gz to a working directory. This will result in a root folder "hardware" that contains all the required hardware project files.
2. Open the Quartus QPF project file "atlas_secureboot.qpf" in the Quartus II 16.1.2 GUI.
3. Open Tools→Qsys to open the Qsys GUI, and select "processor_system.qsys" in order to open the processor project.
4. After reviewing the sample hardware system, click on "Generate HDL..." and the "Generate" in order to build the hardware Qsys subsystem.
5. After generation has been completed successfully, close the Qsys GUI and return to the main Quartus II window.
6. In an embedded command shell, navigate to the hardware root directory and run the following command to force the BSEL signals as needed:
   patch -p0 < csel_bsel.patch
   The command should respond "Hunk #1 succeeded"
7. Implement the hardware design in Quartus Prime by selecting Processing→Start Compilation

Generation of the software application files

1. Generation of the secure boot application

1. Extract the software zipfile software-13-10-17b.tar.gz into the same directory that the hardware was extracted.
   After extraction of the software zip file, you will have the following folders:
   - **software/bin2hex**
     This software is used to make a correctly formatted hex file for initialisation of the secure boot bootloader into Qsys on-chip memory.
   - **software/keys**
     This is the location where your private and public keys are held. On the first run of Make, the keys are generated here, and a library is made with the public key in it for inclusion into the secure boot software executable.

   - **software/mbedssl-library**
     This makefile compiles the mbedssl source code and makes a library of the functions required for the secure boot library.

   - **software/secure_boot**
     This directory contains the main minimal preloader application along with the extensions to

authenticate the next stage of boot.

- **software/spl_bsp**
  This empty directory will contain the output from the BSP generation flow.

- **software/uboot_payload**
  This directory will contain the payload for signed next-stage boot loader images to store in external flash.

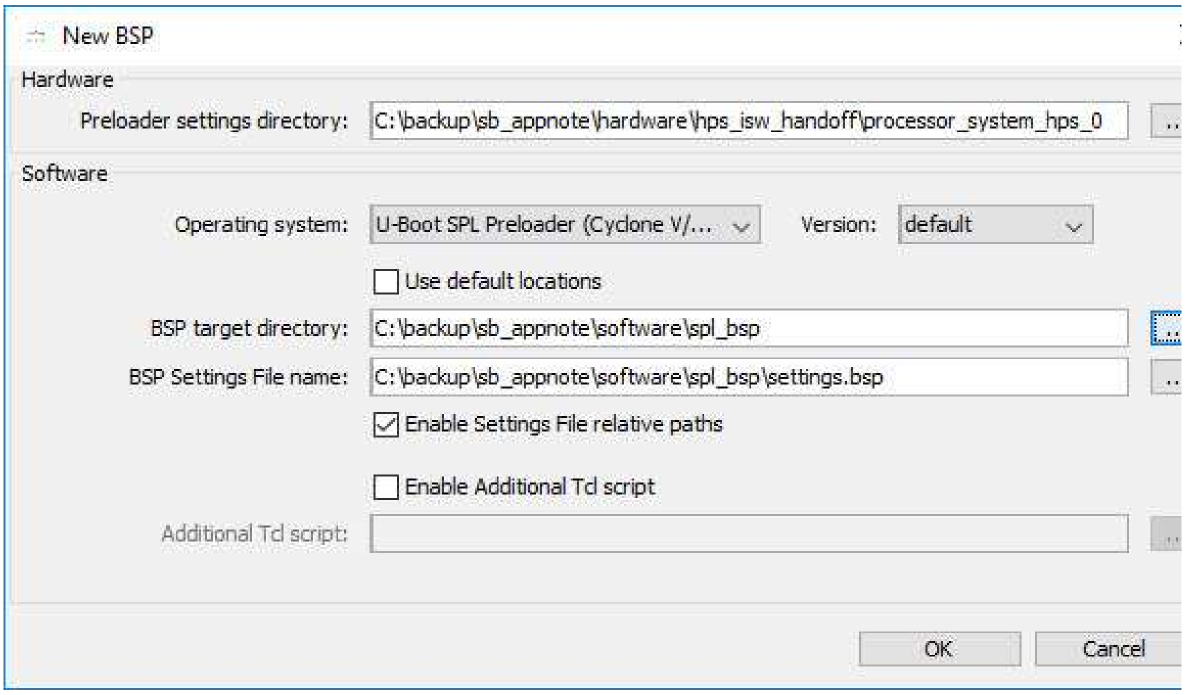
2. In an embedded command shell, change to the software/bin2hex directory and execute "make" to build the bin2hex tool.
3. In an embedded command shell, change to the software/keys directory and execute "make" to generate private/public keys for signing the final software images.
4. In an embedded command shell, change to the software/mbedssl-library directory and exeute "make" to build a library of crypto functions that are used to build the main secure boot executable.
   In the case that running make on mbedssl-library fails, it may be that wget cannot access the internet directly. To resolve this issue, manually download the required mbedssl source file from the link in the error and place it in the mbedssl-library directory. For example, if the error message showed as follows:

