A Survey of Firmware Analysis Techniques and Tools

A Thesis
Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Science
with a
Major in Computer Science
in the
College of Graduate Studies
University of Idaho
by
Bradley A. Whipple

May 2020

AUTHORIZATION TO SUBMIT THESIS

This thesis of Bradley A. Whipple, submitted for the degree of Master of Science with a Major in Computer Science and titled "A Survey of Firmware Analysis Techniques and Tools," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	____________________ Michael Haney, Ph.D.	Date: __________
Committee Members:	____________________ Konstantinos Kolias, Ph.D.	Date: __________
	____________________ Robert Hiromoto, Ph.D.	Date: __________
Department Administrator:	____________________ Terence Soule, Ph.D.	Date: __________

ABSTRACT

This thesis attempts to cover many aspects concerning analysis, reverse engineering, and provenance attribution of firmware from embedded devices. The intended reader of this thesis is someone familiar with, or at least aware of, the software build process, the analysis of software, and reverse engineering of software. This thesis discusses some of the differences between firmware and traditional software and may serve as a bridge for those readers that may be more familiar working in a software environment and are interested in analyzing embedded devices. The thesis will include strategies for retrieving firmware binaries from a target device, reverse engineering with the intent to provide provenance information about the firmware, and briefly cover future work of using machine learning to analyze firmware.

ACKNOWLEDGMENTS

Thank you to the professors at the University of Idaho for their efforts in advanced education and research. Thank you to Alice Allen for her uncanny ability to cut through red tape and guide students through bureaucracy. Thank you to Idaho National Laboratory for enabling me to pursue further education by allowing me to work part time professionally while taking classes full time.

DEDICATION

This thesis is dedicated to my lovely wife who supported me during my pursuit of a higher education. She endured my late nights studying, working, and running a business with patience and love.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1: INTRODUCTION

Note: In this thesis software refers to the personal computer (PC)/desktop environment and the terms are used interchangeably. This thesis also uses embedded and firmware interchangeably to refer to code or devices often removed or abstracted from the end user such as smart thermostats or IOT security cameras. The author recognizes the term embedded is used loosely and there is a lot of gray area between a PC system and a typical embedded system.

Software and firmware analysis in the context of this thesis is the process of acquiring, reverse engineering, and otherwise figuring out what makes a device 'tick'. In other words, its cracking open the hood of a device and figuring out what's inside either by targeting specific information suspected to be within, or by a sweeping drag-net approach; the method being situation dependent. Software and firmware analysis are conducted for a myriad of reasons but the top three are identifying vulnerable libraries, enforcement of licenses and/or intellectual property protection, and attribution classification.

By its nature, software is inherently difficult to analyze. Information available to the researcher performing analysis is often absent or obscured. The process of going from code to device/executable strips information out and given that analysis is performed almost exclusively without code it leaves the researcher with a reduced set of data/information. It is then up to the researcher to piece together a larger picture from small snippets of information. To add to the arduous task the data available to the researcher may have been obfuscated intentionally by the original coder making the analysis even more difficult.

In comparison to software analysis, the analysis of firmware is even more difficult due to the extra layers of obscurity involved. While PC based environments are often limited to a few different processor architecture's and operating systems with code being restricted to running within those environments; embedded systems are more diverse. The number of different processor architectures and design paradigms found within embedded systems is far greater than that of PC environments. Due to the lack of a common design schemas, processor architectures, or operating systems the process of analysis on an embedded system is more complex.

Frameworks have been attempted to structure and organize the firmware analysis process but the problem is often too complex and diverse to solve with a series of regimented steps and tools. Analysis frameworks the author has investigated target a very specific scenario and often combine a series of tools in an attempt to streamline the process. Attempts at creating toolkits or frameworks for analysis overlook a key component to an analysis campaign which is intuition and researcher creativity.

This thesis covers attempts to cover some of the tools used in firmware analysis. This thesis then offers a demonstration of the tools listed as a brief expose' into what is involved during a firmware analysis campaign as well as the diversity of what can be encountered. This thesis will discuss some potential areas of future research and why attempts of developing an analysis framework fail.

CHAPTER 2: COMPARISON OF SOFTWARE TO FIRMWARE

2.1. Background/Intro

This chapter covers some core differences between embedded systems and traditional PC systems that readers may be familiar with. Much of this thesis focuses on the C environment for the sake of comparison. Most firmware, albeit not all, is written in C [1] and while there may be some analysis techniques unique to the output of other programing languages on a PC those will not be discussed as they are out of scope.

2.2. Hardware differences

This chapter and subsequent subchapters discuss some of the hardware differences between a traditional software environment and an embedded environment. While this has somewhat changed in the past ~5-10 years, typically engineers working with a PC platform could remain unaware of the hardware they were working with and conversely an engineer working in an embedded environment had to be intimately aware of all aspects of the hardware. As technology advances and systems become more complex there is a push to further abstract hardware and other layers of complexity so even some dedicated firmware engineers may be unaware of hardware specific details of the device they are supporting or working on. Recent cyber-attacks and vulnerabilities such as Spectre and Meltdown show the importance of being 'hardware aware' [2] even when the scope is restricted to PC systems.

Software engineers often write processor agnostic code and are more concerned with the operating system that will host their application. The reason for the abstraction of hardware is there are many software layers between the hardware and the application. When engineers are working on an embedded device, they often don't have the luxury of having all the inner workings of the CPU, peripherals, memory, etc. abstracted from them through layering of code (although this is changing).

2.2.1. Processor

There are some significant differences between traditional PC based processors and embedded specific processors that extend beyond performance and power consumption. Anecdotally software engineers traditionally do not need to concern themselves with processor architecture [3]; the concern lies within what runtime environment and operating system they intend to design to. When reverse engineering a piece of code that was intended to run on a PC platform the approach is often agnostic to what processor the code is running on.

The above points are not true when it comes to embedded systems. Engineers must not only be capable of understanding code but have an in depth understanding of the memory interface, memory map, user manuals describing registers, and more [4], [5]. Often on an embedded platform memory, storage, peripherals, etc. are all internal to the processor itself with details about specifications to the processor embedded in the part number that is laser etched on top. Referencing the datasheet for the specific processor an engineer/researcher is working on is commonplace when dealing with an embedded target in both development and reverse engineering. Take for example the memory map below in Figure 2-1. This memory map describes the how memory is allocated internally in the processor.

Figure 2. Memory map

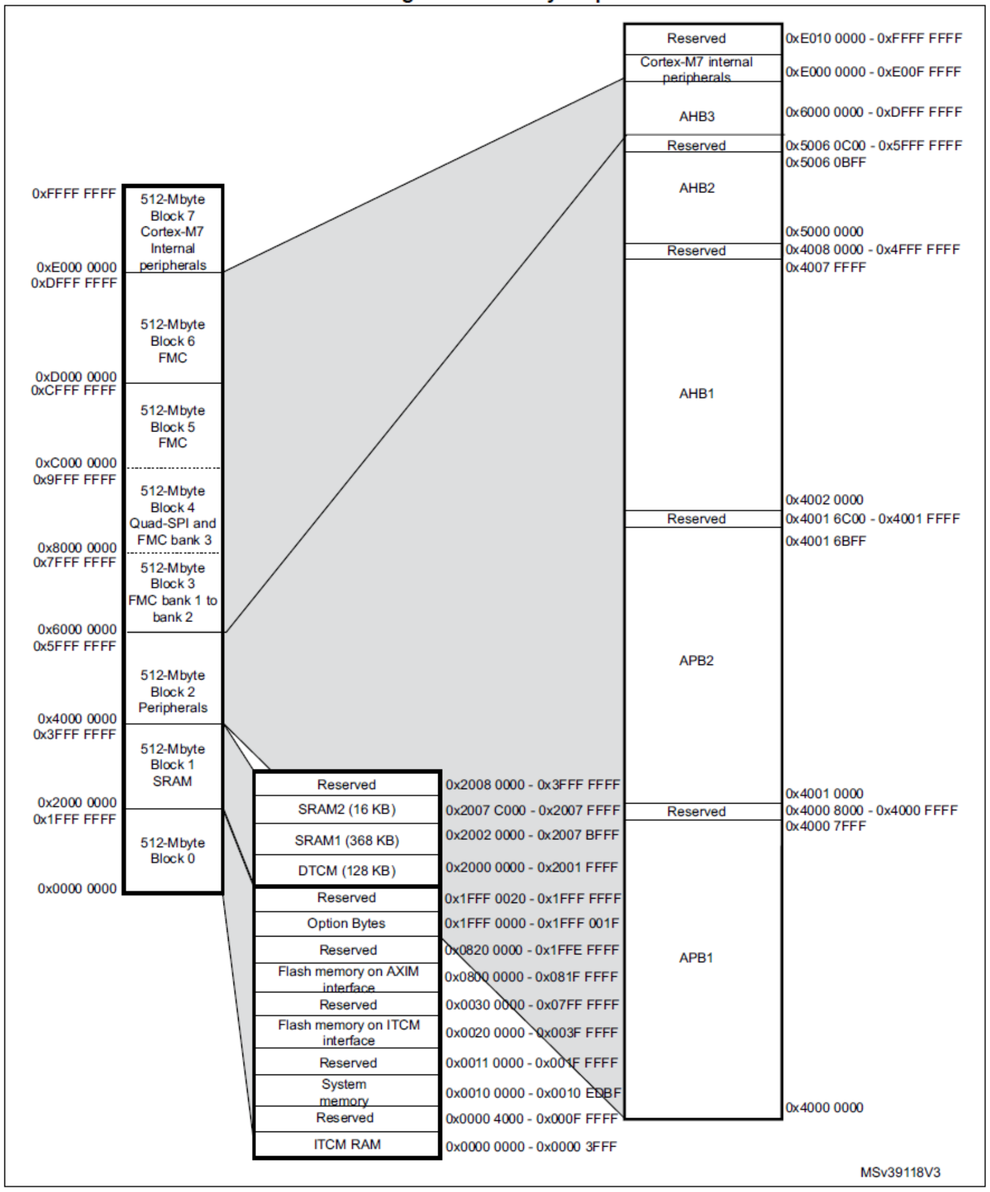
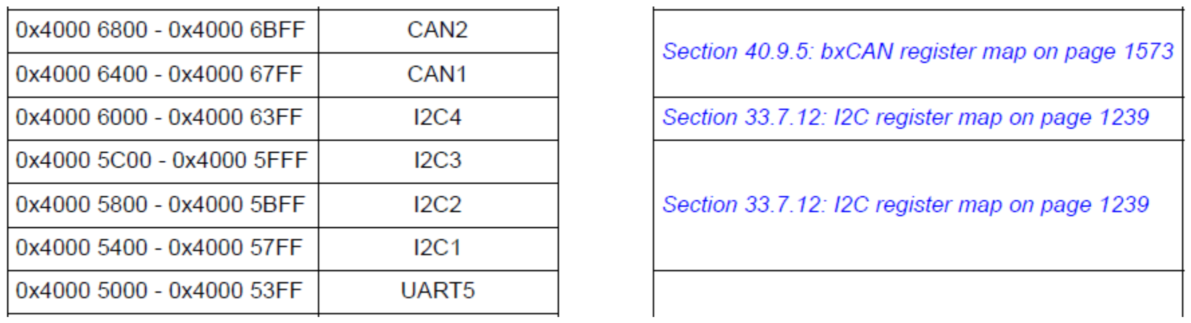

