Microsemi®

Microsemi Secure Boot Reference Design

White Paper

June 2014

Abstract

Networked embedded systems are used in an ever-expanding range of applications, where monitoring and control are a critical element. Unfortunately, this ubiquitous connection can allow network-based attacks and fielded systems may be subject to hardware attacks. This paper describes a reference design created by Microsemi that illustrates a secure boot process which can be used to insure the boot process for an MCU, DSP, or SoC FPGA is secure and authentic. A live demonstration of the accompanying reference design is available in the Microsemi booth for attendees wishing to see the specific implementation details.

Introduction

Embedded systems are used in an ever-expanding range of applications where monitoring and control are a critical element. In industrial process controllers, financial transaction systems, transportation infrastructure and the rapidly developing Machine-to-Machine (M2M) collection of the Internet of Things (IoT), embedded systems literally control our world. The addition of networked connectivity delivers advantages for both the user and the system operator. Unfortunately, the ubiquitous connection can allow network-based attacks against these embedded systems; and fielded systems may be subject to hardware attacks. One of the most vulnerable processes targeted by advanced threats is the system boot process. Protecting it is critical if an embedded system needs to be secure from either network-based threats or hardware-targeted intrusions.

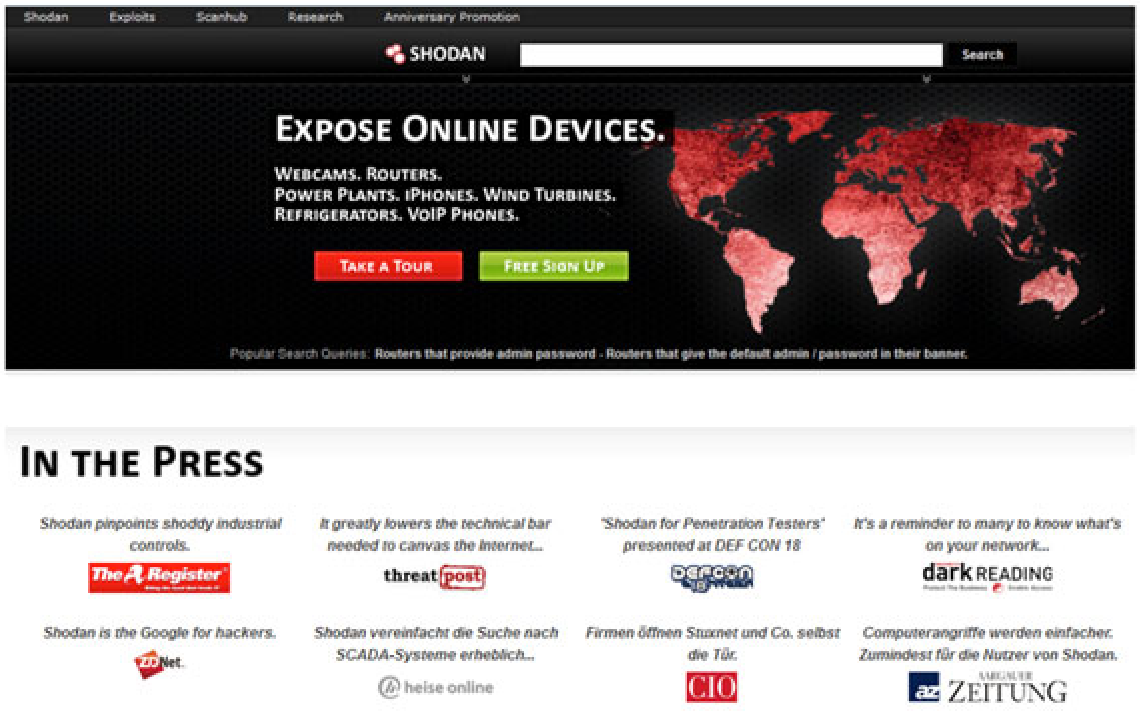
This paper provides a detailed introduction to the Microsemi secure boot reference design. The reference design is detailed and complete enough to demonstrate a real world secure boot implementation. It is organized to simplify customization for a wide range of target applications and with a variety of target processors, many of which, like many MCUs, MPUs, and most DSPs, do not intrinsically support secure boot. The design shows how the code may be authenticated to prevent tampering and encrypted to preserve confidentiality. All the details of the implementation including demonstration boards, schematics, FPGA design files, and applications code are all available as part of the reference design.

Security Threats to Embedded Electronics

The dramatic growth in network connected embedded systems has been driven by the commoditization of 32-bit computing. 32-bit embedded processors with integrated Ethernet MACs running standard operating systems can be bought today for tens of dollars and smaller 32-bit flash MCU's with Ethernet connectivity cost just a few dollars. Unfortunately, low cost connectivity can come with hidden expenses when the system isn't secure.

Surprisingly, a very large number of embedded systems are not secure. The Shodan computer search engine welcome page, which is shown in Figure 1 on page 3, can identify virtually any internet-connected device. Shodan uncovers over 500 million internet control systems that could be at risk.

Recent news coverage with some of the most pertinent articles (Figure 1), illustrate the seriousness of this new form of attack on embedded systems.



EXPOSE ONLINE DEVICES.

WEBCAMS. ROUTERS.
POWER PLANTS. IPHONES. WIND TURBINES.
REFRIGERATORS. VOIP PHONES.

Popular Search Queries: Routers that provide admin password - Routers that give the default admin / password in their banner.

IN THE PRESS

Shodan pinpoints shoddy industrial controls.

It greatly lowers the technical bar needed to canvas the Internet...

'Shodan for Penetration Testers' presented at DEF CON 18

It's a reminder to many to know what's on your network...

Shodan is the Google for hackers.

Shodan vereinfacht die Suche nach SCADA-Systeme erheblich...

Firmen öffnen Stuxnet und Co. selbst die Tür.

Computerangriffe werden einfacher. Zumindest für die Nutzer von Shodan.

Figure 1: **Shodan Demonstrates the Vulnerability of Networked Embedded Systems**

The development of new forms of detection is only one aspect of the rise in vulnerability for networked embedded systems. New software tools have been developed that can be used to detect vulnerabilities of networked equipment. In most cases these tools, like those illustrated in Figure 2 below, are useful for testing a network so that countermeasures can be developed. Unfortunately, unscrupulous uses could detect vulnerabilities in targeted systems, perhaps found through Shodan, and thus make attacks much easier to launch.

Figure 2: **Tools that can be used to Detect Vulnerabilities of Networked Devices**

Risks to Embedded Systems – Design Security, Data Security, and Secure Boot

Risks to embedded systems come from two main sources. Design Security risks are associated with the Intellectual Property of the design itself—the FPGA configuration bitstream or the code for an embedded CPU are just two common examples. If this valuable design information can be stolen, it can have a significant (perhaps even critical) impact on your business. Data Security is associated with the data that resides or flows through an embedded system. It can be customer information, financial transactions, or a surveillance video stream. Significant financial impact can also result if this secure data can be attacked and captured.

An Overview of the Secure Boot Process

One of the most important security capabilities to protect embedded systems is a secure boot process. A secure boot process initializes an embedded processing system from rest. It does this by executing trusted code, free from any tampering by a malicious intruder. Without this level of trust, an alternate boot image could replace the original boot code and allow an attacker to 'hijack' the entire embedded system. Just about any embedded system needs to be free from such attacks, for many embedded systems such an event could prove catastrophic. Financial transactions could be altered, industrial processes sabotaged, or confidential business data compromised. It is easy to see why security, and in particular secure boot, is a growing requirement for many embedded systems.

