Freescale Semiconductor
Application Note

Document Number: AN4235
Rev. 0, 06/2011

Using CSE to protect your Application Code via a Chain of Trust

by: Geoff Emerson, Jurgen Frank
Applications Engineering, Microcontroller Solutions Group

Contents

1 Introduction

This application note describes how some of the features offered by the Cryptographic Services Engine (CSE) module can be used to prevent application code from being altered by an unauthorized party. The CSE module has been integrated for the first time on MPC564xB/C. The module supports authentication of any kind of data and implements the security functions described in the Secure Hardware Extension (SHE) functional specification[1]. By reading this application note the reader must be able to get an insight into how to use some of the CSE features to prevent attacks on the application code. A GreenHills example project is presented which shows how some CSE features have been used to help prevent unauthorized application code changes. This application note introduces a number of concepts that can be used to prevent attacks on code and shows an example implementation.

It is not within the scope of this document to discuss the details of the SHE specification. This application note will focus on the hardware features provided by the CSE module and it is implied that the user is acquainted with the content of the SHE specification. It will be helpful if the reader is also familiar with the contents of AN4234 – Using the Cryptographic Service Engine (CSE): An introduction to the CSE module available from https://www.freescale.com.

1. SHE - Secure Hardware Extension functional specification Version1.1 (rev439) is copyrighted by the AUDI AG and BMW AG ©, 2008

Crytpography can be used to help authenticate messages and data. Since application code binary images are data, it is possible to use a Cypher based Message Authentication Code (CMAC) to authenticate application code. CSE provides a mechanism, called SECURE_BOOT, whereby a specified section of boot code can be authenticated at boot time. The boot code may contain code to authenticate a second section of memory against a CMAC value stored in system flash using the CSE. The second section of memory may contain code to authenticate a third section against another CMAC value stored in system flash, and so on. It is possible that these CMAC values have been previously derived off-line over the appropriate sections of code. Hence, a so called “Chain of Trust” can be established. This means that if required, the entire code flash space could be authenticated with only a small portion of it being authenticated at boot time, thereby decreasing the boot time if the CSE is configured to boot in sequential mode. Because the CSE is effectively a co-processor the rest of the application’s binary image can be authenticated later and in parallel with the main core’s tasks. If authentication fails the application may offer a reduced level of functionality, or set a flag so that at the next garage service the dealer will become aware that the software has been tampered with. CSE provides a selectable means to deny use of the keys stored in secure flash if the SECURE_BOOT process has failed. Provided the SECURE_BOOT process is allowed to complete before executing any potentially modified code, the application can have knowledge if the binary image has been modified.

This application note describes a process which has been used to build a flash image which implements this Chain of Trust approach. By understanding the technique described in this application note the user will be able to adopt the technique to their own requirements.

2 CSE

The Cryptographic Services Engine (CSE) is a peripheral module that implements the security functions described in the Secure Hardware Extension (SHE) Functional Specification Version 1.1. A block diagram of the crypto module is given in Figure 1.