Reserved
0xE010 0000 - 0xFFFF FFFF
Cortex-M7 internal peripherals
0xE000 0000 - 0xE00F FFFF
AHB3
0x6000 0000 - 0xDFFF FFFF
Reserved
0x5006 0C00 - 0x5FFF FFFF
0x5006 0BFF
AHB2
0x5000 0000
Reserved
0x4008 0000 - 0x4FFF FFFF
0x4007 FFFF
AHB1
0x4002 0000
Reserved
0x4001 6C00 - 0x4001 FFFF
0x4001 6BFF
APB2
0x4001 0000
Reserved
0x4000 8000 - 0x4000 FFFF
0x4000 7FFF
APB1
0x4000 0000
0xFFFF FFFF
512-Mbyte Block 7 Cortex-M7 Internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte Block 6 FMC
0xD000 0000
0xCFFF FFFF
512-Mbyte Block 5 FMC
0xC000 0000
0x9FFF FFFF
512-Mbyte Block 4 Quad-SPI and FMC bank 3
0x8000 0000
0x7FFF FFFF
512-Mbyte Block 3 FMC bank 1 to bank 2
0x6000 0000
0x5FFF FFFF
512-Mbyte Block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte Block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte Block 0
0x0000 0000
Reserved
0x2008 0000 - 0x3FFF FFFF
SRAM2 (16 KB)
0x2007 C000 - 0x2007 FFFF
SRAM1 (368 KB)
0x2002 0000 - 0x2007 BFFF
DTCM (128 KB)
0x2000 0000 - 0x2001 FFFF
Reserved
0x1FFF 0020 - 0x1FFF FFFF
Option Bytes
0x1FFF 0000 - 0x1FFF 001F
Reserved
0x0820 0000 - 0x1FFE FFFF
Flash memory on AXIM interface
0x0800 0000 - 0x081F FFFF
Reserved
0x0030 0000 - 0x07FF FFFF
Flash memory on ITCM interface
0x0020 0000 - 0x003F FFFF
Reserved
0x0011 0000 - 0x001F FFFF
System memory
0x0010 0000 - 0x0010 EDBF
Reserved
0x0000 4000 - 0x000F FFFF
ITCM RAM
0x0000 0000 - 0x0000 3FFF
MSv39118V3

Figure 2-1: Memory map taken from the STM32F4 ARM processor manual [6]

This may seem mundane to the uninitiated but on an embedded processor every functionality is controlled through memory or more specifically the Direct Memory Access (DMA) controller [7]. Many communication protocols and other functionalities are supported by designing hardware in the silicon to work with the CPU core. Take for example the Controller Area Network (CAN) interface, or sometimes called “CAN bus”. On a PC platform the protocol is not supported natively on the processor itself, must make use of additional hardware, and often programs will import libraries such as a Dynamic Link Library (DLL) to interact with said hardware. On an embedded system this design paradigm is completely different. Settings for the CAN bus, data out, data in, and many other aspects of this protocol are accessed by writing or reading to/from memory locations. Figure 2-2 was taken from a microprocessor datasheet and shows some communication protocols, including CAN, and their address ranges for that protocol or interface. Navigating to the associated section in the datasheet for each interface would show in greater detail as well as describe functionality for individual bytes and bits of the space in memory. This information can be useful during the design process but also when it comes to analyzing firmware. For example, if there was known functionality about an embedded system, such as a vehicle ECU, memory locations could be pulled from the datasheet and searched for in the firmware file to identify functions that write, read, or initialize the CAN bus.



0x4000 6800 - 0x4000 6BFF	CAN2	Section 40.9.5: bxCAN register map on page 1573
0x4000 6400 - 0x4000 67FF	CAN1	
0x4000 6000 - 0x4000 63FF	I2C4	Section 33.7.12: I2C register map on page 1239
0x4000 5C00 - 0x4000 5FFF	I2C3	Section 33.7.12: I2C register map on page 1239
0x4000 5800 - 0x4000 5BFF	I2C2	
0x4000 5400 - 0x4000 57FF	I2C1	
0x4000 5000 - 0x4000 53FF	UART5	

Figure 2-2: Memory locations associated with various hardware peripherals taken from the datasheet [6]

Much more could be described concerning microprocessor architecture and design paradigms, and in fact many books and university courses do just that [8], [9]. For this thesis however, the topic is far too expansive to try and do any more than scratch the surface of the topic. It is also worth noting that the lines are beginning to blur as many embedded processor manufactures have increased support for their products by releasing advanced toolsets for their products. Toolsets that auto generate clock settings, HALs, peripheral configurations, etc. for the engineer allowing him/her to remain unaware of the chip's inner workings [10].

2.2.2. Storage

Software environments typically can be oblivious to what sort of storage is being used. Software executables rely heavily on the operating system to take care of reading and writing data and the protocols associated with that. Engineers and researchers working with embedded systems must often be more aware of how this happens. Viable targets for storage can take more than one form and are not nearly as standardized.

2.2.2.1. Hard Disk

Hard disks often come in one of two forms, spinning disks and Solid State Drives (SSD). While older spinning disks are still commonly found in computer systems due to the lower cost in terms of storage density, SSD drives are becoming more common. Spinning disks get their name from stacks of spinning platters housed internally that are used for storing data. SSDs employ flash storage and are significantly faster. These storage technologies are almost always present in PC environments but rare in embedded devices. The interface to these storage devices is standardized between manufacturers of the devices and interchangeable between brands. While hard drives can often be the target of forensic investigations [11] [12], they are not typically relevant when analyzing a binary or executable.

2.2.2.2. Flash memory

Even though flash memory is utilized in SSDs it is being introduced as a separate sub chapter. Flash memory by itself is nonvolatile memory usually sold and utilized as an Integrated Circuit (IC) but also incorporated internally as on-chip flash in microprocessors. When seen in a PC environment it usually takes the form of an SSD but by itself is not uncommon in embedded systems. Even though many embedded processors have internal flash it is not uncommon to see a separate flash chip utilized in the design of an embedded system. In terms of performing a forensics analysis or reverse engineering operation reading a flash chip directly may be out of the question for a PC environment but for an embedded device it is commonplace as will be discussed in greater detail in later chapters.

2.2.2.3. EEPROM

EEPROM (electrically erasable programmable read-only memory) is nonvolatile memory like flash but has smaller page size than flash. EEPROM can be a common place to find user or device data. Because of its small page size, sometimes as small as a byte, it is often used for storing times, dates, configurations, password hashes, names, serial numbers, etc. While it may be possible to find a stand-alone EEPROM chip in a PC environment it is heavily abstracted.

2.3. Lifecycle/Software Differences

This chapter attempts to cover some basic differences in how code is developed and run between embedded and PC systems. This topic can be quite extensive and complex and will rely on reader's implied prior knowledge on some subjects.

2.3.1. Executable File Format (PE, ELF)

Researchers working in a PC environment will likely be familiar with the PE file format if working in a Windows environment and the ELF format if working in a UNIX environment. These files have everything needed for the operating system to load the program and run it.

Users may find these files rare when working with embedded systems from an analysis or reverse engineering standpoint. The ELF format is common in firmware development as it is commonly output from the compiler/linker but these files are usually not distributed. The files captured from a device or firmware update will usually be a binary with all the extra information an executable file has stripped out. This is because as part of an embedded system's build process the firmware gets put through a relocation process. During the relocation process physical memory addresses are assigned to offsets given by the ELF file to align with valid memory addresses in the processor to form a single executable binary ready to be run on the target processor.

This section leaves a lot of information about the build chain out intentionally for the sake of brevity. The key takeaway readers should have is the difficulty of analysis from a file that is not in an executable format. There is a high probability that the file encountered by users for analysis will be a simple binary. This file will not have associated addresses associated with it and figuring out a start address will be critical. Take for example a jump to address 0x80001B00; if the start address was 0x00 this would represent a jump to nearly ~2GB into memory. This is usually unrealistic for embedded devices. However, if the start address is defined as 0x80000000 then the jump becomes 0x1B00 (~7Kb) into memory which is more realistic.

Figuring out the base address will be essential for rebasing the binary in analysis tools. If reading directly from the processor using debugging tools these addresses will be explicitly discovered by nature of the tool. Without knowledge of where the binary resides in the address space the user and disassembly tools won't be able to make sense of the functions and control flow.

