Trusted Firmware in Embedded Systems

Isaac Garfield
Department of Electrical and Computer Engineering
Brigham Young Univeristy
Provo, UT, USA

***Abstract*—This work presents a simple implementation of signed firmware to verify the authenticity of software running on the NUCLEO-L433RC-P. It explores implementations in C and in ARM Assembly.**

I. Contents

II. Introduction

SECRUITY operates around the principal of a root of trust. Trust is bestowed upon some software not implicitly, but because we know that we can trust someone who trusts that software. At the beginning of this trust chain is a ”Root of Trust”, some software or service that must be trusted inherently. Within computer systems, the root of trust is the bootloader, the very first part of the system to execute code. If someone can compromise the bootloader, they have free range over the entire system. However, with a working trusted bootloader, we can insert and trust other protections against insecure systems and components to protect the whole system. In this project I will replace the first part of the system to run on a STM32 chip with a ”secure” option.

III. Background

A. Security

While all modern computer systems have some security built in, to get a sense of the field we'll look at Apple. The first code to execute on an iPhone or iPad is from the Boot ROM, written during chip fabrication, and implicitly trusted. This Boot ROM contains the Apple Root Certificate Authority Public Key, which verifies the iBoot bootloader. Afterwards, every stage of the bootloader verifies the authenticity of the following stage before running it. (https://support.apple.com/guide/security/boot-process-for-iphone-and-ipad-devices-secb3000f149/1/web/1).

Apple also provides regular secure software updates for their hardware. The devices verifies the server using a certificate authority and the software package using a SHA-256 hash. The server verifies the device using unique specifiers, including a Exclusive Chip Identification (ECID). (https://support.apple.com/guide/security/secure-software-updates-secf683e0b36/web).

Apple signs only the latest public iOS version, making downgrading from a public release impossible. Apple's security restrictions also prevent sideloading of applications or OS software. (https://discussions.apple.com/thread/256083412).

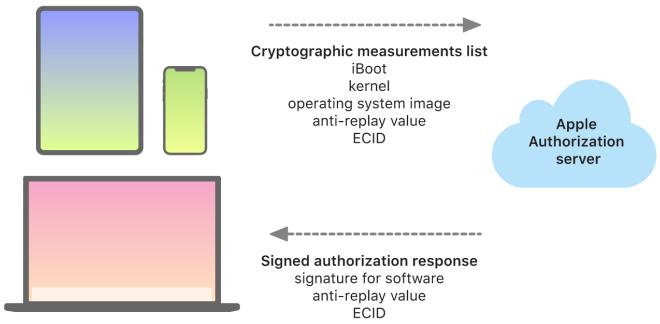

Fig. 1: Apple Software Update Trust Model

B. Our Hardware

For this project, I use the NUCLEO-L433RC-P, a development board for the STM32L433 series chips. The part is a Low Power Microcontroller with a 32-bit ARM-Coretex M4 processor. The STM32L433xx includes 64KB of SRAM and 256KB of Flash memory. We reset this device with a push button or over USB. We get data or feedback from this device primarily via an LED indicator or UART connection.

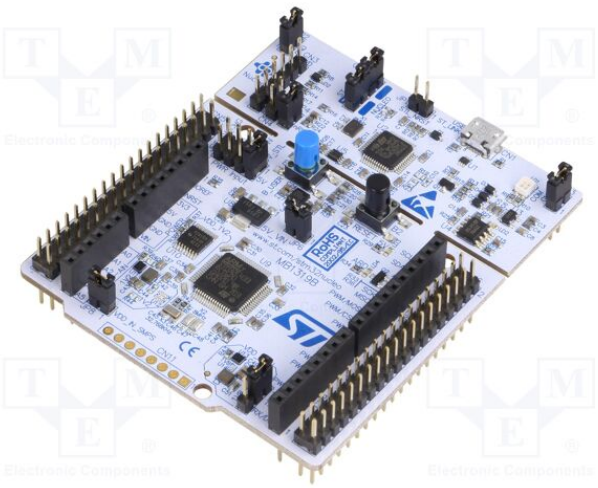

Fig. 2: NUCLEO-L433RC-P

C. Our Software

STMicroelectronics offers a configuration tool and code generator called STM32Cube. This code generator creates a basic project structure including a main function (in C), a Hardware Abstraction Layer (in C), a linker script (ld file), and a startup script (ARM asm). The main function is where we normally write our code. The linker script can be used to change memory layout, heap and stack sizes, and the program entry function. The startup script includes everything the processor needs to do to create a platform for the C program to run.