Root-of-Trust –The Starting Point for Implementing Secure Systems

A hardware Root-of-Trust is essential to system security. It is an entity that can be trusted to always behave in the expected manner. As a system element, it supports verification of system, software and data integrity and confidentiality, as well as the extension of trust to internal and external entities. The Root-of-Trust is the foundation upon which all further security layers are created, and it is essential that its keys remain secret and the process it follows is immutable. In embedded systems, the Root-of-Trust works in conjunction with other system elements to ensure the main processor boots securely using only authorized code, thus extending the trusted zone to the processor and its applications.

Multi-Stage Secure Boot Process Description

Initializing embedded processing systems from rest requires a secure boot process that executes trusted code free from malicious content or compromise. Figure 3 on page 5 illustrates the various phases a secure boot process must go through to adequately protect the initialization of an embedded system. Validation of each stage must be performed by the prior successful phase to ensure a 'chain-of-trust' all the way through to the top application layer. The immutable boot loader (Phase 0) code can be embedded and validated within the hardware Root-of-Trust device, which ensures the integrity and authenticity of the code.

Each sequential phase of the secure boot is validated by the previously trusted system before code and execution is transferred to it.

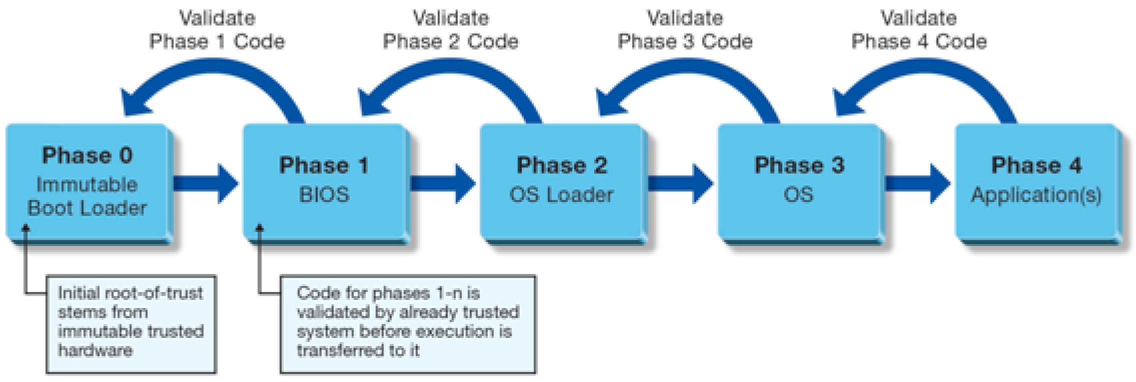

Validate Phase 1 Code
Validate Phase 2 Code
Validate Phase 3 Code
Validate Phase 4 Code
Phase 0 Immutable Boot Loader
Phase 1 BIOS
Phase 2 OS Loader
Phase 3 OS
Phase 4 Application(s)
Initial root-of-trust stems from immutable trusted hardware
Code for phases 1-n is validated by already trusted system before execution is transferred to it

Figure 3: **Multi-Stage Secure Boot Process – Simple Overview**

It is essential that any code be validated prior to delivery and execution to ensure that no compromise has occurred that could subvert or damage the boot of each phase. This can be done using either symmetric or asymmetric key cryptographic techniques. One approach is to build an inherently trusted RSA or ECC security key into the immutable Phase-0 boot loader. The developer uses the RSA or ECC private key to digitally sign the Phase-1 code. During Phase-0, the Root-of-Trust subsystem validates the digital signature of the Phase-1 code before allowing execution. The boot process is aborted if invalid. It is critically important that the inherently trusted security key and the immutable Root-of-Trust signature checking process cannot be modified by a would-be hacker. If a hacker could substitute another security key or subvert the process, they could ‘spoof’ subsequently loaded digitally signed code.

Description of a Secure Boot Example Design

In order to better understand the details of the secure boot reference design, it is useful to discuss the secure boot in the context of an example design. A typical networked embedded system usually contains a target processor that manages the system peripherals, controls network data transfers, and records/ reports quality of service metrics to a central management unit. The target processor might be booted from an external SPI flash memory, and the booted code run from on-chip SRAM. The boot code, along with the associated application code, could be updated remotely to fix bugs and provide feature enhancements. In a secure implementation a hardware Root-of-Trust (RoT) would secure the boot process and perhaps manage the system power and other ‘low-level’ tasks to off-load the main processor and thus improve system efficiency. The RoT might also manage the Point of Load (PoL) power regulators, making it easy to shut down system power as a penalty if critical system intrusion is detected.

Figure 4 on page 6 shows an example design consisting of a main MPU as the target processor and a SmartFusion®2 SoC FPGA as the hardware RoT for the system. The main processor implements all the high-level control functions and interfaces to the rest of the system through standard interfaces such as USB, PCIe®and DDR memory.

In addition to the secure boot function the hardware RoT is responsible for implementing a secure remote update of application code, FPGA bitstreams or new boot loaders as well.

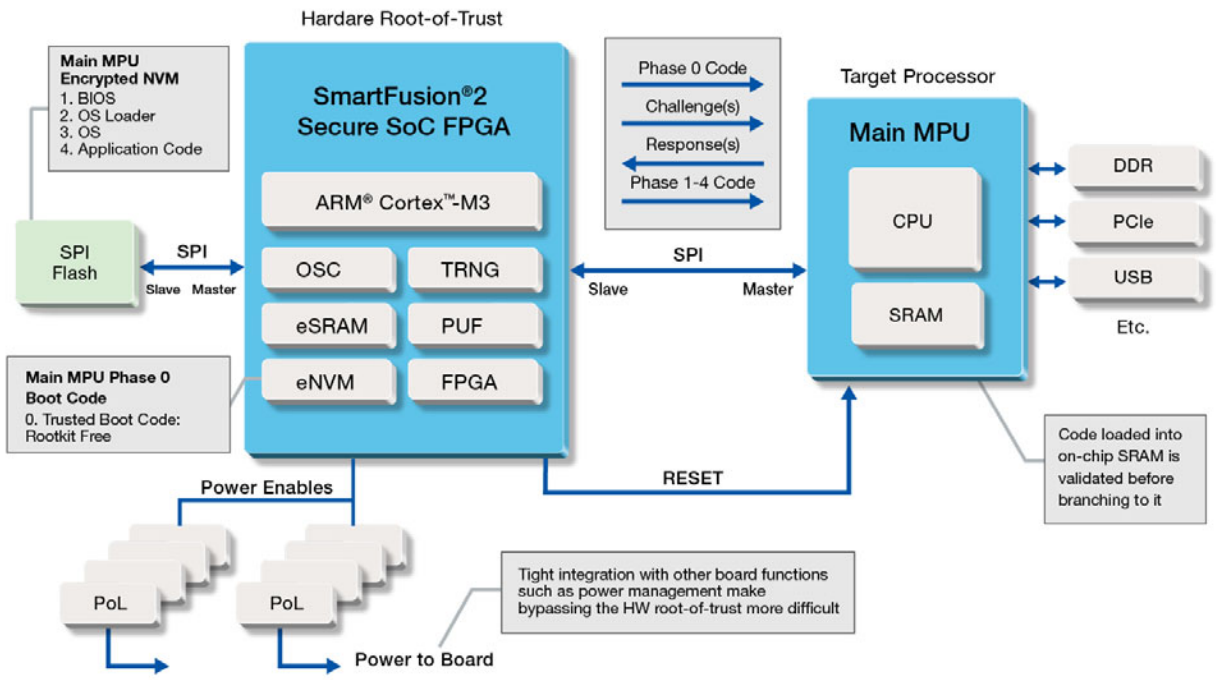


Figure 4: **Block Diagram and Key Secure Boot Processes for an Example System**

The SmartFusion2 SoC FPGA device has an integrated processor (ARM®Cortex™-M3), on-chip oscillator (OSC), True Random Number Generator (TRNG), embedded SRAM (eSRAM), embedded Nonvolatile Flash Memory (eNVM), a Physically Unclonable Function (PUF), crypto accelerators, and FPGA fabric. These key hardware elements, along with a variety of intrinsic security features, make the SmartFusion2 SoC FPGA an ideal target for an exceptionally secure hardware RoT.