2.3.2. Bootloader

Bootloaders are a mainstay in embedded systems, without them it would not be possible to push firmware updates to the device. The most common analogy to the PC environment is the BIOS/UEFI. The purpose of a bootloader is to enable an embedded system to update its firmware. Bootloaders are typically very simple pieces of code that either jumps to application code or replaces existing application binary with new a new binary. It is not entirely uncommon for a bootloader to unencrypt a binary during the update process.

Bootloaders are needed due to the nature of how embedded processors work. Upon powering up or triggering a reset an embedded processor's Program Counter (PC), commonly called Instruction Pointer (IP), will start at a predefined memory location. On a design without a bootloader (like a prototype) this will start executing application code. In order to self-program and update the firmware this flow will need to be interrupted to execute a bootloader first. The bootloader can either be placed at the predefined start location or the first instruction of application code can jump to the bootloader memory location. Some embedded processors support other bootloader mechanisms such as enabling bootloader fuses. With the bootloader running the application can now be swapped. If no update is required, the bootloader will simply return process flow back to the start of application code.

Bootloaders can be implemented in different ways. Sometimes guidelines will be published from the chip manufacture but ultimately it is up to the designer to implement a bootloader as he/she determines is necessary. Accounting for the bootloader is important to any user analyzing or reverse engineering an embedded system.

2.3.3. Operating Systems

Operating systems on embedded devices are not at all similar to what is running on a PC. Usually the operating systems on an embedded device is called a Real Time Operating System (RTOS) due to the necessity of processing data coming in from peripherals and executing tasks in real time and on a pre-defined schedule. These RTOS's are lightweight, minimalistic, and are part of the same code base as the project itself. Usually firmware engineers import the code for the entire RTOS into their project and then write application code in files alongside the RTOS. Researchers should keep this in mind when investigating a firmware image as the OS will likely reside alongside application code in a binary.

2.3.4. Libraries

The use of libraries in embedded systems is common although likely not to the extent of PC based software. The big difference between embedded systems and PC based systems and how libraries are used is that in an embedded binary the library has been statically compiled and included alongside the application code. Embedded systems rarely have the ability to dynamically load a library and execute code from it so all functionalities borrowed from a library must be included and fit within the systems useable memory. In a PC system, libraries can be compiled statically but for the sake of keeping executables small libraries are kept dynamic and the executable relies on the operating system to import libraries.

2.4. Communication Similarities/Differences

When comparing typical communication protocols found in PC environments vs embedded environments there are some significant differences. It is assumed the reader is familiar with communication protocols used in PC systems such as Ethernet, serial, USB, etc. This thesis does not cover low level communication mechanisms such as SATA, pipelines, ram-processor interface etc. as those are often out of scope and far from the prerogative of PC engineers/ researchers. Below are some of the most common communication protocols found in embedded systems.

2.4.1. Ethernet

Ethernet has been prevalent in PC systems for some time and is becoming more common in embedded systems. The low level workings of Ethernet will not be discussed in this section. Although the Ethernet used in PC systems is identical to the usage in embedded systems monitoring the traffic over Ethernet is often different. PCs are conducive to host-based monitoring as the resources required for simultaneous monitoring are present and software has been written to capture and analyze the traffic. In an embedded system this capture, and analysis must usually be done off-host on a different machine. A common way to accomplish this is to configure a network in a manner that allows capture from a mirror port and analyze on a PC; although a network Test Access Point (TAP) could also be used.

2.4.2. Serial

Some readers may be familiar with the DB9 serial port commonly found on PCs in the 90's and early 2000's. This form of communication is still very prevalent in embedded systems but with some major caveats. Before elaborating some terms need to be defined:

- Universal Asynchronous Receiver Transmitter (UART): This is a communication interface. It consists of a transmit (TX) and a receive (RX).
- RS-232: This is a communication standard that defines voltages, physical connections, timing, etc. The DB9 connector PC users may be familiar with are a part of RS-232, which is an implementation of UART.
- Serial: This term is part of common speak that encompasses both above and not really a part of a defined standard. It is used colloquially and can mean many things.

The biggest difference between the UART on embedded systems and the UART behind RS-232 is voltage levels. Connecting an embedded system directly to a DB9 would likely damage it as the voltage of RS-232 is +/-12V whereas common voltages for embedded devices will be 3.3V or 5V.

Some embedded processors will have more than one UART interface and the pinout diagram of the chip will usually list the pins associated with the UART interfaces in pairs as (TX0, RX0), (TX1, RX1), etc. When investigating these interfaces hardware will need to be used such as offerings from FTDI described in section 4.6.4. When connecting hardware to the processor the TX is connected to RX and vis-versa. Because UART does not implement a separate clock signal the baud rate will need to be known or guessed. Since only a small number of baud rates are standard and only a few of those are commonplace it is usually not that difficult. Choosing the incorrect baud rate will usually still result in data which further limits the need to try every common baud rate before knowing whether data is present on the communication interface.

2.4.3. I^2C

I^2C (pronounced "I-squared-C") is a communication bus that is address based, uses a master device with multiple slave devices, and is a common communication mechanism for sub-components of a circuit board. I^2C is sometimes referred to as TWI (two wired interface) due to trademark concerns [13] but the two are synonymous. Each slave device on the bus must have its own unique address, otherwise multiple slaves may try to respond to data transmissions. Because I^{2}C is a relatively slow protocol it is usually reserved for devices that do not inherently demand high speed rates of data transfer. Low speed devices such as humidity sensors, light sensors, analog to digital converters, etc. are ideal candidates for I^{2}C. I^{2}C relies on two signal wires (SDA-data, and SCL-clock) that multiple devices can connect to. Both signal wires have a pullup resistor to positive voltage as seen in Figure 3. The low level bitstreams that are part of the I^{2}C protocol are not being presented due to the existence of tools to adequately abstract them from the user.

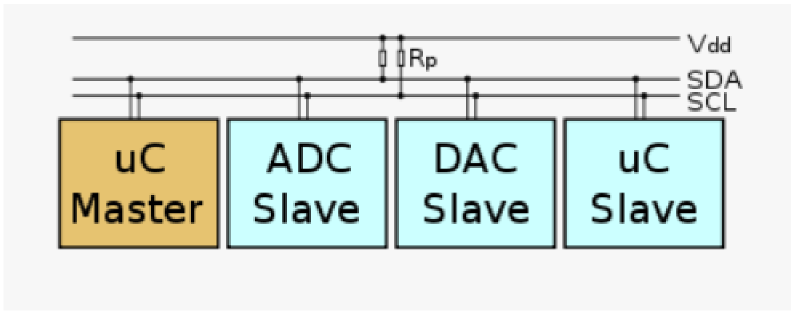


Figure 2-3: Wire diagram for I^{2}C [14]

2.4.4. Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) is often found in embedded systems, and like I^{2}C, is used as a component to component communication protocol such as microprocessor to flash, display, or other component. The protocol relies on a minimum of three signals; Master Out Slave In (MOSI), Master In Slave Out (MISO), and a clock signal (SCK). More signals are required if multiple devices are connected to the same SPI bus which are usually designated Slave Select (SS) which simply enables or disables a device. If only one slave is connected to the SPI bus it is common practice to hardwire the SS signal to its enabled condition.

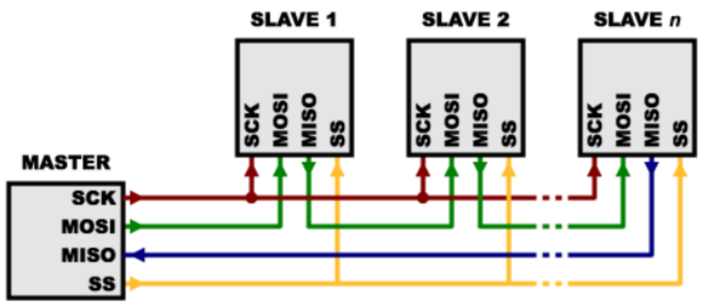


Figure 2-4: Wire diagram of SPI communication protocol [15]

2.4.5. Communication Similarities/Dissimilarities Summary

These protocols have been listed because it may be necessary to listen in on or otherwise investigate the protocols. Investigation methods will be briefly discussed in later chapters. A part of analysis on embedded devices usually involves going to the circuit board level and taking into consideration the entire design of a device and what is communicating to what. Using the information provided readers may be more aware of what they are investigating at a component level.

CHAPTER 3: TOOLS AND PRACTICES

3.1. Tool introduction

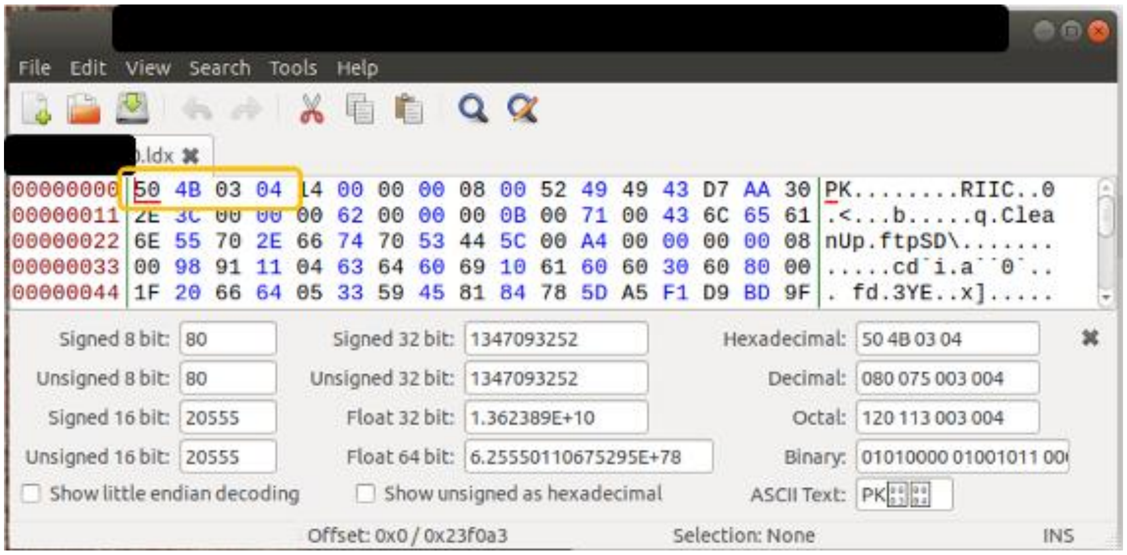
This section covers some tools used to inspect and analyze firmware. The scope of this topic is quite large. There are many tools that are as universally applicable to firmware as they are to software.