When a reset signal is received, the program counter (pc) is set to 0x8001c9c. By default, 0x8001c9c contains the `Reset_Handler` which calls all the necessary routines to put the processor into a known state. This includes setting stack pointer, the interrupt table, and the bss region. Once initialization is finished, the main() function is called. Inside main we can call the HAL provided functions to initialize peripherals like UART, Counters, System Clocks, and LEDs. We then have a working platform and connected peripherals to run whatever software we want, like blinking an LED.

IV. Investigation

The very first thing I did was try to understand the system I was working in. This meant using gdb and objdump to find where the program was starting and what it was doing. By the power of looking it up, I found that the reset handler would be the first thing that was called, so I found where that was called. I investigated the order and operations of functions to try to find the best place to insert some verification logic. I found two potential spots, the first would be to write some code in `Reset_Handler`, and the second would be to write some code immediately before the branch to main().

V. Design Part One - C

I then realized that I should probably get something, anything, to work before I jumped into my ideal result. So I closed the objdump output and exited gdb. I jumped to the C main file. I wanted to create a signature I could store somewhere known in memory and to compare against it and either continue execution or exit the program. I know that string literals exist somewhere in memory, which means that if I put a string literal on the first line of the program, I should be able to find that in a disassembled program. I inserted `char a[] = "Signed";` as the first line of my C program, and compiled. I then ran objdump on the output to find where that string was saved and found it in the `.data` section. Linker scripts allow you to access variables with a C extern, so we used `extern const char _sdata;` to get the symbol, and took its address to get a pointer to the beginning of the data section. I set two `uint32_t` values to the known values of the string. These were local variables, so they were created on the stack, rather than in the `.data` section.

I then compared the memory contents of `.data`, to a local hex representation of the string. This worked well to determine that the firmware was signed, and I wanted to improve on it.

```
63  void check_str() {
 1      // Correct big endian str
 2      uint32_t v1 = 0x6e676953;
 3      uint32_t v2 = 0x000a6465;
 4
 5      extern const char _sdata;
 6      uint32_t* sdata_ptr = (uint32_t*)&_sdata;
 7
 8      if (v1 == sdata_ptr[0] && v2 == sdata_ptr[1]) {
```

Fig. 3: Comparison in C Code

VI. Design Part Two - ARM Asm

The first thing to change was to store the signature string literal in main.c some other way. This was partially a response to the desire to use other string literals. When I added string literals for Uart printing, it moved the signature I cared about around within the data section. I had a few options of how to store a value in memory and read it and compare it against some local or immediate value.

1) Option A: Store Linker, Read C
2) Option B: Store ASM, Read C

3) Option C: Store Linker, Read ASM
4) Option D: Store C (String Literal), Read ASM

A. Store

We used Store C (String Literal), read via linker symbol in C (Read C) in Part One. For part two I wanted the comparison to happen as soon as possible after startup. If I stored data with the linker directly, then it would be present in memory as soon as the program was loaded. That gives me the entire execution of the program to make my comparison. My best option was Option C, to store with the linker, and to read from ASM.

The linker script provides a handy utility which allows you to store specific data at specific points. So I took my string "Signed", converted it to a little endian hex number, and stored it as a QUAD (8 bytes). This worked great, I could find the variable and compare against it in my C code.

B. Load and Compare

I then proceeded to work on running my comparison in assembly, before main was called. The provided `startup.s` file is where `Reset_Handler` is defined. I added `verify_boot` to the startup script. (Here, my initial project stopped letting me insert breakpoints before main was called. If they were inserted, the would never be activated. Switching to a new project worked well initially, but eventually had gdb issues too.) This script loaded `_sdata` into a register, and filled other registers with the known comparison values. It performed the comparison, then jumped either to `Reset_Handler` or `Infinite_Loop`. Once `verify_boot` was added to the startup script, I updated the linker script with `ENTRY(verify_boot)`.