In order to better understand the example design, a quick overview of the SmartFusion2 SoC FPGA architecture is given in the following section. After the SmartFusion2 SoC FPGA overview, we will return to the discussion of the example design with a better understanding of the key features and capabilities of the SmartFusion2 SoC FPGA device and their use in implementing the secure boot capability.

SmartFusion2 SoC FPGA Architecture – A Quick Overview

The SmartFusion2 SoC FPGA family architecture, as illustrated in Figure 5 on page 7, combines most of the common function blocks required in just about any digital electronics system. The system controller provides overall supervisory control of configuration, power management, and JTAG functions. The FPGA fabric includes the user configurable logic and connects to the MSS, DDR controllers and serial controllers. These main function blocks all work together efficiently to provide the designer with a best-in-class programmable system-on-chip (SoC) for a wide range of applications.

What sets SmartFusion2 SoC FPGA devices apart from other programmable devices are the differentiated features that have been included for increased reliability, advanced security, and low power.

For example, the sub-blocks shown in purple are immune against single event upset (SEU) occurrences by the use of flash memory cells instead of SRAM cells. These innovations result in dramatic reliability increases compared to SRAM-based programmable devices.

Design and data security are also radically improved by the inclusion of advanced security functions such as AES-256, SHA-256, SRAM-PUF, and NRBG in the system controller.

The next few sections provide an overview each of the main blocks and their differentiated features that assist in implementing security related functions for your designs.

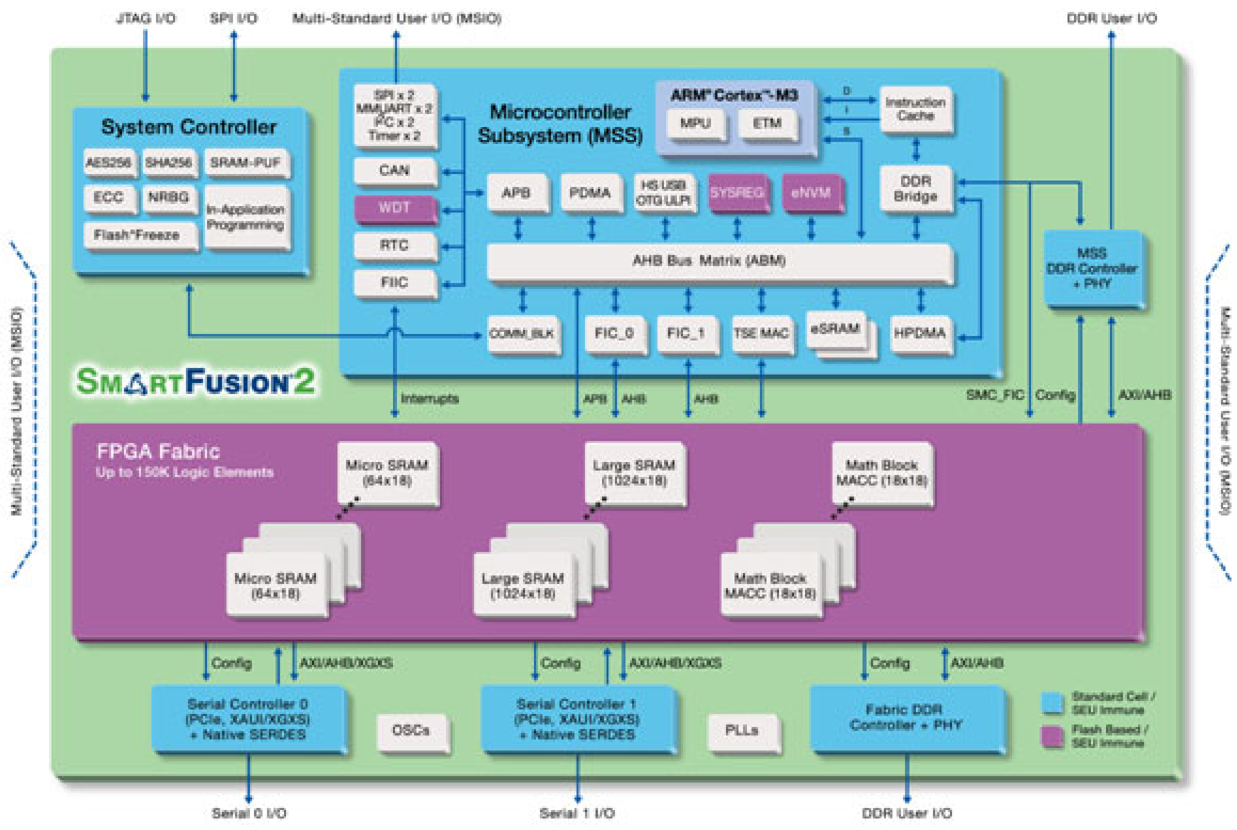


Figure 5: **SmartFusion2 Architecture Block Diagram**

SmartFusion2 SoC FPGA Microcontroller Subsystem Description

The SmartFusion2 SoC FPGA family combines several key processing, memory storage, and data transfer functions within the Microcontroller Subsystem (MSS). By combining these functions into a common block, SmartFusion2 SoC FPGA makes it easier to implement control functions using a traditional MCU development process. The 'hardened' blocks of the MSS result in a more highly integrated, lower power and lower cost implementation for embedded control applications. Key elements of the MSS that are useful in implementing Secure Boot applications include the ARM-Cortex-M3 CPU; memory related functions including the cache, SRAM, flash code storage blocks, High-Performance DMA (HPDMA) and the external DDR memory controller; a variety of high speed peripherals for data communications including Tri-speed Ethernet, USB, and a peripheral DMA controller (PDMA); several lower speed communications peripherals like SPI, I^{2}C, UART and CAN; timers and counters; and interfaces to the FPGA fabric.

FPGA Fabric

The ability to customize user logic and efficiently and easily integrate it with the other key blocks is perhaps one of the most important capabilities of the SmartFusion2 SoC FPGA family. SmartFusion2 SoC FPGA devices are optimized for advanced security, high-reliability, and low power consumption applications. Several key features of the embedded FPGA fabric contribute to the best-in-class results the SmartFusion2 SoC FPGA family delivers.

SmartFusion2 SoC FPGA fabric is composed of five key building blocks: the logic module, the large SRAM, the micro SRAM, the Mathblock and the routing resources that connect everything together. These low level elements can be used to construct the wide range of functions required in embedded systems designs.

High-Speed Serial Interfaces – SERDES Interface

The SmartFusion2 SoC FPGA high-speed Serializer Deserializer (SERDES) interface block supports multiple high-speed serial protocols using a SERDES transceiver of up to 5 Gbps, supporting Peripheral Component Interconnect Express (PCI Express®), eXtended Attachment Unit Interface (XAUI), and Serial Gigabit Media Independent Interface (SGMII). In addition, any user defined high-speed serial protocol implemented in the IGLOO®2 FPGA fabric can access SERDES lanes through the external physical coding sublayer (EPCS) interface. The SERDES block is configurable to support single or multiple serial protocol modes of operation. The SERDES block is connected to the FPGA fabric through an AXI/AHBL interface or EPCS interface.