3.2. Hex Editors

Hex editors are used to view and edit the contents of a file in the raw byte format. They are not specific to working with embedded systems and are just as applicable in a PC environment. Often bytes are displayed in hexadecimal format by default, but decimal, binary, and octal representations are supported on most hex editors. Hex editors are somewhat interchangeable with minor variations in capability or layout. The number hex editors available is far too great to include all of them in the report but the two most used by the author are presented below. Hex editors can be especially useful when comparing two files, looking for artifacts in files such as strings, or editing data at a very low level. Another technique is to use a hex editor to view the first bytes in a file. These first bytes are often referred to as a file signature and identify the type of file [16]. The file extension cannot be relied on to identify the type of file and the author has found numerous times where files with odd manufacture specific extensions end up being a common compressed file type such as zip.

3.2.1. Bless Hex Editor

Bless is an open source hex editor for Linux that comes with a Graphical User Interface (GUI). It is free to use as it is licensed under the terms of the GNU General Public License (GPL) [17]. While it is not the author's favorite hex editor it is the one most often used by the author in a Linux environment due to the ease of installation and zero cost. The file signature 0x50 4B 03 04 is boxed in orange in Figure 3-1 which indicates a zip file even though the file extension was not zip. This information was useful during the analysis of a PLC's firmware.


[redacted]

[redacted].ldx

```
00000000 50 4B 03 04 14 00 00 00 08 00 52 49 49 43 D7 AA 30 PK........RIIC..0
00000011 2E 3C 00 00 00 62 00 00 00 0B 00 71 00 43 6C 65 61 .<...b.....q.Clea
00000022 6E 55 70 2E 66 74 70 53 44 5C 00 A4 00 00 00 00 08 nUp.ftpSD\.......
00000033 00 98 91 11 04 63 64 60 69 10 61 60 60 30 60 80 00 .....cd`i.a``0`..
00000044 1F 20 66 64 05 33 59 45 81 84 78 5D A5 F1 D9 BD 9F . fd.3YE..x].....
```

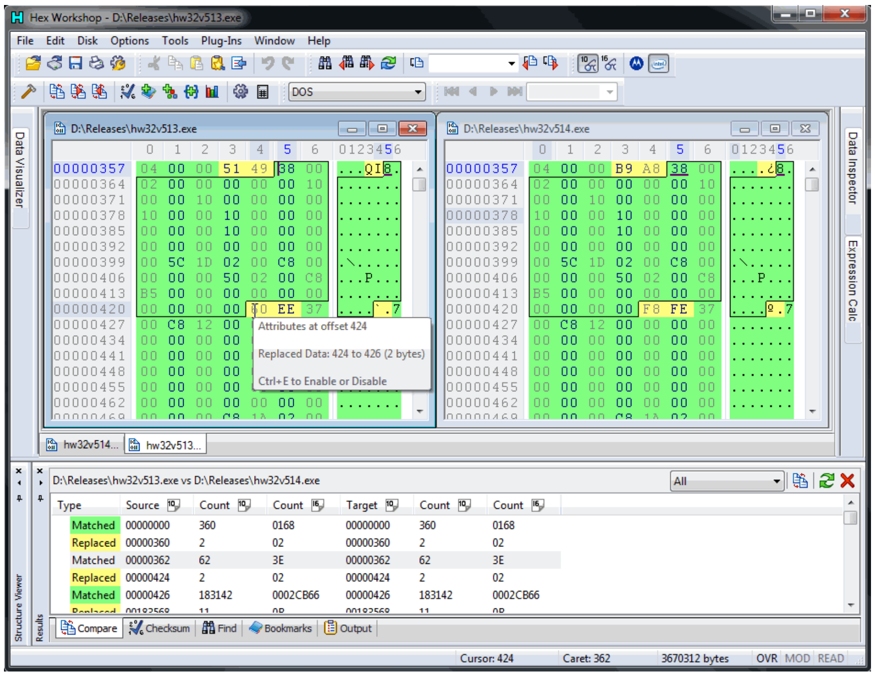
Signed 8 bit: 80
Unsigned 8 bit: 80
Signed 16 bit: 20555
Unsigned 16 bit: 20555
Signed 32 bit: 1347093252
Unsigned 32 bit: 1347093252
Float 32 bit: 1.362389E+10
Float 64 bit: 6.25550110675295E+78
Hexadecimal: 50 4B 03 04
Decimal: 080 075 003 004
Octal: 120 113 003 004
Binary: 01010000 01001011 00
ASCII Text: PK
☐ Show little endian decoding
☐ Show unsigned as hexadecimal

Offset: 0x0 / 0x23f0a3 Selection: None INS

Figure 3-1: Bless Hex Editor with a zip file loaded. The file header is outlined in orange

3.2.2. Hex Workshop

Hex Workshop from BreakPoint Software [18] is the author's preferred hex editor and what he cut his teeth on. This particular hex editor also has one of the best implementations of comparing files than any other hex editor tried. Comparing files at the binary level may not be useful for most analysis applications but it was especially useful when comparing tuning files from vehicle ECMs. Figure 3-2 below shows a screenshot from Hex Workshop comparing two files.



```
D:\Releases\hw32v513.exe
          0  1  2  3  4  5  6   0123456
00000357  04 00 00 51 49 38 00  ...QI8.
00000364  02 00 00 00 00 00 10  .......
00000371  00 00 10 00 00 00 00  .......
00000378  10 00 00 10 00 00 00  .......
00000385  00 00 00 10 00 00 00  .......
00000392  00 00 00 00 00 00 00  .......
00000399  00 5C 1D 02 00 C8 00  .\.....
00000406  00 00 00 50 02 00 C8  ...P...
00000413  B5 00 00 00 00 00 00  .......
00000420  00 00 00 00 60 EE 37  ....`.7
00000427  00 C8 12 00
00000434  00 00 00 00
00000441  00 00 00 00
00000448  00 00 00 00
00000455  00 00 00 00
00000462  00 00 00 00 00 00 00  .......
```

Attributes at offset 424

Replaced Data: 424 to 426 (2 bytes)

Ctrl+E to Enable or Disable

```
D:\Releases\hw32v514.exe
          0  1  2  3  4  5  6   0123456
00000357  04 00 00 B9 A8 38 00  ....¿8.
00000364  02 00 00 00 00 00 10  .......
00000371  00 00 10 00 00 00 00  .......
00000378  10 00 00 10 00 00 00  .......
00000385  00 00 00 10 00 00 00  .......
00000392  00 00 00 00 00 00 00  .......
00000399  00 5C 1D 02 00 C8 00  .\.....
00000406  00 00 00 50 02 00 C8  ...P...
00000413  B5 00 00 00 00 00 00  .......
00000420  00 00 00 00 F8 FE 37  ....º.7
00000427  00 C8 12 00 00 00 00  .......
00000434  00 00 00 00 00 00 00  .......
00000441  00 00 00 00 00 00 00  .......
00000448  00 00 00 00 00 00 00  .......
00000455  00 00 00 00 00 00 00  .......
00000462  00 00 00 00 00 00 00  .......
```

D:\Releases\hw32v513.exe vs D:\Releases\hw32v514.exe

Type	Source	Count	Count	Target	Count	Count
Matched	00000000	360	0168	00000000	360	0168
Replaced	00000360	2	02	00000360	2	02
Matched	00000362	62	3E	00000362	62	3E
Replaced	00000424	2	02	00000424	2	02
Matched	00000426	183142	0002CB66	00000426	183142	0002CB66

Cursor: 424 Caret: 362 3670312 bytes OVR MOD READ

Figure 3-2: Screenshot from Hex Workshop comparing two different files. Similarities are highlighted in green with dissimilarities highlighted in yellow.

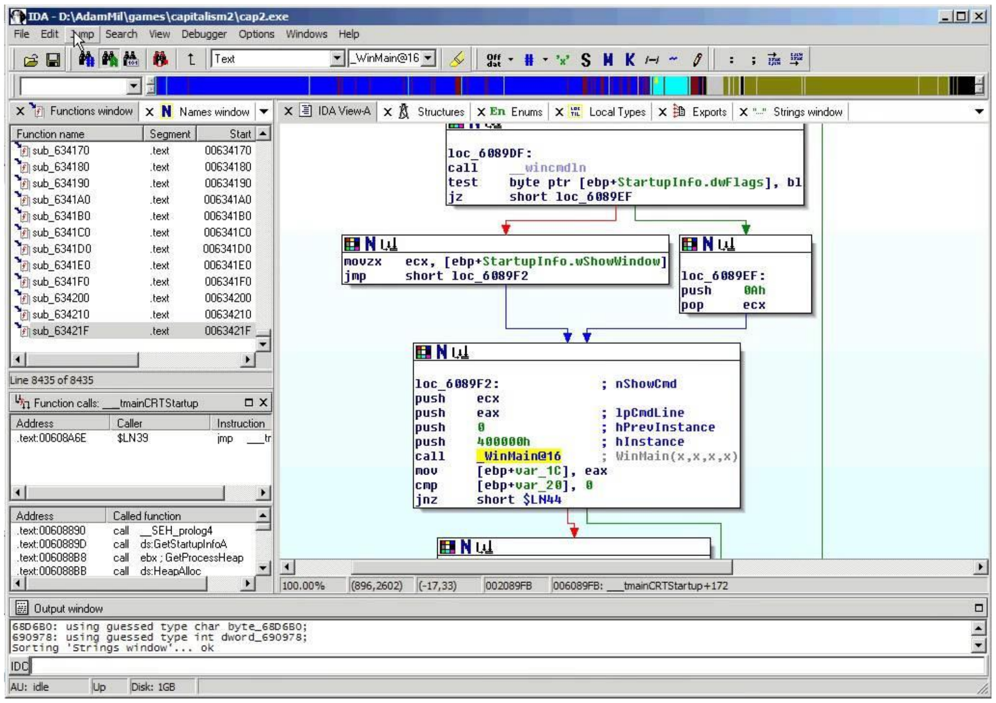
3.3. Disassemblers

Disassemblers are a family of software that takes binary files and translates them into assembly. These programs are as useful when reverse engineering firmware as they are software. Many disassemblers also work to identify functions and control flow as well as breaking assembly up into functional blocks. The important aspect of a disassembler as it applies to embedded systems is the software must support the target architecture (processor) in order to be useful. There are more disassemblers out there than the four listed, but these are the 4 the author is most familiar with and in the author's opinion the most well-known.