As we can see in Figure 5, we sucessfully loaded into `verify_boot`. Here I ran into issues that aren't fully resolved. Debugging assembly with gdb is not going smoothly because register values don't match what I expect, and single stepping is not working correctly. This could potentially be due to using the wrong file format (the board is programmed with .bin, gdb works with .elf). But it's probable another system is not working appropriately.

```
57  /**
 1   * @author: Isaac Garfield
 2   * @brief: Check first 8 bytes of _sdata matches KEY val
 3   *         Go to Infinite_Loop or Reset_Handler on answer
 4   *
 5  */
 6      .section    .text.verify_boot
 7      .weak   verify_boot
 8      .type   verify_boot, %function
 9  verify_boot:
10    ldr r3, =_sdata
11  /*  ldr r3, =0x00000020 /* Just pull the variable from linker */
12  /*  ldr r3, =0x20000000 */
13
14    ldr r1, =0x6e676953
15    ldr r2, [r3]
16    cmp r1, r2
17    bne Infinite_Loop
18    ldr r1, =0x000a6465
19    ldr r2, [r3]
20    cmp r1, r2
21    bne Infinite_Loop
22
23    b Reset_Handler
```

Fig. 4: ASM Function

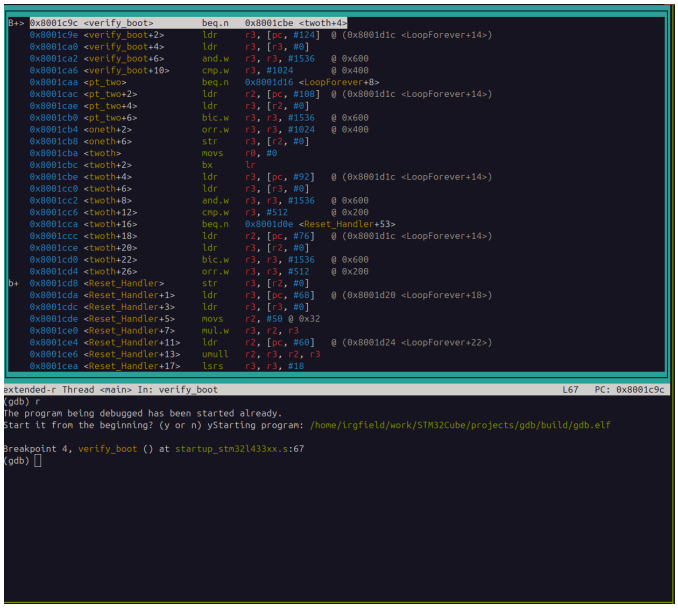

Fig. 5: Verify Boot Running First

VII. Lessons Learned

1) The reset button does very little. It mostly just sets the pc to the `Reset_Handler`, which does everything else.
2) The Linker defines the ENTRY of program, which the program uses on startup.
3) The `Reset_Handler` is defined and written in assembly for each STM32 product line.
4) ARM is not one instruction set, it's a whole family with many variants.
5) GDB has a hard time inserting breakpoints before main is called, with mixed success.
6) QUAD and LONG statements in a linker allow canaries to be placed at arbitrary locations.
7) Make doesn't detect LDSCRIPT changes and wouldn't automatically update code, so everything had to be cleaned and re-run.
8) Assembly is hard, and designing your own project is a challenge. But a good challenge.

VIII. Conclusion

I don't consider the ARM design fully functional because I can't prove that it works, although some of my tests appeared to function correctly, but I believe this project was successful because I gained huge understanding of how this microcontroller actually works, and what a bootloader needs to do. I also learned what linkers actually do and how program memory really works. One of the most interesting thing is that I doubt the validity of some of my gdb experiments. I think there may be some observer effects that cause undefined behavior.

Hardware security encompasses our trust in the devices we have and the secrets they possess. The ability to guarantee that the bootloader functions correctly is problem with million or billion dollar solutions for large companies. In this project I researched how we can guarantee that the software we run is the software we hope to allow to run, and built software which would restrict the usability of this device to those who are able to produce signed software. In the future I hope to continue this project and resolve the issues with the assembly implementation.