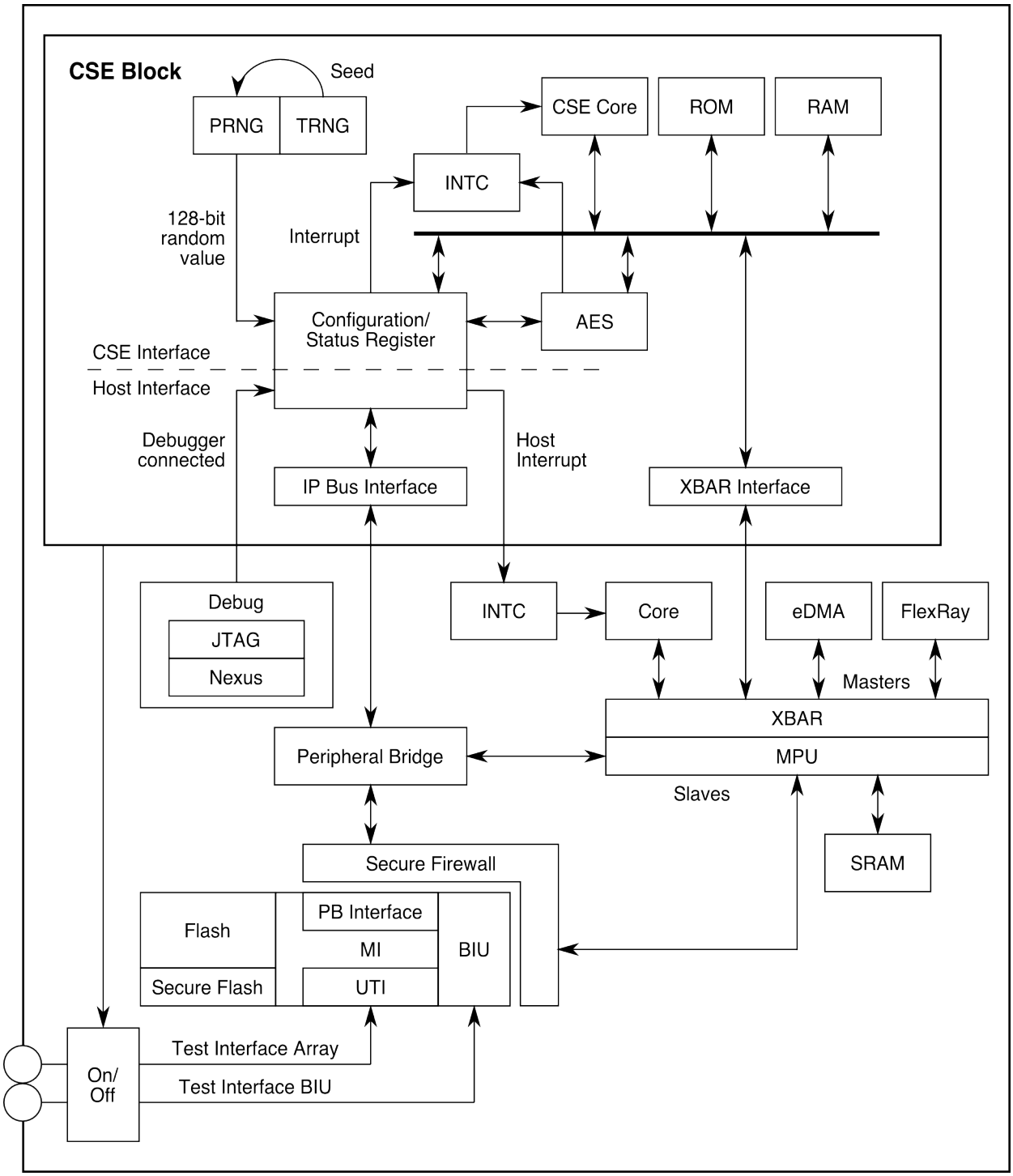


Figure 1. CSE module block diagram

The CSE design includes a host interface (via the peripheral bridge) with a set of memory mapped registers that are used by the CPU to issue commands. Furthermore a system bus interface (via the crossbar interface) allows the CSE to directly access system memory. Here the crypto module behaves like any other master on the Crossbar switch (XBAR). Through the host interface the user can configure and control the CSE module, like putting the module into low power mode, enabling interrupts for finished command processing or suspending command processing. A status and error register will give further system information. For a complete list of CSE commands refer to the MPC564xB/C reference manual which is available at https://www.freescale.com. Two dedicated blocks of system flash memory are used by the CSE for secure key and firmware storage. These blocks are not accessible by other masters from the system and therefore are called secure flash. The