   ```
   --2017-10-05 13:27:29-- (try:20) https://tls.mbed.org/download/mbedtls-2.4.0-apache.tgz
   Connecting to tls.mbed.org (tls.mbed.org)|79.170.91.36|:443... failed: Connection timed out.
   Giving up.

   make: * [mbedtls-2.4.0-apache.tgz] Error 4
   ```

   Then manually download https://tls.mbed.org/download/mbedtls-2.4.0-apache.tgz with an internet-connected machine/browser and place in the mbedssl-library directory. Once copied, run make to continue the build flow.

5. Generate the board-specific initialization files. The minimal preloader sources need to be configured to correctly configure the pin multiplexing and DDR interfaces. This is achieved by generating preloader sources from the BSP generator.
   1. From an embedded command shell, execute "bsp-editor" to launch the preloader generator.
   2. Open "File→New HPS BSP" and point the preloader settings directory at ../hardware/hps_isw_handoff/processor_system_hps_0 directory. Please note that this directory will only exist after hardware generation has been completed successfully.
   3. Uncheck "use default locations" and select the BSP target directory to be the software/spl_bsp directory that was created when you extracted the software source. When completed click on ok, and acknowledge that the directory will be overwritten.


New BSP

Hardware

Preloader settings directory: C:\backup\sb_appnote\hardware\hps_isw_handoff\processor_system_hps_0

Software

Operating system: U-Boot SPL Preloader (Cyclone V/...   Version: default

☐ Use default locations

BSP target directory: C:\backup\sb_appnote\software\spl_bsp

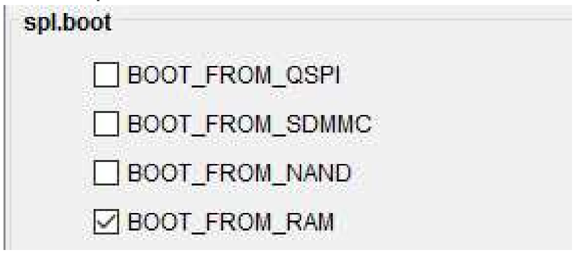
BSP Settings File name: C:\backup\sb_appnote\software\spl_bsp\settings.bsp

☑ Enable Settings File relative paths

☐ Enable Additional Tcl script

Additional Tcl script:

4. In the spl.boot section, disable BOOT_FROM_SDMMC and enable BOOT_FROM_RAM.


spl.boot

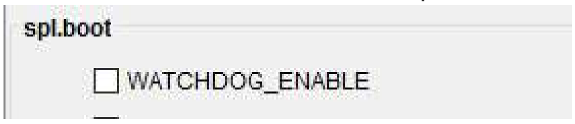
☐ BOOT_FROM_QSPI

☐ BOOT_FROM_SDMMC

☐ BOOT_FROM_NAND

☑ BOOT_FROM_RAM

5. In the advanced section, disable spl.boot WATCHDOG_ENABLE


spl.boot

☐ WATCHDOG_ENABLE

6. Click generate, and then exit the preloader generator GUI.

6. Now that the keys, TLS library, and preloader settings have been generated, the secure boot executable can be built. Do this by changing to the software/secure_boot directory and running "make". This completes generation of the secure boot authentication software.

2. Generation and signing of the next stage boot loader

1. Generate the payload executable. In our case this will be u-boot, so that will be generated next. In an embedded command shell, move to the software/spl_bsp directory and execute "make uboot" to generate a u-boot executable file.
2. Sign and package the payload executable ready for storing in flash memory by changing to the software/uboot_payload directory and executing "./encrypt_payload.sh"

3. Preparing the system for boot

The on-chip memory of the FPGA needs to be initialized with the secure boot executable code. To do this, perform the following steps:

1. With the hardware project open in Quartus Prime, select Processing→Update Memory initialization file to update the Quartus database with the new memory file contents
2. In Quartus Prime, select Processing→Start→Start Assembler to create a new programming file including the preloader in the on-chip RAM.

After following the steps listed, you should have all the files needed to execute a secure boot environment. The next and final process is to package these files for your specific flash device and program it.