One of the biggest differences when reverse engineering firmware, aside from processor architecture, is that having access to an Executable Linkable Format (ELF) file is often rare. The file type and format of the firmware retrieved will vary but it will almost exclusively always be static, stripped, and not have address information affiliated with the firmware. What this means to the researcher will be discussed in greater detail in a later chapter but in short having a static file means all code and functionality are included in one binary as opposed to importing some dynamic library. Having a stripped binary means variable names, function names, and other useful bits of information will be absent. Part of having access to an executable, such as the case when working in a PC environment, is that addresses are included, which libraries are imported are included, and the file isn't always stripped so sometimes function names can be retrieved. Having a disassembler that can handle these restrictions is important if it is to be used with a firmware file.

3.3.1. IDA Pro

IDA Pro (or sometimes just called IDA) from Hex-Rays [19] , seen in Figure 3-3, has a long history of being the *De facto* standard when it comes to disassemblers. The software has been able to outshine its competition and it demands a hefty price tag for the privilege. Hex-rays takes a tiered or Downloadable Content (DLC) approach to its licensing [20] with each additional functionality or architecture support costing extra. With each license module cost in the thousands it's easy to achieve a $10k+ price tag for a fully supported software package which is well out of reach for the casual reverse engineer. Hex Rays offers a freeware version of IDA with reduced functionality and reduced processor architecture support.



Function name	Segment	Start
sub_634170	.text	00634170
sub_634180	.text	00634180
sub_634190	.text	00634190
sub_6341A0	.text	006341A0
sub_6341B0	.text	006341B0
sub_6341C0	.text	006341C0
sub_6341D0	.text	006341D0
sub_6341E0	.text	006341E0
sub_6341F0	.text	006341F0
sub_634200	.text	00634200
sub_634210	.text	00634210
sub_63421F	.text	0063421F

Address	Caller	Instruction
.text:00608A6E	$LN39	jmp ___tr

Address	Called function
.text:00608890	call __SEH_prolog4
.text:0060889D	call ds:GetStartupInfoA
.text:006088B8	call ebx ; GetProcessHeap
.text:006088BB	call ds:HeapAlloc

```
loc_6089DF:
call    __wincmdln
test    byte ptr [ebp+StartupInfo.dwFlags], bl
jz      short loc_6089EF
```

```
movzx   ecx, [ebp+StartupInfo.wShowWindow]
jmp     short loc_6089F2
```

```
loc_6089EF:
push    0Ah
pop     ecx
```

```
loc_6089F2:                    ; nShowCmd
push    ecx
push    eax                    ; lpCmdLine
push    0                      ; hPrevInstance
push    400000h                ; hInstance
call    _WinMain@16            ; WinMain(x,x,x,x)
mov     [ebp+var_1C], eax
cmp     [ebp+var_20], 0
jnz     short $LN44
```

```
100.00%  (896,2602)  (-17,33)  002089FB  006089FB: ___tmainCRTStartup+172
```

```
68D6B0: using guessed type char byte_68D6B0;
690978: using guessed type int dword_690978;
Sorting 'Strings window'... ok
```

Figure 3-3: Screenshot of IDA in use

3.3.2. Binary Ninja

Binary Ninja is another commercial offering for a disassembler. While not as full featured as IDA Pro it is significantly cheaper. At the time of writing the Personal/Student license cost

$149 USD and the Commercial License costs $599 [21]. There is also a demo version of Binary Ninja although compared to the freeware version of IDA it seems a bit more restrictive by prohibiting users to load or save databases, only allowing 25 minute sessions, and not allowing access to the API [22]. Of the four disassemblers listed the author has the most experience with Binary Ninja in a professional setting. Binary Ninja was chosen within the author's professional setting at the time due to its lower cost and excellent API which is relatively easy to use provided the user is familiar with Python. One of the frustrating aspects of Binary Ninja is it currently lacks the capability to rebase a file. While it is possible to do this with a script the lack of native capability is inconvenient.

3.3.3. Ghidra

Ghidra is an open source reverse engineering tool (disassembler) developed by the National Security Agency (NSA). Ghidra was initially released at a the RSA conference in March of 2019 and soon followed by the release of the source code in April of the same year [23]. Ghidra comes with a gambit of capabilities, and as a piece of software with a sense of completeness, not often seen with open source tools. A tool of this caliber being released as open source caused a stir among the reverse engineering community [24], [25], [26] with many forum users pointing out that IDA Pro may soon have stiff competition as users expand the functionality through software updates and plugins. This hype may be partially credited to the nature of the organization that developed it as the NSA has long been the cyber weapons shop for the US government. Given this tool was so new at the time of writing this thesis (< 1 year) the author has had limited experience with the software.

3.3.4. Radare2

Radare2 is an open source command line decompiler. There is a front end for Radare2 called Cutter which must be installed separately. Because it is open source it is often a great option for academic work although some users may find the command line interface difficult to use. The author has the least amount of time on this debugger but anecdotally from reading reviews and exchanges among professionals in the reverse engineering space it remains a viable option.

3.4. Debugging Hardware & Software

Readers may already be familiar with debuggers such as OllyDbg or the debugger built into their programming development environment. For the readers who are not familiar; a debugger enables a user to step through code or a binary and see the effects each line of code or instruction has on registers and memory. Software engineers may often do this with code to analyze what their code is doing but in the reverse engineering and binary analytics field this is rare due to only having access to the binary being typical.

Debugging with embedded devices is similar in concept to debugging with a PC. The major difference is usually extra hardware is involved. On a PC platform the processor architecture on the machine is often compatible with the binary. In an embedded system the processor architecture will be different as well as having different registers, memory locations, and peripherals. In order to debug the target system, it will need to interface with the PC through extra hardware. Below in Figure 3-4 the author has connected an ARM development board from STMicroelectronics (STM) [27] to a debugger from IAR systems called the J-Link [28]. It’s worth noting that the development board pictured comes with built in debugging hardware (left side of board) called ST-LINK/V2 [29]. The IAR J-Link was used to demonstrate debugging a target device that does not have built in debugging hardware as is the case with nearly 100% of production devices.

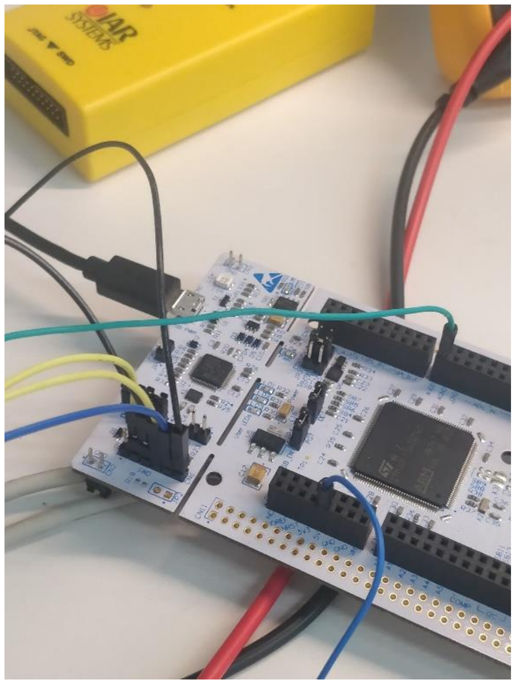


Figure 3-4: The author hooked up the Jlink to debug an ARM development board from STM

As opposed to following previous sections pattern and listing off some debuggers, some debugging hardware interfaces will be listed instead. The hardware that is available is usually distributed by the manufacture of the chipset in question; Texas Instruments, Microchip, Atmel, STM, etc. all have different debuggers that are not compatible with each other. There are third party vendors of debuggers, such as Laterbach [30], that make debuggers but even their products are only compatible with different architectures if the appropriate hardware module/add-on is used. The following interfaces are listed because these are what the debugger will need to physically connect to. Understanding them and how to find them will make it possible to start debugging an embedded system.

3.4.1. JTAG

JTAG (pronounced “J-tag”) stands for Joint Test Action Group which is the group that came up with the standard. JTAG is an IEEE standard for “Test Access Port and Boundary-Scan Architecture” [31] and has been adopted by most microprocessor companies. JTAG has long been the D*e facto* standard for debugging interfaces; while other companies have come up with their own debugging mechanisms industry parlance often uses JTAG as the go-to vernacular.

The pins/signals required for interfacing with JTAG on a chip are listed and described below. There are 4 required, 1 optional, and ground needs to be connected for voltage reference but is usually not explicitly mentioned in instructions or manuals.