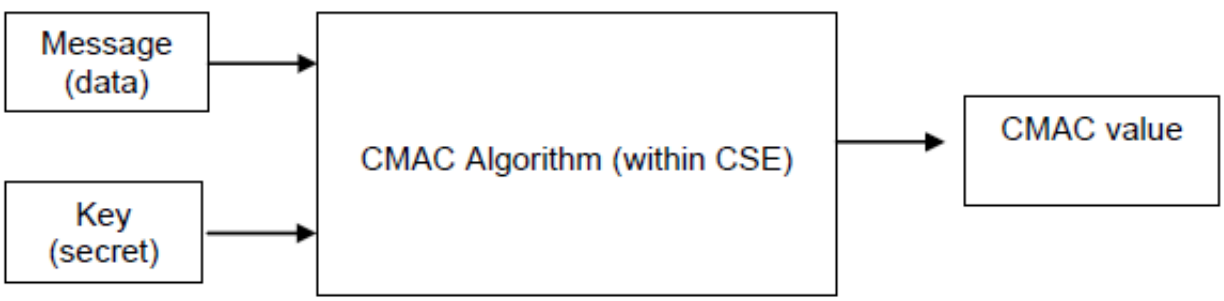
command processing is done by a 32-bit CSE core with attached ROM and RAM running at system frequency. After system boot the core comes out of reset and executes boot code from the module ROM. This code will load the firmware from the secure flash into the module RAM and start executing from there. This reduces the flash accesses by the crypto core on the Crossbar. The AES block is a slave to the crypto internal bus. It processes the encryption (plaintext → cipertext) and decryption (cipertext → plaintext) and offers AES CMAC authentication. This application note deals only with the authentication capabilities of the CSE.

3 CMAC and BOOT_MAC

3.1 CMAC

A CMAC provides a method for authenticating messages and data. A CMAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a CMAC. The CMAC value protects both a message's data integrity as well as its authenticity, by allowing verifiers (which also possess the secret key) to detect any changes to the message content. Figure 2 shows the components of a CMAC scheme.

Figure 2. Components of a CMAC scheme



In the case of CSE the CMAC output is always 128 bits. CSE provides a VERIFY_MAC command. This allows previously derived CMAC values to be confirmed.

3.2 BOOT_MAC

CSE has a mechanism which allows users to authenticate boot code in flash. The MCU can be configured so that on every boot a pre-defined section of code is authenticated and the generated MAC will be compared with a value previously stored in secure flash. This is supported only for flash boot modes. It is not supported for other boot modes (serial download, standby wakeup to RAM) as per the SHE specification.

The key used to authenticate the boot code is called BOOT_MAC_KEY. A value for comparison is stored in secure flash and is called BOOT_MAC. Extra information is added to the start of the boot block after the Reset Configuration Half Word. These parameters are provided to the CSE as inputs to a CMAC operation and define the location and size of the boot code to be authenticated. If the boot code is not authenticated, keys which are marked as boot protected cannot be used.

Table 1. Example of extra information added to the start of the boot block

Address	Content	Comment
0x0	0x15A	Reset Configuration Half Word

Table continues on the next page...

Table 1. Example of extra information added to the start of the boot block (continued)

0x4	0x10	Start address for BOOT_MAC calculation
0x8	0x1000	Length of code to be authenticated in bytes (4 KB)
0xC		This address is skipped
0x10	Code starts here	

In this example the boot code starts at 0x10 and CSE will authenticate 4 KB of code. Address 0xC is skipped because CSE can authenticate code significantly faster if authentication starts on a 64-bit boundary. Please refer to AN4234 – Using the Cryptographic Service Engine (CSE):An introduction to the CSE module, for details of how to program the BOOT_MAC and BOOT_MAC_KEY.

Using the BOOT_MAC and CMAC of the CSE it is possible to build applications which are capable of detecting that the code has been tampered with. An example scheme for doing so is described in the following section.

3.3 Building code suitable for of the Chain of Trust Scheme

The following section describes a scheme which has been successfully implemented and tested on silicon. A sample GHS project has been created for demonstration purposes. This is available in AN4235SW, which can be downloaded from https://www.freescale.com. User of other compilers will need to port the code. The video mentioned later will demonstrate the principle to those without a Lauterbach probe. An example perl script which has been used in the binary image generation, is available in AN4235SW. Note that this is for demonstration of the concept being discussed here only and is not intended for production purpose. In order that the Chain of Trust can be implemented, linker sections must be defined in the project linker file at build time.

For example:

```
.my_app_code1_section_all ALIGN(0x8) : {*(.my_app_code1_section_rodata)
*(.my_app_code1_section)} > int_flash
```

This defines a section where the code and any constant data for a particular code function can be located.