System Security Block

The SmartFusion2 SoC FPGA system security block manages all the programming and security related functions included on SmartFusion2 SoC FPGA devices. The best-in-class security features of IGLOO2 FPGAs include Encrypted User Bitstream-Key Loading, Certificate-of-Conformance support, X.509 Compliant Digital Certificate Management, Elliptic Curve Cryptography (ECC) support, an Advanced Encryption Standard (AES-128/256) engine, a Secure Hash (SHA-256) engine, a Non-Deterministic Random Bit Generator (NRBG), Differential Power Analysis (DPA) countermeasures, anti-tampering countermeasures, single-use debug passcodes, SRAM-PUF, zeroization and FlashLock® protection of bitstream decryption keys and security settings. Some of the higher-level functions (like zeroization, AES, SRAM-PUF, NRBG and SHA-256) are easily included in user designs as security services accessible through the security block.

This overview of the SmartFusion2 SoC FPGA architecture will be sufficient for our purposes of understanding the key device features and the support available for a wide range of embedded system designs. This paper will now focus on the security features needed to support a secure boot capability. Additional details on other SmartFusion2 SoC FPGA features are available in the SmartFusion2 SoC FPGA User Guides available on the SmartFusion2 SoC FPGA product page listed in the "To Learn More" section at the end of this paper.

High-Level Description of Secure Boot

Before jumping into a very detailed view of the secure boot reference design let's first orient ourselves with a high-level overview. This overview breaks the secure boot process into three key steps, steps that would typically be found in any secure boot implementation. With a good understanding of these steps, a much more detailed view of the reference design can be more easily digested.

The starting point for the reference design, based on our previously described example design, is illustrated in Figure 6 on page 9. The main hardware elements of the example design are the SmartFusion2 Secure SoC FPGA, the SPI flash memory, the target MPU, and the supporting peripherals. The SmartFusion2 SoC FPGA device stores in secure eNVM the boot code, WhiteboxCRYPTO™ code the CEK Encryption Key, the CSK signing key, and the RSA keys.

The SPI flash memory contains the application code encrypted with CEK and signed with CSK. This is the starting point for the secure boot design.

The goal of the secure boot process is to transfer the application code to the main MPU, successfully decrypt, and authorize the code on the main MPU, and then begin execution.

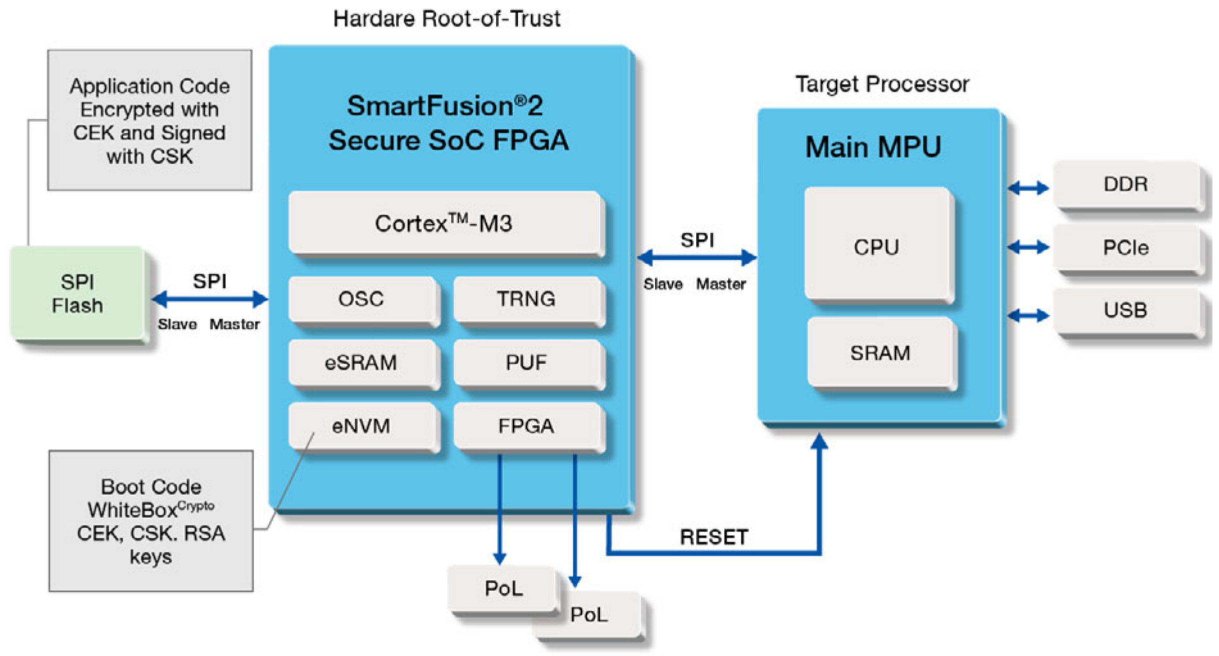


Figure 6: **Starting Point for Secure Boot in the Example Design**

The first step is illustrated in Figure 7 below. The main action in this step is to send the boot loader to the target processor. In our example design the boot loader consists of the WhiteboxCRYPTO code, a number used only once (or nonce), the obfuscated AES security key, and the RSA public key. This is all sent in plain-text form to the target processor when it begins its start-up process. The AES key and nonce are used by the target processor to calculate a CBC-MAC for code authentication using WhiteBoxCRYPTO.

The SmartFusion2 SoC FPGA device also calculates the CBC-MAC and compares it to the target transmitted value to authenticate that the boot loader code is tamper-free.