- **TDI:** Test Data In; serial data from debugger to target
- **TDO:** Test Data Out; serial data from target to debugger
- **TCK:** Test Clock
- **TMS:** Test Mode Select; controls the Test Access Port (TAP) controller state transitions
- **TRST:** Test Reset; optional, resets the TAP controller

The important part for users is to not understand the inter-workings and low-level implementations of JTAG but how to connect and use it. While there are standards for connectors as seen in Figure 3-5 these are not always present on the board. In the author's experience if the connector is present on the board it will rarely be populated (footprint is there but physical connector is not soldered) such as the case in Figure 3-6. If a user desires to connect to a production device (not prototype status) there is a strong possibility a soldering iron will be involved. If a JTAG standard header is not present the first step should be consulting the datasheet for the target CPU. This datasheet should have a pinout describing what each pin on the chip does. Use this to identify which pins relate to JTAG; these will be the pins that need connections to, either directly or follow a PCB trace to a suitable solder/connection point.

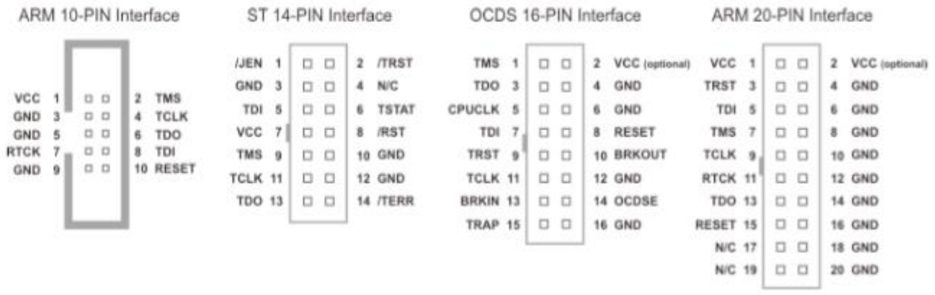

ARM 10-PIN Interface
VCC 1, GND 3, GND 5, RTCK 7, GND 9
2 TMS, 4 TCLK, 6 TDO, 8 TDI, 10 RESET
ST 14-PIN Interface
/JEN 1, GND 3, TDI 5, VCC 7, TMS 9, TCLK 11, TDO 13
2 /TRST, 4 N/C, 6 TSTAT, 8 /RST, 10 GND, 12 GND, 14 /TERR
OCDS 16-PIN Interface
TMS 1, TDO 3, CPUCLK 5, TDI 7, TRST 9, TCLK 11, BRKIN 13, TRAP 15
2 VCC (optional), 4 GND, 6 GND, 8 RESET, 10 BRKOUT, 12 GND, 14 OCDSE, 16 GND
ARM 20-PIN Interface
VCC 1, TRST 3, TDI 5, TMS 7, TCLK 9, RTCK 11, TDO 13, RESET 15, N/C 17, N/C 19
2 VCC (optional), 4 GND, 6 GND, 8 GND, 10 GND, 12 GND, 14 GND, 16 GND, 18 GND, 20 GND

Figure 3-5: Some standard JTAG connectors [32]

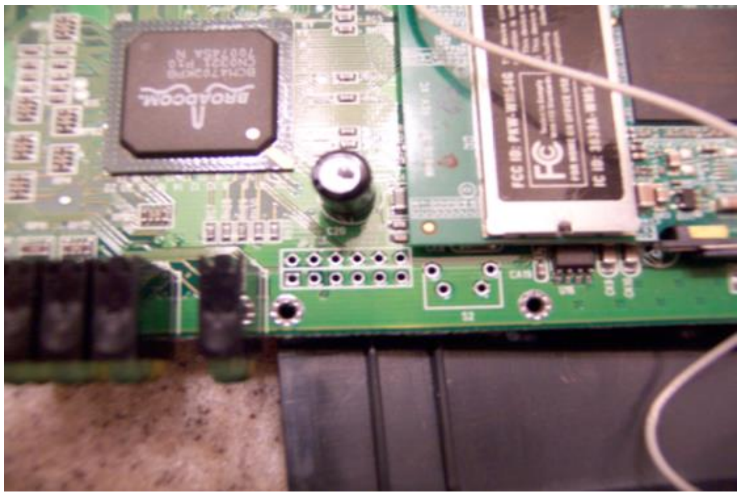


Figure 3-6: Unpopulated JTAG connector on a DD WRT router [32]

Unfortunately, successful connection to the JTAG interface does not guarantee the ability to read from memory and start debugging the code that is currently present on the chip. Many processors come with options to lock down the read capabilities of debugging through a mechanism called fuses. The term ‘fuse’ is a carry-on term to an era when literal fuses were used inside the chip and would be permanently set by running current through specific pins to disable functionality. Most current processors do not use this, and configurations can be set and re-set.

3.4.2. SWD

Serial Wire Debug (SWD) is an alternative to JTAG for ARM processors [33]. One of the drawbacks of JTAG is that it uses a lot of signals (relatively). As some chips push to reach a smaller form factor, spare pins become a premium. 8 pin microprocessors are not out of the question and dedicating 4 to JTAG leaves only 2 General Purpose I/O (GPIO) once ground and Vcc are accounted for. The SWD protocol cuts this in half and only uses two pins; they are listed below along with their descriptions [34].

- **SWDCLK**: Serial Wire Clock
- **SWDIO**: Serial Wire Debug Input/Output; bi-directional signal carrying data.

The guidelines for connecting to SWD are identical to those for connecting to JTAG. It is a high likelihood that if a production device is the target, wires will need to be soldered on the circuit board in order to interface with SWD. Consult the chip manual for identifying pins designated for SWD functionality.

3.4.3. SWIM

Single Wire Interface Module (SWIM) is a proprietary protocol to STMicroelectronics [35]. It is another progression in lowering the number of pins required for debugging, in this case only 1 is needed. Going through SWIM would be superfluous and is presented to highlight the broad spectrum of debugging interfaces a user might encounter.

3.4.4. BDM

Background Debug Mode (BDM) is a debug interface proprietary to Freescale (formerly Motorola). It is similar to SWIM in that it needs only 1 pin for data [36]. Going through BDM would be superfluous and is presented to highlight the broad spectrum of debugging interfaces a user might encounter.

3.4.5. Debugging Hardware & Software Summary

In summary every embedded processor will have some form of debugging interface. The best way to identify the interface is through the chip's manual. JTAG is a well published standard set forth by IEEE and can be found on nearly any processor. SWD is something ARM came up with and can be found on a variety of manufactures that license the ARM core. Various other debug interfaces exist that are exclusive to the chip designer/manufacturer. It is up to the discretion of the chip designer which debug interface or interfaces to implement. Production devices will rarely, if ever, have a debug port or connector populated on the circuit board. Many times, a debug connection will not be made available on the circuit board and pins must be traced in order to find viable connection points.

3.5. Decompressing/Extracting

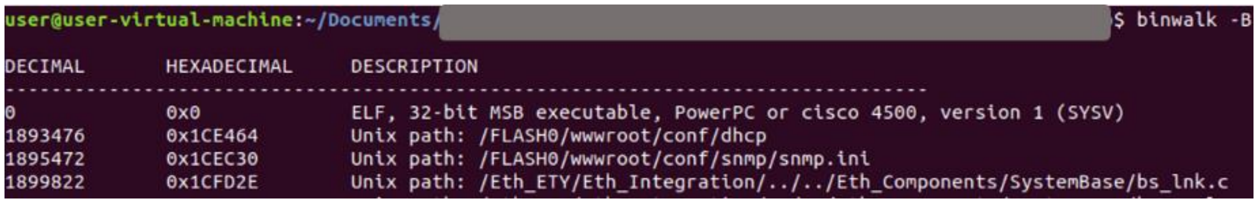
When analyzing embedded systems, it often becomes necessary to extract the contents of a file. This is similar to working in a PC environment with the exception of sometimes being more specialized or specific.

3.5.1. *Binwalk*

Binwalk [37] is a command line tool for performing various analysis related tasks on a binary file. The author has extended experience using the open source version but recently Binwalk Pro has been released and is a cloud-based version. The author highly recommends this tool as a first step in analyzing a piece of firmware, especially if the firmware was acquired from a vendor and physical access to the target device is not possible or greatly restricted. Binwalk can be used as a quick way to determine what is in a file, what sections of the file may be empty data, what sections of the file may be encrypted or compressed, recognizing processor architectures, as well as extracting known file types. Some of the flags the author uses the most are as follows;

`-B`	`Scan a target file for common file signatures`
`-A`	`Scan a target file for common opcode signatures`
`-E`	`Calculate a file’s entropy and generate an entropy graph`
`-e`	`Automatically extract known types`

In Figure 3-7 is a screenshot of Binwalk running with the –B flag and recognizing various characteristics and file signatures from a piece of firmware that runs on a PowerPC platform.



```
user@user-virtual-machine:~/Documents/[redacted]$ binwalk -B

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             ELF, 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV)
1893476       0x1CE464        Unix path: /FLASH0/wwwroot/conf/dhcp
1895472       0x1CEC30        Unix path: /FLASH0/wwwroot/conf/snmp/snmp.ini
1899822       0x1CFD2E        Unix path: /Eth_ETY/Eth_Integration/../../Eth_Components/SystemBase/bs_lnk.c
```

Figure 3-7: Screenshot from Binwalk using the -B flag

3.5.2. *CyberChef*

CyberChef is touted as being “*The Cyber Swiss Army Knife - a web app for encryption, encoding, compression, and data analysis*” [38]. The list of capabilities is long, but it provides an easy method of uploading a file and performing a list of operations on the file in sequence. Bitwise manipulation, data massaging, and encryption can all be performed with one run using an easy-to-use, drag-n-drop GUI.