NOTE

In general the process for only one section of the code will be shown as an example. In the sample project there are three sections of code. Only the first will be shown.

The linker file must also export values concerning the start address, end address and length of these section.

For example:

```
CMAC_APP_CODE1_START_ADDR = ADDR(.my_app_code1_section_all);
CMAC_APP_CODE1_LENGTH = SIZEOF(.my_app_code1_section_all);
CMAC_APP_CODE1_LENGTH_IN_BITS = 8 * SIZEOF(.my_app_code1_section_all);
CMAC_APP_CODE1_END_ADDR = (CMAC_APP_CODE1_START_ADDR + CMAC_APP_CODE1_LENGTH) ;
```

These values will be used both by an offline utility to generate a CMAC and later at run time by the application when it will verify that the binary image in the section has not changed since compile time. In the C-source file code is placed into particular sections using a #pragma compiler directive.

For example:

```
#pragma ghs section rodata=".my_app_code1_section_rodata",text=".my_app_code1_section"
```

This means that both the constant data and code binary which is generated from the following source code will be located in the section called my_app_code1_section_all.

CMAC and BOOT_MAC

A CMAC for each of the sections must be generated by an offline computer using a key. Later at run time the offline generated values can be compared to the values generated by the CSE over the same binary data using the same key. In order to derive a CMAC offline a key must be used. Each section of binary data will be authenticated by a different key. This is user definable. A file containing section names and keys is used to tell both the offline tool, and later the CSE, which key is used for each section.

For example (contained in file SECTION_KEYS.txt)

```
CMAC_APP_CODE1 CSE_KEY_7
CMAC_APP_CODE2 CSE_KEY_7
CMAC_APP_CODE3 CSE_KEY_10
```

In this case the binary data in section 1 and 2 will both be authenticated using KEY_7 and the binary data in section 3 will be authenticated using KEY_10. It will be shown later that these keys can have different properties. The values of the keys being used are defined in a text file. For example (contained in file KEYS.txt).

```
CSE_KEY_10 68B674CB8198A2503A285100F4DDC40A
CSE_KEY_7   2b7e151628aed2a6abf7158809cf4f3c
```

Key Numbers are defined in a text file also. These are used by the application code to verify that the code has not changed since compile time. This file maps the user key string unto the CSE's Key_ID value. For example (contained in file CSE_KEY_INDEX.txt).

```
CSE_KEY_7          10
CSE_KEY_10         13
```

Note that these values must match the values given in the Memory Slots table in the CSE chapter of the MPC564xB/C reference manual.

Prior to the next step the application must be compiled. A perl script will extract location and size information for each of the sections from the .map file produced by the compiler. This information will in turn be used to extract the required binary data for each of the sections from the s-record file produced by the compiler. In the case of the sample project these are .run files.

To extract the binary data for each of the section a utility called srec_cat is called. This is called from within a perl script and uses the various files discussed above to determine which data gets extracted and written to particular files for further processing. For example, the following command is generated and executed by the perl script:

```
srec_cat.exe Flash.run -fill 0xFF -within Flash.run -range-padding 4 –crop 32768 32972 -
output CMAC_APP_CODE1.hex –vmem 32 -line-length=46
```

The srec_cat utility is available for download from the internet. Version 1.47.D001 was used in the example project development.

Note that 32768 and 32976 are the literal values of CMAC_APP_CODE1_START_ADDR and CMAC_APP_CODE1_END_ADDR as defined above. The output is formatted to have line length of 46 characters. This is because the output file (CMAC_APP_CODE1.hex in this case) is re-read by the next part of the script.The –fill 0xFF and range-padding 4 options are required to pad the image as it is VLE code and if the last word in the range is a 16-bit instruction it needs to be extended by 16 bits (of "erased flash" value) in order that the CMAC can be verified correctly by the CSE.

The CMAC for each of the output hex files can now be derived. The file output from by the srec_cat utility is stripped of addresses and spaces. A CMAC utility is used to derive the CMAC. As input it requires the data, length of the data in bits and the key value. A perl script is used to call the CMAC utility.