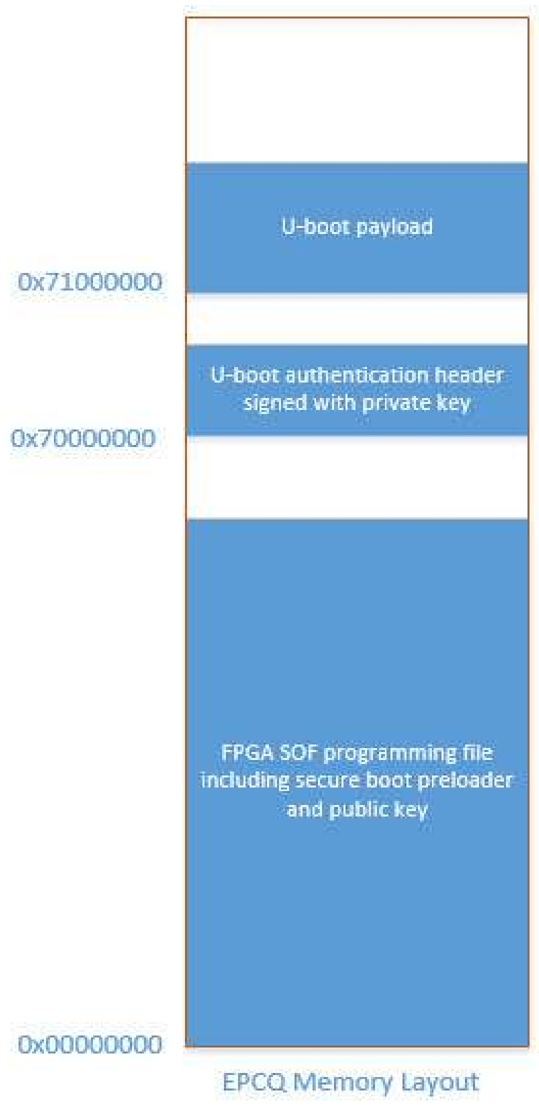
4. Preparing files for flash programming

The quartus_cpf convert programming files executable will package a programming file and multiple data files ready for programming.

1. From an embedded command shell, move into the hardware directory, and run the following command to build the .jic file ready for programming into the EPCS flash device:
   quartus_cpf --convert output_files/convert.cof

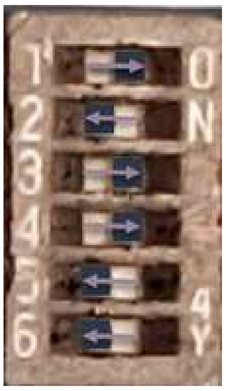
   When completed, this command will create a file "secure_boot_flash_image.jic" containing

- The programming file for the FPGA including initialized on-chip memory
- A header that contains the size and signature of the payload
- The payload boot loader image

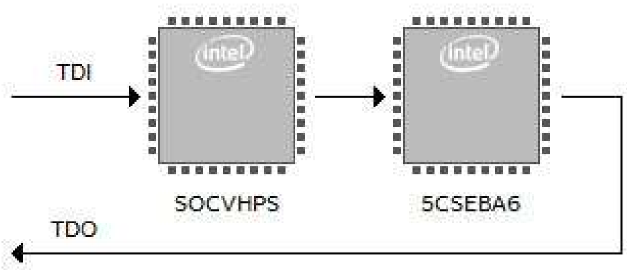

U-boot payload
0x71000000
U-boot authentication header signed with private key
0x70000000
FPGA SOF programming file including secure boot preloader and public key
0x00000000
EPCQ Memory Layout