3.5.3. *SRecord*

SRecord [39] is part of a large group of tools the author lumps together which are incredibly useful when needed but are rarely needed. The tool SRecord is a Unix command line tool that was created to manipulate Motorola S-Record files (often called SREC files). SREC files follow a standard format, they store data as plain text so are human readable, and part of each line of the file is a header, data, and checksum. In Figure 3-8 is a snippet of a SREC the author has come across. The manufacturer distributed this firmware file as part of a zipped package and once extracted it could be opened in a text editor. The data in red was not sent to the target device but rather was used by the PC based application and not technically part of the SREC file.

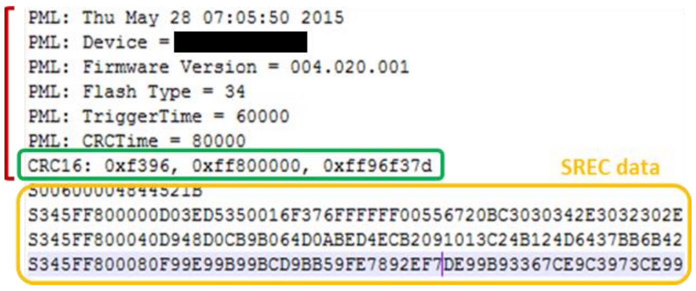

```
PML: Thu May 28 07:05:50 2015
PML: Device = [redacted]
PML: Firmware Version = 004.020.001
PML: Flash Type = 34
PML: TriggerTime = 60000
PML: CRCTime = 80000
CRC16: 0xf396, 0xff800000, 0xff96f37d
S00600004844521B
S345FF800000D03ED5350016F376FFFFFF00556720BC3030342E3032302E
S345FF800040D948D0CB9B064D0ABED4ECB2091013C24B124D6437BB6B42
S345FF800080F99E99B99BCD9BB59FE7892EF7DE99B93367CE9C3973CE99
```

Figure 3-8: A snippet from an SREC file the author has come across

SRecord was used to then extract and manipulate data contained in the file. Data was extracted from plain text and stored as a raw binary. This binary could then be loaded into a disassembler, rebased to the appropriate address, and analyzed. Alternatively, if modifications were added to the file, it could then be repackaged using SRecord with all checksums and metadata fixed.

3.5.4. U-Boot tools

U-boot tools is a subset of the project U-Boot [40] and distributed on their Github page as a part of U-boot. This toolset falls into the same broad category as SRecord in section 4.5.3 as a tool that is rarely needed but when it is, users will find it incredibly useful. U-Boot is a bootloader for embedded systems and U-boot-tools is a set suite of command line tools for editing extracting, packaging, and otherwise editing u-boot utilities and images. The U-Boot bootloader uses U-Boot images; think of them like executables or ROMs which can be loaded to an embedded device. A user may not have a need to examine the bootloader itself, however as the author has encountered real world scenarios from experience, the need to build a U-boot image. In one such scenario the author experienced; there was a need to alter a firmware image for a piece of equipment from the OEM without access to the source code, in other terms popularized by the news, “hack” the device. A tool already discussed in section 4.5.1 called Binwalk was able to extract the U-Boot image but once the alterations were made the only way the author was able to repackage the image was using U-Boot-tools.

3.6. Communication Snooping

When analyzing embedded systems, it may become necessary to snoop (eavesdrop) a communication protocol. Users already familiar with software like Wireshark will already be familiar with the concept. In fact, Wireshark itself is a very relevant tool when dealing with embedded devices due to the widespread adoption of network connected embedded systems. Many other snooping tools will be similar to Wireshark but will rely on some extra hardware and may not have as robust of a user interface.

3.6.1. Wireshark

Since Wireshark has been mentioned already it would be prudent to cover it. Without getting into too much detail about how network packets are routed to various devices, Wireshark allows a user to capture network packets, analyze them, and even save them for later use. The software is used heavily whenever development or analysis is taking place and ethernet connections are involved.

3.6.2. Saleae Logic Analyzer

Saleae is a company that makes and sells a line of logic analyzers popular among engineers and hobbyists. A logic analyzer is a tool used to measure/read digital signals. The functionality sometimes crosses with an oscilloscope, a tool to measure voltage with respect to time. The differentiating factor between the two is usually a logic analyzer will represent voltages as digital values (0 or 1) where an oscilloscope will have very high resolution with respect to voltage (microvolts).

The real value of a logic analyzer comes from its accompanying software. The software can often decode and display common protocols (I2C, SPI, Serial) as opposed to just signal values. This of course requires the user to appropriately identify signals on the target device and designate signal lines on the logic analyzer appropriately. Eavesdropping signals on a circuit board may sound out of scope for most scenarios but armed with the right plug-in, the

Saleae logic analyzer can passively listen and stitch together sections of a flash chip. Given a scenario in which an embedded processor has an accompanying flash chip it may be prove fruitful to use a logic analyzer to see what the processor reads/writes at boot vs normal operation.

3.6.3. *Bus Pirate*

The Bus Pirate is an open source device that interfaces with a computer and can 'speak' electronics. The device supports many protocols that electronic chips rely on to communicate over copper traces on a circuit board; a graphic representing the Bus Pirate can be seen in Figure 3-9. Some open source tools such as FlashRom support the Bus Pirate which makes it easy to read the contents of flash chips. The Bus Pirate in a way is the antonym of a logic analyzer; instead of passively listening to the device it can be used to actively interrogate the device.

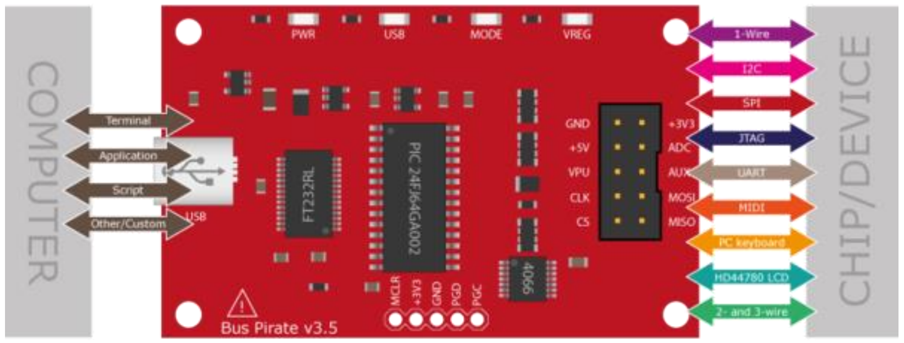


Figure 3-9: Rendering of the Bus Pirate device. It acts as an interface between a computer and electronic communication protocols.

3.6.4. *FTDI FT232*

The FTDI FT232 is a USB to serial adapter; technically USB to Universal Asynchronous Receiver/Transmitter (UART). FTDI (Future Technology Devices International) is the company behind the device and has been known for its line of FT232 products since the 90's. Really this shouldn't be thought of as a single device but a family of devices as there are several knock offs and other devices that do the same thing. A serial interface used to be common on PCs but this is no longer the case which makes this device, or one like it, necessary to communicate to devices using UART. It's not just exposed serial interfaces on the front or back panel that users should investigate with a USB-Serial adapter, but non-connected pins assigned to a serial peripheral on the processor as well. The author has discovered a telnet like menu operating on a serial interface not exposed externally of the device.

3.6.5. neoVI

The neoVI is a device made by Intrepid Control Systems that allows a computer to communicate or snoop vehicle communication protocols such as CAN or J1939. This tool would likely only be useful if a vehicle based embedded device was being investigated as the use of CAN on production devices outside of vehicles is rare. The author has extensive knowledge of tools from Intrepid Control Systems, like neoVI and valueCAN, during his tenure at a company which reverse engineered vehicle ECMs.

3.6.6. Communication Snooping Summary

The devices listed in this section were primarily listed to highlight the diversity of tools that may be needed to thoroughly investigate an embedded device. The selection of tools a user chooses to employ will ultimately reflect the goals and purpose of research or analysis.

CHAPTER 4: PROVINANCE DISCOVERY IN ANALYSIS

4.1. Motivation for Provenance attribution

Analysis on embedded systems is motivated by many factors but a major goal is provenance attribution; or in other words where the binary came from and what sub-components (such as 3rd party libraries) make up the binary. There are three main reasons to discover the origin and hereditary of a binary file: potentially vulnerable library identification, license enforcement and intellectual property protection, and attribution classification.

4.2. Vulnerable Library Identification

Discovering which libraries are used in a binary is incredibly useful for vulnerability detection. Libraries present a major pathway for vulnerabilities to persist and propagate. A common practice for software developers is to recycle code, grabbing third party libraries that provide desired functionality and incorporating it into their project. This design paradigm is done for efficiency sake as the cost of implementing every little capability in-house (TCP stack, HTTP server, drivers, etc..) into a device would be very costly. The result is that nearly every product is comprised of multiple libraries, each with their own patch cycle. Hand the product off to a junior engineer, as is common industry practice, and the problem is exacerbated as the maintenance engineer may not have clarity or insight as to which libraries were used and how. Such was the case with Busy Box version 1.21.1 and a vulnerability in its NTP implementation. Busy box used openNTPd, a separate library, for its NTP implementation. From the time openNTPd was patched until the time Busy Box incorporated the patch was 7 years [41]. This significant vulnerability remained in production code for a significant time after a patch was available due to a lack of situational awareness to incorporated libraries and their respective patch cycles.

The scenario of an engineer not knowing what code is in the device is far more common than people think. The author can speak from personal experience when he was working as an embedded engineer. Anecdotally, roughly 80 percent of the code base was reused with the remaining 20 percent changing depending on the application. The majority of the code used in the device was untouched and unseen by the author. As an engineer working for the company, there wasn’t a reason to know what libraries were used as only the code that needed to be fixed, or to support new applications, was seen. It wasn’t until efforts were made towards supporting a new application that it was discovered the code that was handling the reading/writing to the SD card was an unlicensed library. This isn't necessarily a vulnerability but highlights the widespread problem of unseen libraries being used.