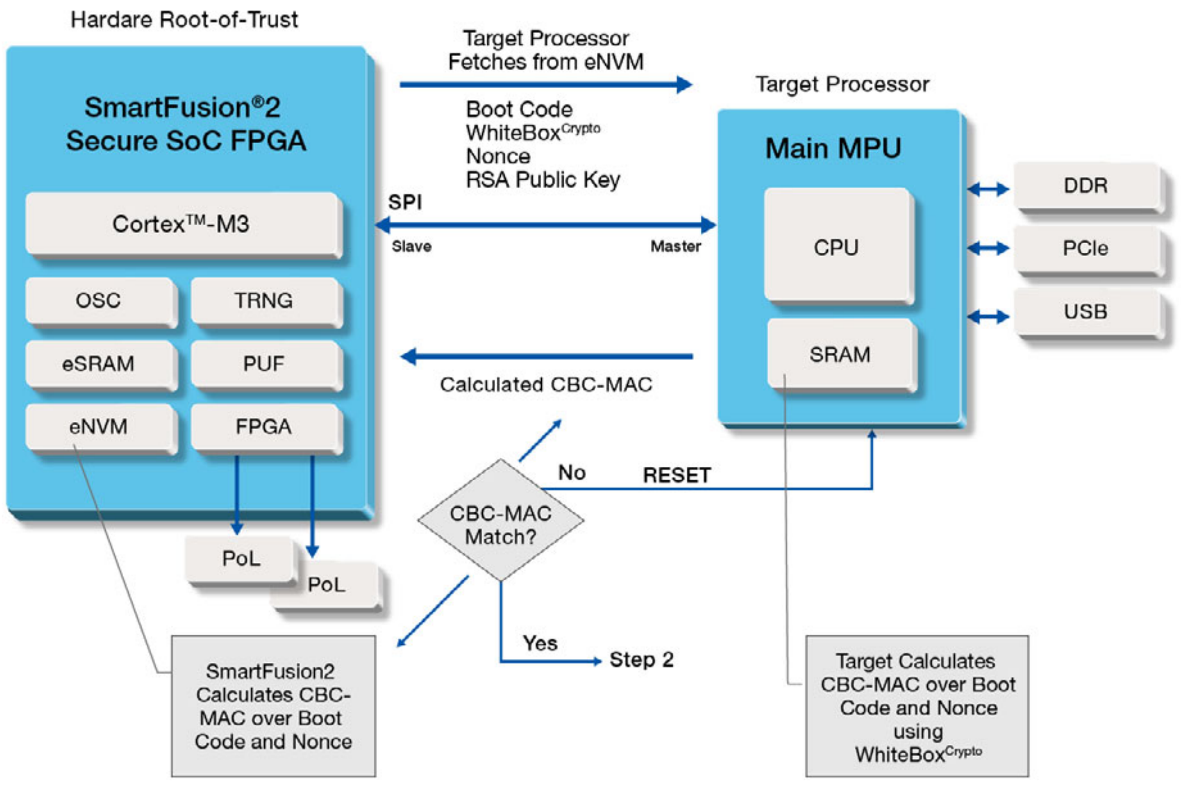


Figure 7: **Step 1 Sends the Boot Loader to the Target Processor**

The second step in the secure boot process, illustrated in Figure 8 on page 11, is to establish a shared key between the SmartFusion2 SoC FPGA device and the target processor for exporting CEK and CSK in encrypted form to the target processor. CEK and CSK will be used by the target processor to decrypt and authenticate the encrypted application code.

The target processor generates the ESK and encrypts it with the WhiteBoxCRYPTO AES key. The RSA public key is used to encrypt the result and the target processor sends this to the SmartFusion2 SoC FPGA device.

The SmartFusion2 SoC FPGA device recovers the ESK and uses this now shared key to wrap the CEK and CSK. These can now be securely transmitted to the target processor.

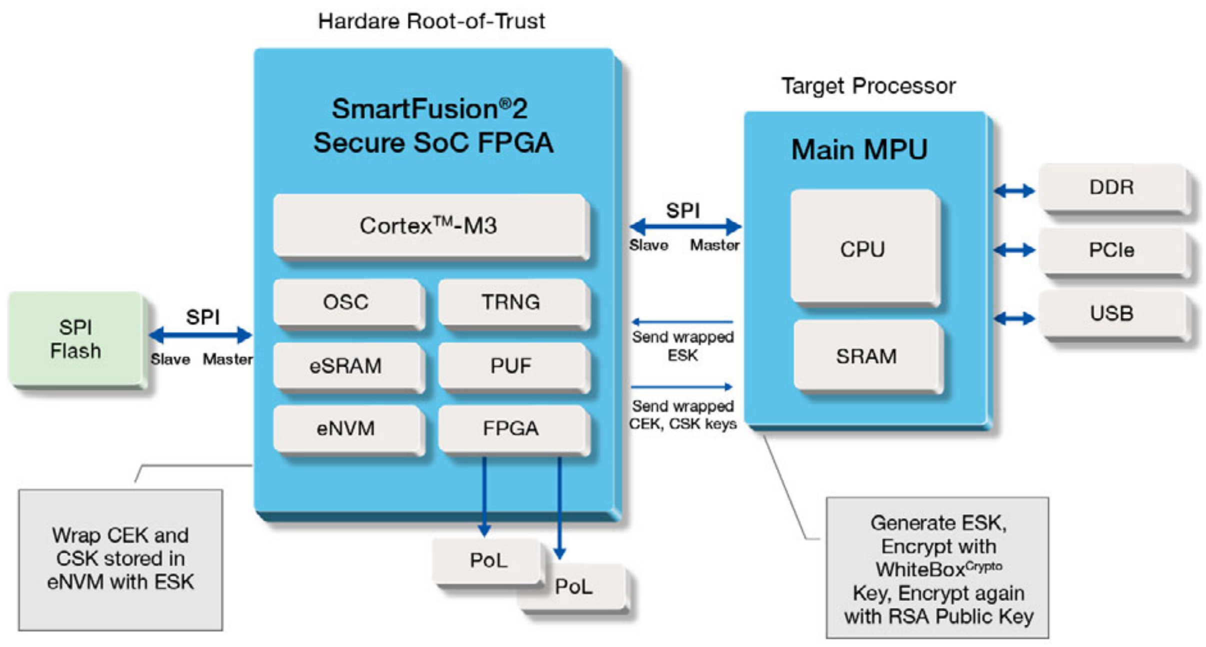

Hardare Root-of-Trust
SmartFusion®2 Secure SoC FPGA
Cortex™-M3
OSC
TRNG
eSRAM
PUF
eNVM
FPGA
PoL
PoL
SPI Flash
SPI
Slave
Master
Wrap CEK and CSK stored in eNVM with ESK
SPI
Slave
Master
Send wrapped ESK
Send wrapped CEK, CSK keys
Target Processor
Main MPU
CPU
SRAM
DDR
PCIe
USB
Generate ESK, Encrypt with WhiteBoxCrypto Key, Encrypt again with RSA Public Key

Figure 8: **Step 2 Establishes the Shared Key for the Encrypted Application Code**

The third and final step in the secure boot design, shown in Figure 9 on page 12, is to authenticate and decrypt the application code. The received CEK and CSK are unwrapped by the target processor, which then requests the secured application code. This code is then transmitted by the SmartFusion2 SoC FPGA device over the SPI port. The application code is received and decrypted, and if authenticated successfully the target processor can begin execution. The application code is known to be secure and tamper-free. The secure boot process is completed.

Note that trust has been extended to the application code so it can now be used to further extend trust to other secure transactions.

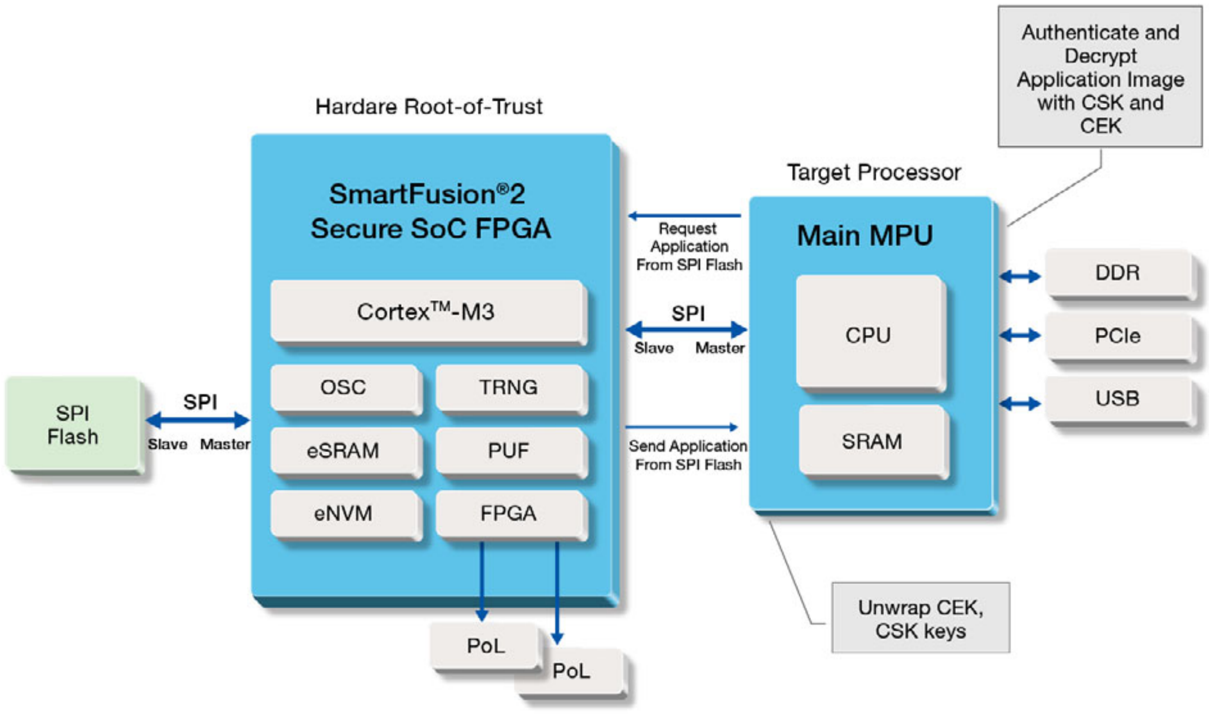


Figure 9: **Step 3 Authenticate, Decrypt, and Begin Execution**

Detailed Description of the Secure Boot Process in the Reference Design

Now that the high-level secure boot process has been described and understood in the context of an example design, it is appropriate to look at the step-by-step process at another level of detail. The detailed process contains many more sub-steps, but still follows the high-level outline as previously described. When the implementation is divided into individual steps, it’s much easier to identify the routines that can be used in a custom implementation.