For example, the following command is generated and executed by the perl script:

```
AES_CMAC_CMD.exe 2b7e151628aed2a6abf7158809cf4f3c 1664 CMAC_APP_CODE1_128.hex
```

(AES_CMAC_CMD.exe is provided in the software release AN4235SW). Here 2b7e151628aed2a6abf7158809cf4f3 is KEY_7 which is associated with CMAC_APP_CODE1. 1664 is the binary length in bits. This produces the CMAC value for the section of binary in question. The CMAC value is also written by the perl script to a header file (called keys_and_cmacs.h) and eventually gets included in the build of the boot code section. The boot code section can be protected by the BOOT_MAC and secure boot process.

For example (contained in keys_and_cmacs.h)

```
#define CMAC_APP_CODE1_0 0xc260efe7
#define CMAC_APP_CODE1_1 0xb830769d
#define CMAC_APP_CODE1_2 0x80052766
#define CMAC_APP_CODE1_3 0xe142fbd2
#define CMAC_APP_CODE1_KEY_NUMBER CSE_KEY_7
#define CMAC_APP_CODE1_START_ADDR 32768
#define CMAC_APP_CODE1_LENGTH_IN_BITS 1664
```

Notice that the key number is also written to this file along with the start address and code length values. These can be used by the application at run time to verify the code has not been changed since it was compiled.

The boot code will be authenticated at boot time by the secure boot process. If the keys authenticating the application functions are marked as boot protected (BP=1) then, if the boot protected code is somehow altered, CSE will not allow access to those keys and the verification command will return CSE_KEY_UNAVAILABLE. If the keys authenticating the application function are not marked as boot protected and the function binary is somehow altered the verification command will not be successful as the CMAC will not match the value stored in system flash.

Given below is a example of an application function which will check the authenticity of the next section before it is executed.

```
uint32_t my_function_code1(uint32_t parameter)
{
    vuint32_t result1, CMAC_mismatch;

    unsigned long long length = CMAC_APP_CODE2_LENGTH_IN_BITS ;
                                                /* function code length in bits*/
    failcount=0; CMAC_mismatch=0;

    CSE.P1.R = CMAC_APP_CODE2_KEY_NUMBER; /*  */
    CSE.P2.R = (unsigned long long)&length; /* msg length */
    CSE.P3.R = CMAC_APP_CODE2_START_ADDR;
    CSE.P4.R = (vuint32_t)&my_app_code2_build_cmac;
    CSE.P5.R= 128; /* verify all 128 bits */
    CSE.CMD.R= CSE_VERIFY_MAC;

    while (CSE.SR.B.BSY ==1){} /*wait until CSE is idle*/

    if (CSE.ECR.R != CSE_NO_ERR) {failcount++;}
    if (CSE.P5.R != 0) {CMAC_mismatch++;}
    if (CMAC_mismatch != 0)
    {
      function_code2_bad = 1;
    }

  result1 = parameter*parameter; //or whatever - application code
  return(result1);
}
```

In this example the application code sets function_code2_bad = 1 if the memory section containing my_function_code2 is not authenticated. Altering the binary image associated with the my_function_code2 will make this check fail.

The scheme is represented pictorially in Figure 3. Note that this image can be zoomed in so the detail is clearer. Note also that the compilation step needs to be performed twice. On the first compile the boot code is compiled with potentially incorrect CMAC constant values. The first compile step is required to generate the binaries for the sections protected by the CMACs so that the address and length information can be extracted. These CMACs can then be derived based on the map and run file. On the second compile the boot code will definitely have the correct CMAC constant values available to it.