5. Programming the flash memory on the DE-10 Nano board

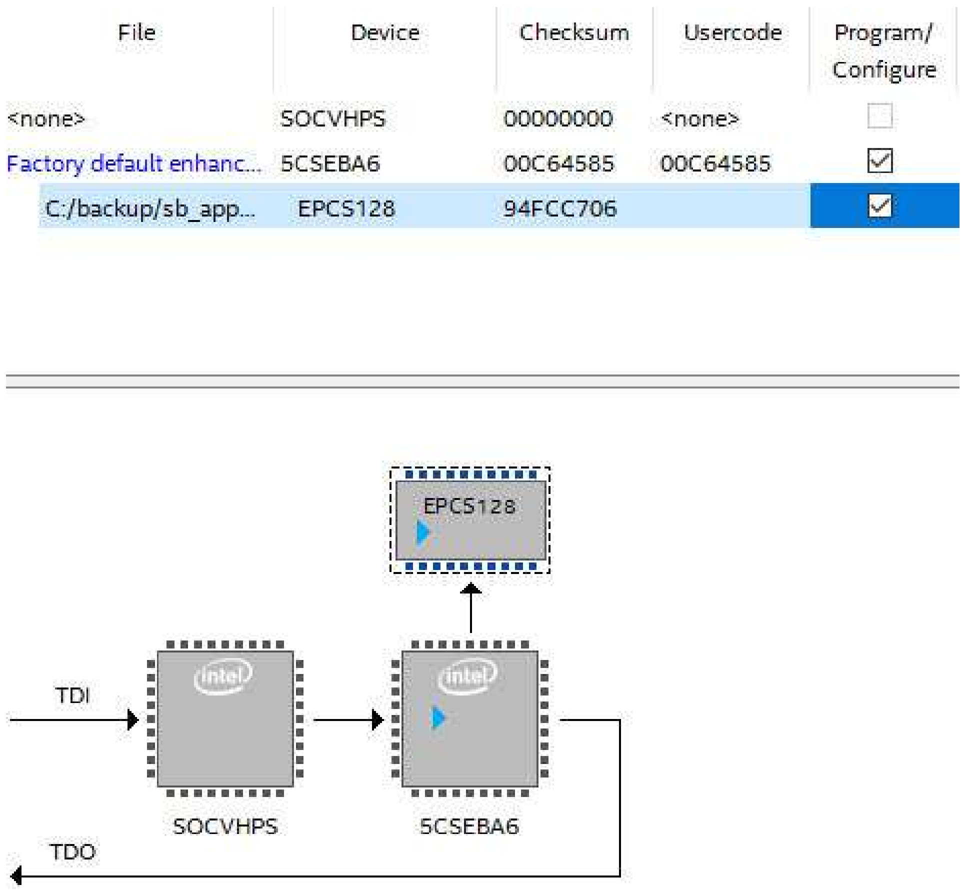
1. Ensure that the SW10 Dipswitches are set to 0,1,0,0,1,1 to select the Active serial mode to be selected.



2. Power the board, and with the JTAG USB Cable connected to J13, open the Quartus Prime programmer from Quartus Prime, by clicking on tools→programmer.
3. In Quartus Prime programmer, click on Hardware Setup, and select the DE-SOC usb cable as the currently selected hardware, before clicking on close
4. In the main Quartus Prime programmer window, click on Auto Detect to auto-detect the JTAG chain, which should contain two devices - the SOCVHPS and the 5CSEBA6 device



5. In the top list of devices, double-click on the file column next to the 5CSEBA6 device. In the resulting file selection gui, select "secure_boot_flash_image.jic" from the hardware directory of the project.
6. An EPCS128 device will appear in the list. Select this device, and check the box to Program/Configure the device.



File	Device	Checksum	Usercode	Program/ Configure
<none>	SOCVHPS	00000000	<none>	☐
Factory default enhanc...	5CSEBA6	00C64585	00C64585	✓
C:/backup/sb_app...	EPCS128	94FCC706		✓

7. Click Start to start programming the flash device.

6. Running the secure boot demonstration

Now that the flash is programmed, it should be possible to re-boot the board and see the authentication output from the secure boot program.

1. Connect a USB cable to the J4 RS232 port on the board
2. Configure a terminal program, such as Putty, and configure it to receive at 115200 baud, no parity, 8 Data Bits, 1 Stop bit, no flow control.
3. Power up the board to observe the terminal output. Output should be similar to the following

```
INIT: MPL build: Jul 24 2017 14:17:51
INIT: Initializing board.
INIT: MPU clock = 800 MHz
INIT: DDR clock = 400 MHz
INIT: Initializing successful.
```

```
SBOOT: Sig
SBOOT: Read
SBOOT: Auth
SBOOT: Boot
U-Boot 2013.01.01 (Jul 24 2017 - 14:52:54)

CPU : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone V Board
I2C: ready
DRAM: 1 GiB
MMC: ALTERA DWMMC: 0
QSPI: QSPI is still busy after poll for 10000 times.
SF: Calibration failed (read)
*** Warning - spi_flash_probe() failed, using default environment
```

Attachments (10)

Show options

Previous 1 2 Next

2017-10-05_15-00-06.jpg (48.75 KB)
05 Oct 2017 - 14:04 | Version 1 | Kris Chaplin

hardware-13-10-17.tar.gz (156.00 KB)
13 Oct 2017 - 11:53 | Version 1 | Kris Chaplin

secure_boot_flash_image.jic (16.00 MB)
13 Oct 2017 - 11:53 | Version 1 | Kris Chaplin

software-13-10-17b.tar.gz (61.45 KB)
13 Oct 2017 - 11:53 | Version 1 | Kris Chaplin

Previous 1 2 Next

More actions | 10 attachment(s) | Borrar todo | Select all