Figure 10 on page 13 below shows the detailed implementation used in the Microsemi secure boot reference design. The tasks executed in the RoT are shown below the dashed blue line and the tasks executed by the target processor are shown above the dashed line. Tasks are shown in the blue boxes and labeled numerically to indicate the order of execution. In the target processor section tasks are grouped, within yellow boxes, to show what portion of the code they operate on. For example, step 6, calculation of the CBC-MAC, is done within the Boot Code group while step 15, execute application code, is done on the loaded application code. Other yellow boxes indicate various code and data blocks used during the process. Green boxes indicate keys, tags and other values used by the cryptographic functions.

Pink boxes indicate steps executed if an exception condition occurs—such as the detection of invalid code. The flow from task-to-task is indicated by blue arrows.

Blue arrows also indicate the use of security keys and tags and the movement of code from the RoT to the target processor (shown by the dashed blue lines).

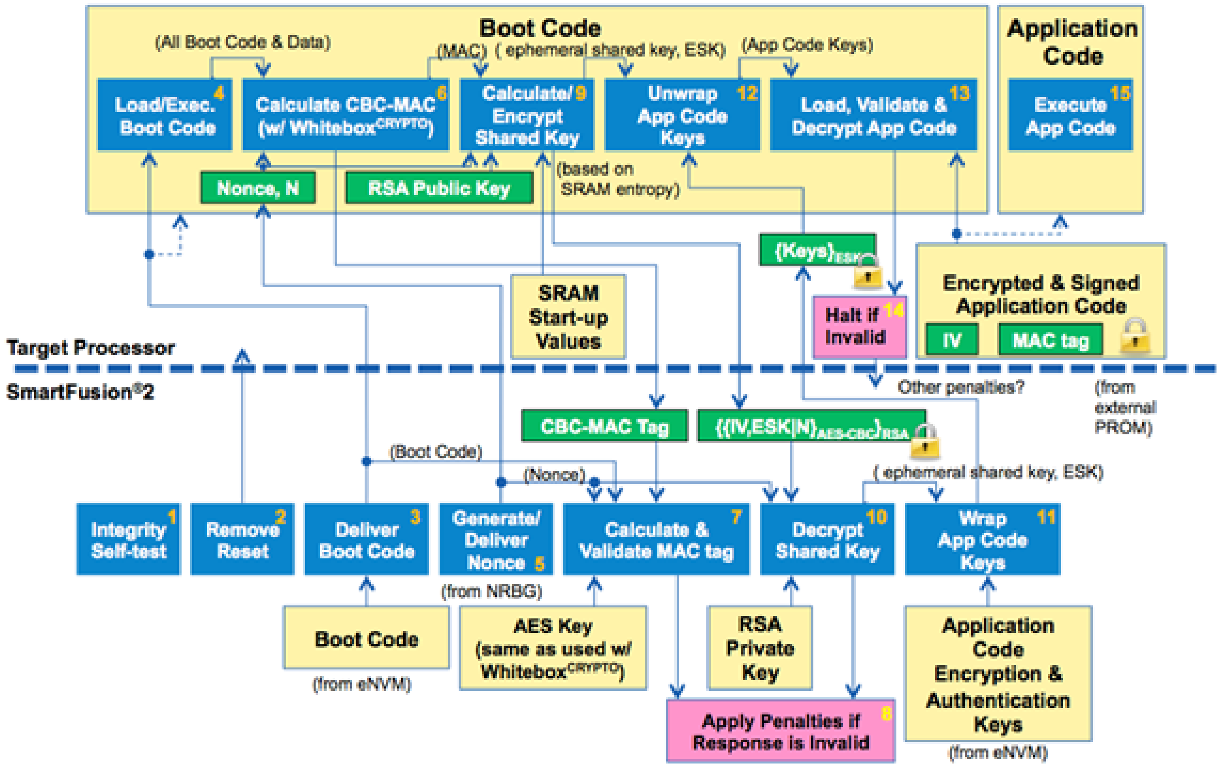


Figure 10: **Detailed Secure Boot Flow Diagram**

The process used in the reference design and described in Figure 10 is, still basically a three stage process. During the first stage, the Boot Code is transferred to the target processor in a secure and validated fashion (the large yellow box labeled ‘Boot Code’). Once it is known that the boot code is securely delivered and validated, the second stage transports and unwraps the shared keys. These keys are used to decrypt and authorize the target processor application code stored in external nonvolatile memory. During the third stage, the secure application code (the smaller yellow box labeled ‘Application Code’) is authorized, decrypted, and available to execute. A more detailed description of each step in the overall secure boot process is given below:

1. The SmartFusion2 SoC FPGA boots itself securely from its internal eNVM. After which a self-integrity test can be performed. This test is a Smartfusion2 SoC FPGA built-in feature that provides assurance against both natural and maliciously induced failures on all the nonvolatile configuration memory segments, including security keys, security settings, the FPGA fabric configuration, and any eNVM pages declared as ROM (all the write-protected pages). The SmartFusion2 SoC FPGA holds the target processor in reset state until it completes the power-on integrity self-test.
2. The SmartFusion2 SoC FPGA device releases the reset of the target processor so that it can start to boot from its internal boot ROM.
3. These tasks (steps three and four) operate in parallel. Once the target processor is released from reset, the boot-loader program stored in the target processors internal boot ROM fetches additional boot code through the SPI bus. This code will be loaded into its internal SRAM for execution.