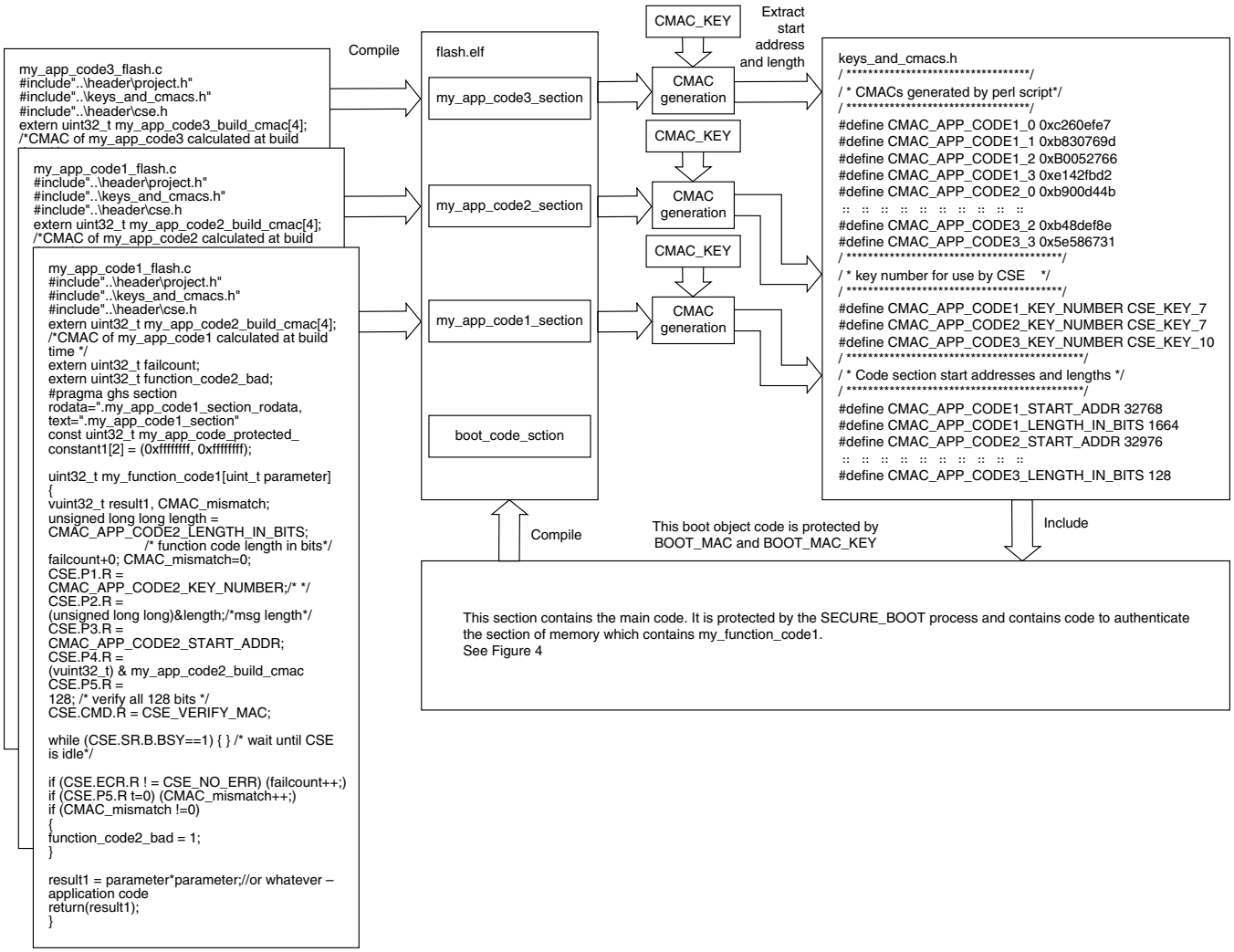


Figure 3. Summary of the scheme for generating code suitable for the Chain of Trust

```
#include"..\ header \ project.h"
#include"..\ keys_and_cmacs.h"
#include"..\ header \ cse.h"
"pragma ghs section rodata = ".my_boot_code_section_rodata", text=".my_boot_code_section"
/ * CMACs of my_app_code *.c calculated at build time */
const uint32_t my_app_code1_build_cmac[4] = {CMAC_APP_CODE1_0, CMAC_APP_CODE1_1,
CMAC_APP_CODE1_2, CMAC_APP_CODE1_3};
.. .. .. .. .. .. .. .. .. ..
const uint32_t my_app_code3_build_cmac[4] = {CMAC_APP_CODE3_0, CMAC_APP_CODE3_1,
CMAC_APP_CODE3_2, CMAC_APP_CODE3_3};
const uint32_t Boot_Protected_Constant[2] = {0xffffffff,0xffffffff};
uint32_t failcount = 0; uint32_t function_code1_bad = 0; uint32_t function_code2_bad = 0;
uint32_t function_code3_bad = 0;
vuint32_t r1, r2, r3;
int main(void)
{
unsigned long long length = CMAC_APP_CODE1_LENGTH_IN_BITS;
vuint32_t CMAC_mismatch; r1=0, r2=0, r3=0;
while ((CSE.SR.B.BOK == 0) & (CSE.SR.B.BFN == 0)) { }
/ * wait until CSE is has finished Secure Boot process */
/ * issue_BOOT_OK */
CSE.CMD.R = CSE_BOOT_OK;
while (CSE.SR.B.BSY == 1) { } / * wait until CSE is idle */
while (CSE.SR.B.BFN! = 1) { } ; / * trap if Secure Boot Finished is not set */
/ * BFN will be set because either the SECURE_BOOT passed and BOOT_OK sets it or
SECURE_BOOT failed and set it */
failcount = 0; CMAC_mismatch = 0;
CSE.P1.R = CMAC_APP_CODE1_KEY_NUMBER; / * */
CSE.P2.R = (unsigned long long) & length; / * msg length */
CSE.P3.R = CMAC_APP_CODE1_START_ADDR;
CSE.P4.R = (vuint32_t) & my_app_code1_build_cmac;
CSE.P5.R = 128; / * verify all 128 bits */
CSE.CMD.R = CSE_VEERIFY_MAC;
while (CSE.SR.B.BSY == 1) { } / * wait until CSE is idle */
if (CSE.ECR.R! = CSE_NO_ERR) {failcount++;}
if (CSE.P5.R! = 0) {CMAC_mismatch++;}
if (CMAC_mismatch! = 0)
{
failcount++; / * the code for my_function_code1 was not authenticated */
function_code1_bad = 1; / * app code to deal with that */
}
        r1 = my_function_code1(2);
        r2 = my_function_code2(2);
        r3 = my_function_code3(2);
while(1);
}
```

Figure 4. Main.c

4 Using the sample perl script

The sample perl script is included in the software package AN4235SW.

Example use:

```
perl generate_MACS_over_object_code.pl -m Flash -s SECTION_KEYS.txt -k KEYS.txt
```