4. The SmartFusion2 SoC FPGA emulates the target processors SPI memory and uses its internal eNVM to source the expected target processor boot code. When the transfer is complete, the target processor begins to execute the boot code.
5. The SmartFusion2 SoC FPGA generates a nonce using its NRBG and delivers it to the target processor along with the boot code. This nonce is regenerated at every boot-up and is used in subsequent steps to ensure the integrity and authenticity of the messages received from the target processor on every power-up.
6. The target processor executes the boot code, which contains a WhiteboxCRYPTO AES instantiation with an obfuscated AES key. WhiteboxCRYPTO algorithmically obfuscates a symmetric key and is used to transmit this key over a plaintext interface, namely the SPI port. The target processor calculates a cipher block chaining (CBC)—message authentication code (MAC) using the WBC key over all the boot code, including the nonce, and writes the resulting signature back to the SmartFusion2 SoC FPGA through the SPI interface. Since the nonce is different at every boot-up, the resulting signature varies each time the target processor boots up and protects the boot process against replay attacks (attacks that attempt to use a previously captured boot record to overwrite the current code, perhaps to install a previous code revision with known exploits). It is critical to have the AES key in obfuscated form since the boot code is transferred in plain text to the target processor. The WhiteboxCRYPTO AES algorithm uses the obfuscated AES key and performs the CBC-MAC calculation in such a way that the actual AES key extraction is difficult or infeasible.
7. In parallel with step 6, the SmartFusion2 SoC FPGA calculates a CBC-MAC over the same data (boot code and nonce) using the actual AES key and waits for the signature from the target processor. The SmartFusion2 SoC FPGA validates the received signature against the one it calculates internally. If it passes, the boot process continues to execute the now-trusted code in the target processor’s on-chip SRAM.
8. If the CBC-MAC tag from the target processor doesn’t arrive within the predefined time window, or the tags do not match, the SmartFusion2 SoC FPGA penalizes the system by asserting the reset signal to the target processor. The SmartFusion2 SoC FPGA can also impose additional penalties such as shutting down power, disabling communication, or zeroizing (erasing all internal data) itself.
9. The target processor’s boot code reads a section (for example, 3KB) of uninitialized SRAM, which is being used as a random bit generator. The SRAM contents are compressed to generate a 128-bit ephemeral shared key (ESK) by the target processor using the WhiteboxCRYPTO AES algorithm, including the MAC tag from step 6. This key and the partial nonce, N (up to a total number of bits equal to the RSA key size – 1024-bits) are double-encrypted, first with WhiteboxCRYPTO AES algorithm in CBC mode usingbit part of the nonce (for example, the last 16-bytes) as an Initialization Vector (IV), then using an RSA public key that was delivered as part of the now-authenticated boot code, and is sent to the SmartFusion2 SoC FPGA. It is critically important to encrypt the calculated ESK and partial nonce before RSA encryption to prevent man-in-the-middle (MITM) attacks (where an ‘impersonator’ communicates with each of the parties by using the public keys in an attempt to discover secret information).
10. The SmartFusion2 SoC FPGA performs double-decryption on the received message of 1024-bits to get the ESK and partial nonce. Double-decryption starts with the RSA decryption using the corresponding RSA private key and then the AES decryption using the actual AES key.

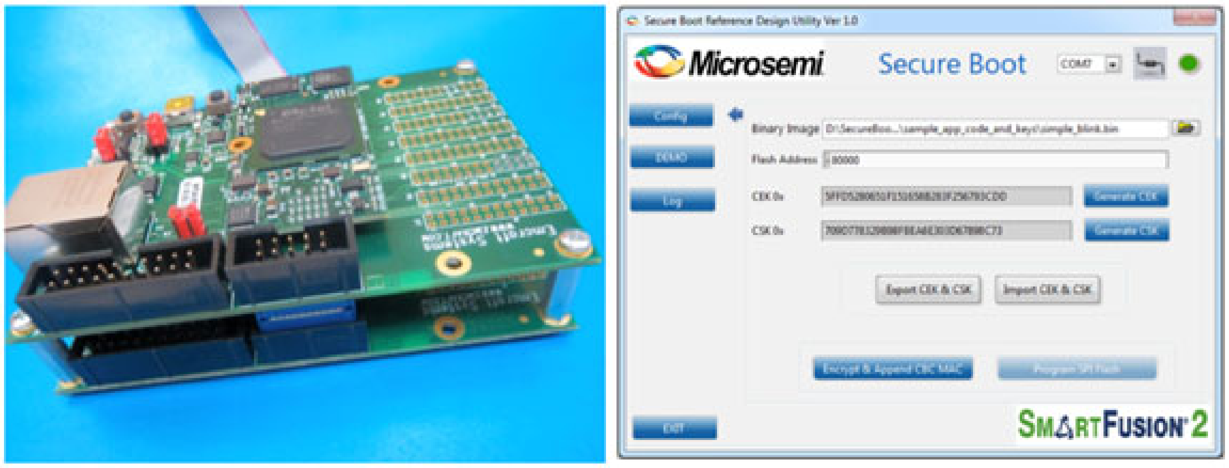
    After double-decryption, the SmartFusion2 SoC FPGA checks that the (partial) nonce is correct. If not, the target processor is forced into reset as a penalty.

11. The ESK is used to wrap the code signing keys, content encryption key (CEK) and content signing key (CSK). The code signing keys are used to encrypt and digitally sign the subsequent phase, the application code. The key wrapping is performed using the 128-bit AES key wrap algorithm as defined in RFC3394. The SmartFusion2 SoC FPGA performs the key wrapping and the wrapped keys are delivered to the target processor.
12. The target processor’s boot code unwraps the authentication and encryption keys (CSK and CEK) using the ESK.
13. The target processor’s boot code fetches the subsequent code, that is, the application code, from the external SPI flash through the SmartFusion2 SoC FPGA, and checks the validity of the enclosed signature using the CSK.
14. If not valid, the target processor halts and penalties can be applied.
15. If valid, the application code is decrypted using the CEK and executed. The multi-stage boot continues, with the SmartFusion2 SoC FPGA monitoring the process wherever it can. For example, the application code could generate a heartbeat by encrypting a counter with ESK.

The above description provides enough background on the key elements of the secure boot process and the reference design implementation for the purposes of this paper. More information on the operation of these elements of the secure boot reference design are available in the Secure Boot Reference Design User Guide, the URL of which is given in the "To Learn More" section at the end of this paper.

Reference Design Description and Operation

The Microsemi secure boot reference design uses two SmartFusion2 starter kits (SF2-484-STARTER-KIT) stacked on top of each other, as shown in Figure 11 below. The SmartFusion SoC FPGA development board serves as the target processor, with the SmartFusion SoC FPGA Cortex-M3 processor acting as the target, and the SmartFusion2 SoC FPGA development board acts as the Root-of-Trust, implementing all the previously described secure boot functions. Signals between the two boards use a simple GPIO connector that carries the Reset signal and the SPI interface signals between the target processor and the Root-of-Trust, as shown previously in the system block diagram of Figure 6 on page 9.



Binary Image D:\SecureBoo...\sample_app_code_and_keys\simple_blink.bin
Flash Address 80000
CEK 0x 5FFD52B0651F1516588283F256783CDD
CSK 0x 709D7783298B8FBEA6E303D67898C73

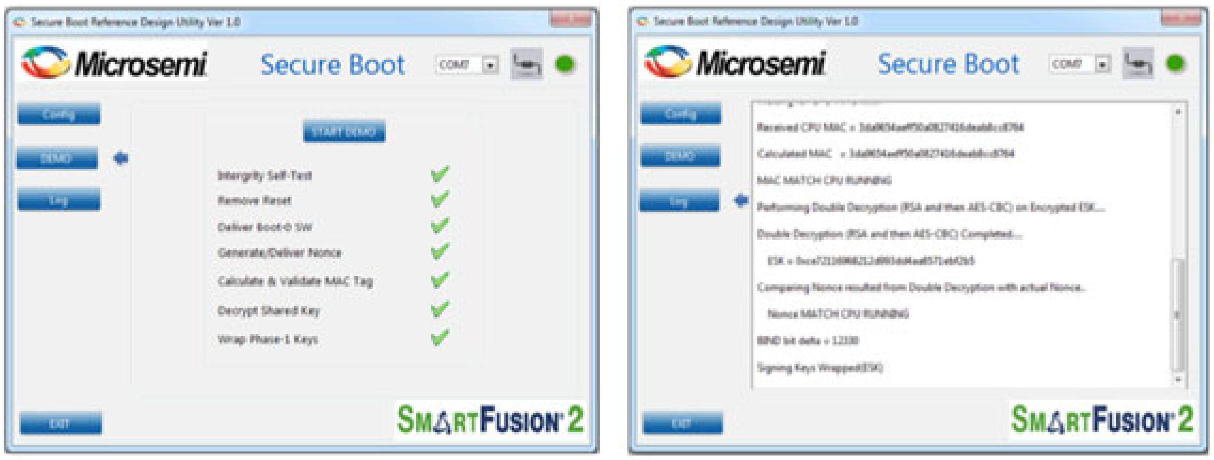
Figure 11: **Secure Boot Reference Design Hardware and Configuration Mode Screen**

The Microsemi secure boot reference design has several features that make it easy to observe and control the operation of key processes. The three main operating modes of the design are shown in the screen shot of the design utility user interface given in Figure 11.