Following –m is the name of the run and map files which are produced by the GreenHills compiler. Following –s is the name of the file which define both section names and their corresponding key names. Following –k is the name of the file which contains the key values. An additional file called CSE_KEY_INDEX.txt is also required but is not specified in the script parameters. This is because there is no need for it's content to change.

5 Getting the example project working on silicon

A Lauterbach *.cmm script is supplied in AN4235SW which will guide the user through the steps required to see the benefits of using the techniques described in this application note. The Lauterbach script is called Chain_Of_Trust_demo.cmm. The Lauterbach should be configured in single window mode for best results. Users will need to change the path variable in the first line of the script to point to the directory containing the cmm script. A short video demonstrating the example project has been created. This will prove useful for readers without access to either a GHS compiler and/or a Lauterbach debugger. Users need to click on the yellow box in the bottom right hand corner when they have read the information which has been presented. The video is presented at http://www.youtube.com/user/freescale.

6 Making it even harder for attackers

Each CSE module has a unique identifier (UID) stored in its secure flash. This is programmed by Freescale during manufacture. It is possible to make it so that CSE modules have unique keys stored in secure flash, so that no two parts contain the same set of keys. By maintaining a database of keys referenced by UID, manufacturers have the ability to make it more difficult and expensive to attempt to attack systems protected by CSE. The effort required to establish the value of the keys in secure flash is likely to be enormous. However if manufacturers make the keys unique, then even if it is possible to discover the keys on a single CSE module this information will be useless in assisting attacks on other systems. This approach also has the advantage that the effects of a disgruntled employee disclosing non-unique keys are potentially going to be vastly reduced.

7 Conclusion

This application note has shown how to use the CSE features to authenticate code via a chain of trust. A sample scheme has been presented and this should provide users with some insight into how to go about designing software which can be protected from modification using the features of the CSE. The mechanism described allows boot time to be minimised if the CSE is configured in sequential mode.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN4235
Rev. 0, 06/2011