The Configuration Mode allows the user to select a binary image for the secure boot code, identify the starting address in the flash memory for storing the secure boot code, create the keys to be used during encryption, encrypt and append the CBC MAC, and program the SPI flash memory with the encrypted secure boot code.

The Demo mode, the screen shot of which is shown in the left side of Figure 12 below, tracks the execution of the main elements of the secure boot process during execution. The user simply clicks the Start Demo button and begins the execution of the reference design. As the main elements of the secure boot process are executed, a green check mark appears at the appropriate element to show successful execution. A red cross appears if operation fails. The displayed secure boot operations are:

- Integrity Self-Test
- Remove Reset
- Deliver Boot-0 SW
- Generate/Deliver Nonce
- Calculate and Validate MAC Tag
- Decrypt Shared Key
- Wrap Phase –1 Keys



```
Intergrity Self-Test          ✓
Remove Reset                  ✓
Deliver Boot-0 SW             ✓
Generate/Deliver Nonce        ✓
Calculate & Validate MAC Tag  ✓
Decrypt Shared Key            ✓
Wrap Phase-1 Keys             ✓
```

```
Received CPU MAC = 3da9654aef950a0827416deabdcd784
Calculated MAC   = 3da9654aef950a0827416deabdcd784
MAC MATCH CPU RUNNING
Performing Double Decryption (RSA and then AES-CBC) on Encrypted ESK...
Double Decryption (RSA and then AES-CBC) Completed...
   ESK = 0xca7211698212d993dd4ea8571ebf2b5
Comparing Nonce resulted from Double Decryption with actual Nonce..
   Nonce MATCH CPU RUNNING
BIND bit delta = 12330
Signing Keys Wrapped(ESK)
```

Figure 12: **Secure Boot Demo and Log User Interface Screen Shots**

A log of all the key operations done during execution of the secure boot are available in the Log mode. A screen shot of the logged operations is shown on the right side of Figure 12. This log lists all the operations executed during the secure boot process making it easy to identify and track execution when considering the creation of a custom implementation.

Using the Reference Design Elements in a Customized Design

Many of the key firmware (processor code) and hardware (FPGA fabric implemented functions) elements of the secure boot reference design can be used as starting points for a custom implementation. For example, the SPI flash emulator, implemented in the FPGA fabric, can be easily reused for any target processor that requires an SPI memory for its boot process. The secure boot code is stored within the SmartFusion2 SoC FPGA eNVM or eSRAM memories and is supplied to the target processor during its boot phase.

The AES Controller, implemented in FPGA fabric, is also available as an IP core and implements the AES-256 encryption engine and the secure boot state machine. The controller has an AHB slave interface making it easy to connect to a customized SmartFusion2 SoC FPGA design.

The firmware used in the reference design to implement the secure boot process has several key functions that can be easily reused in a custom design. Of the functions executed on the SmartFusion2 SoC FPGA processor, some are available as system services. For example, nonce generation utilizes NRBG system services API calls. Other key functions that can be reused that execute on the SmartFusion2 SoC FPGA processor include the boot code signature calculation and validation function, the decryption and validation of the Ephemeral Shared Key (ESK), the wrapping application code signing keys function, and the SPI flash loader function. Several of the functions executed on the target processor can be reused as well. Some of the more obvious ones include the boot code signature calculation using the Microsemi WhiteboxCRYPTO library, the generation of the 128-bit ESK, the AES-CBC encryption function, the RSA encryption function, unwrapping the application code signing keys function, the application code validation function, and the decryption of the application code function.

More information on the reuse capabilities of the key elements of the secure boot reference design are available in the Secure Boot Reference Design User Guide, the url of which is given in the "To Learn More" section at the end of this paper.

WhiteboxCRYPTO Library Description

The Secure Boot reference design uses the Microsemi WhiteboxCRYPTO Library to implement the cryptographic functions executed on the target processor in a protected fashion. To protect encrypted information, it is imperative that the key never reveals itself in system memory. Standard implementations of AES and RSA leave both the algorithm and key vulnerable to tampering and reverse engineering. Microsemi's WhiteboxCRYPTO Library combines mathematical algorithms, data, and code obfuscation techniques to transform the key and the related crypto operations in complex ways requiring deep knowledge in multiple disciplines to attack. Hence, the key is *never* present in static or runtime memory. Rather, the key becomes an inert collection of data that is useless without the uniquely generated Whitebox algorithm that knows how to use that data to achieve the same output as the classical crypto counterpart. The uniquely generated Whitebox algorithm uses an accompanying encoded key and performs proper encryption, decryption, or verification of sensitive data without revealing the actual key.

WhiteboxCRYPTO comes in two crypto-algorithm variants: WhiteboxAES and WhiteboxRSA.

- WhiteboxAES decomposes a 128-bit, 192-bit, and 256-bit AES key and obfuscates AES encryption or decryption so that the keys are never revealed in the memory or on-disk.
- WhiteboxRSA encrypts, decrypts, signs, and verifies sensitive data while protecting 512, 1024, 2048, and 4096-bit RSA keys from discovery in memory or on-disk.

The WhiteboxCRYPTO Library allows you to:

- Generate a unique crypto library for each application shipped, and encode the same classical key for each.
- Produce a single library and encode many keys to work with it
- Encode multiple keys for multiple libraries by simply supplying an AES or RSA key, tweaking a few configuration variables, and running WhiteboxCRYPTO in any build environment.
- Easily generate new libraries which can be integrated into an application using a fully documented application programmer interface.

For more information on Microsemi's WhiteboxCRYPTO product refer to the link given in the "To Learn More" section at the end of this paper.

Conclusion

The Microsemi Secure Boot Reference Design provides a significant head-start for creating a custom secure boot implementation using the SmartFusion2 SoC FPGA as the Root-of-Trust for the secure boot process, since a significant number of key design elements are easily reused in a custom implementation. For example, the hardware design is based on SmartFusion2 and SmartFusion SoC FPGA development boards which have a significant amount of design IP freely available to reuse in a custom implementation. Additionally, many of the firmware (software routines executed on the Root-of-Trust or target CPUs) related functions are easily reused. For example, the key encryption, decryption and validation steps in the Root-of-Trust as well as the steps executed on the target, such as the WhiteboxCRYPTO functions, the application code validation and decryption functions. The combination of the SmartFusion2 SoC FPGAs best-in-class security capabilities and the extensive secure boot reference design provides a significant head start to any design that used the SmartFusion2 SoC FPGA as the Root-of-Trust for a secure boot process.

To Learn More

1. Secure Boot Reference Design
2. Securely Booting an External Processor with a SmartFusion2 SoC FPGA, User Guide
3. SmartFusion2 SoC FPGA Product Information Webpage
4. Microsemi Security Webpage
5. Microsemi WhiteboxCRYPTO Product Information Webpage
6. WhiteboxCRYPTO – Strength of Security Whitepaper

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996
E-mail: sales.support@microsemi.com

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for communications, defense & security, aerospace and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; security technologies and scalable anti-tamper products; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 3,400 employees globally. Learn more at www.microsemi.com.

© 2014 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.

55900177-1/6.14