4.3. License Enforcement and Intellectual Property Protection

Enforcing license agreements or protecting intellectual property rights requires proof that it is being used. Unless an insider is involved, access to code is usually impossible with only access to the binary available. Identifying code use with only access to a binary is desirable.

Another related example drawn from the authors previous work experience, albeit anecdotal, was a case where the company he was working for (Call it company 'A') sued another company (company 'B') claiming they had stolen intellectual property. Company A had also sued a previous employee claiming he was working for company B and handing over trade secrets. All the intellectual property company A had was in their source code for the product they sold. This company sued on the ground that company B had stolen the code and were then using it in their product. The claim ultimately fell apart in court because Company A had zero way of proving their code was being used. The nature of what company A did would make a binary easily retrievable by one of their employees and if nothing else, through a request of discovery. Since they had the binary the utility of a tool to perform some sort of analysis as to what functions were in it would have been huge.

4.4. Attribution Classification

At the tail end of incident response or discovering a new piece of malware that hasn't been seen before is the attempt to provide attribution to who designed it. This is mostly done through analysis of common patterns in the kill chain and methods utilized by attackers. Understanding what is in malware, how it works, and the goals of a piece of malware help attribute the origination of the malware.

CHAPTER 5: ACQUISITION OF FIRMWARE FOR ANALYSIS

5.1. Acquiring Firmware

Before any analysis can be performed, firmware must first be acquired or captured. Technology moves quickly and as such many techniques must also adapt. There is no one technique or tool to work in every situation which often demands creativity when attempting to get a copy of firmware.

5.1.1. Non-Invasive Means of Capturing Firmware

There are two over-arching categories of capturing firmware: non-invasive and invasive. This section and subsequent sections cover methods to capture firmware without opening the target device (non-invasive).

5.1.1.1. Open sources

The first step users should take when investigating a device is to simply search around online and find out as much as possible regarding a device. It may so happen to be that every version of firmware for a device is hosted on the manufacture’s website available for download. Sometimes the firmware can be freely available, sometimes it requires an account to be setup (most likely for marketing purposes), other times firmware may be restricted but with a little social engineering can be acquired support representative. The author has had decent luck with playing slightly dumb, maybe an intern or new hire, saying something along the lines of “I need this version of firmware because it’s what my boss wants to deploy the device with”. Every company has their own policy about distributing firmware, some keep it closely guarded in the proverbial fort Knox and others give it out openly. Once firmware has been obtained then analysis can start. Analysis is covered in the next chapter.

The other part of device reconnaissance not strictly affiliated with firmware is to learn what’s on the device in both hardware and software terms. Users may stumble across a blog where someone else has done most of the work for them, hopefully even describing techniques they used and what they learned. They may share not only pictures but a list of hardware and software libraries they found. An example of this can be seen from an image in Figure 5-1. This image is from an Amazon Dot teardown. The author has never owned an Amazon dot but within a few minutes of online searching found some information on the hardware which was relevant to the analysis taking place. Based on this picture, information on the hardware architecture can be discovered such as discovering they are using on board, external from the processor, flash. Based on past experiences it is suspected there may be some worthwhile information on this chip, potentially even personal information that could be extracted like Amazon user accounts, settings, and history. It can also be seen what processor they're using. Knowing what processor they're using can help analysts strategize and tailor their toolset for that particular chip and architecture.

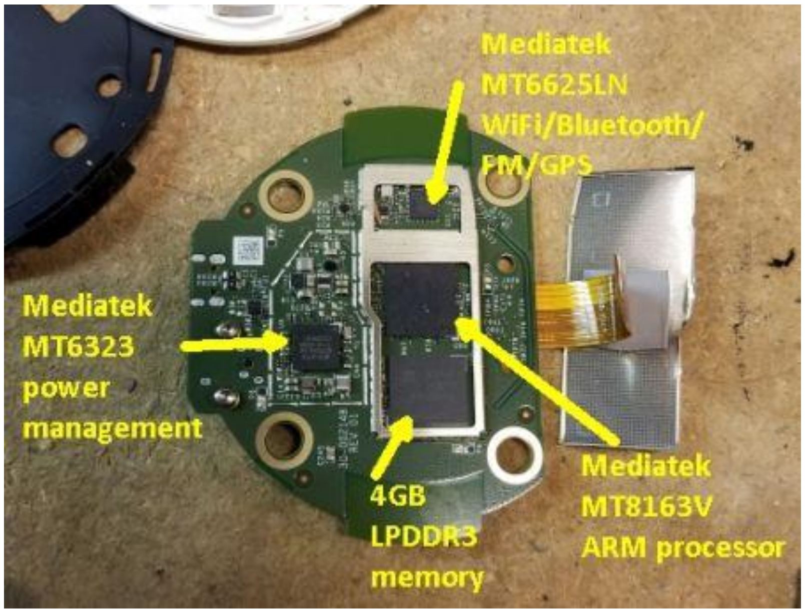

Mediatek MT6625LN WiFi/Bluetooth/ FM/GPS
Mediatek MT6323 power management
4GB LPDDR3 memory
Mediatek MT8163V ARM processor

Figure 5-1: Amazon echo dot teardown. Image taken from microcontrollertips.com [63]

If a copy of the firmware cannot be obtained by searching open sources online it is likely a physical device will be needed. The sections following this require a physical device.

5.1.1.2. Acquiring from OEM software

If a copy of firmware was not able to be acquired from online sources analysts will need to elevate efforts to a physical device. The first step is to understand how a firmware update happens in the first place and there are a lot of ways this can happen. If it's an Internet of Things (IoT) device, the device may pull firmware automatically from the cloud or it may require a physical media device like a USB Flash drive. If it updates from some computer application, users may have an opportunity to extract firmware from the application in some form of another.

An example of this is when the author was working on reverse engineering heavy duty trucks (Think 18-wheeler semi-trucks). A legitimate firmware update looked something like as follows. A truck rolls into the dealership for regular maintenance or warranty work. The dealership technician will almost certainly have a special diagnostic laptop running software unique to the make of vehicle and as part of the maintenance will plug the laptop, through an adapter, to the diagnostic port of the truck to check for diagnostic codes or to simply capture data from the truck for logging purposes. As part of the maintenance process the truck may receive a firmware update for its Engine Control Module (ECM). The modern car/truck is a highly computerized machine and the ECM can be considered the ‘brain’ of the vehicle. It runs code that controls the truck; everything from tuning to speed limiting and will often get patches released for it.

That process was investigated and monitored for a particular make of heavy-duty engine called Paccar. Paccar engines aren’t used in small vehicles like passenger cars and pickups but are used in long haul trucks, RVs, and marine applications. Paccar uses a software application called Davie to run diagnostics on its ECMs, a screen shot of this software can be seen in Figure 5-2. While modern versions of Davie have gone through a major facelift and operate completely differently, at the time the author was able to capture a copy of the firmware because over the course of a firmware update to the ECM, Davie would drop copies of it in a folder under the installation directory and later delete those files. All that had to be done to capture those firmware files was run a firmware update and copy the files before the firmware update finished.

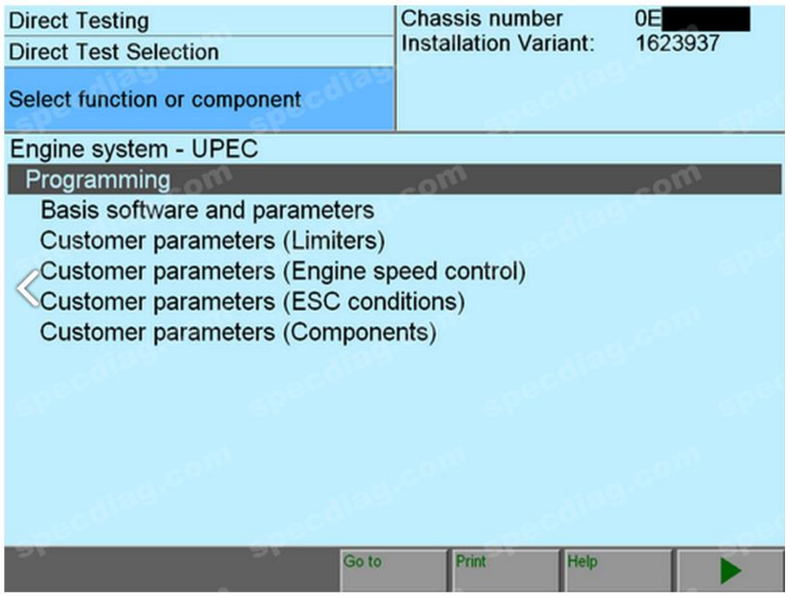

Direct Testing
Direct Test Selection
Select function or component
Chassis number 0E[redacted]
Installation Variant: 1623937
Engine system - UPEC
Programming
Basis software and parameters
Customer parameters (Limiters)
Customer parameters (Engine speed control)
Customer parameters (ESC conditions)
Customer parameters (Components)

Figure 5-2: Screen shot of Paccar Davie tool taken from http://specdiag.com [42]

Another potential avenue for extracting a firmware image from the updating software is to reverse engineer the software itself and/or possibly modify the software. This method isn't one of the authors strong points as an engineer, so is usually avoided by the author if possible.

5.1.1.3. Capture by snooping the communication protocols

If grabbing a copy of firmware directly from the computer application fails, then other methods should be attempted. Another method of grabbing firmware is over the wire. To attempt this, analysts will most likely need a way to initiate a firmware update which is often done using the software. Generally speaking the method will follow the process of monitoring whatever bus/communication the device is using to update the firmware (CAN bus, IP network, Serial, etc.), send the firmware update to the device, save the captured bus traffic, analyze and try to extract the firmware image from the bus traffic. The process is rarely exactly the same and the author has done this in the following scenarios:

Scenario 1: It may be found that a firmware update is simply being sent by FTP to the target device. An example of this is when the author was working with a PLC and looking to capture the firmware over the network. The setup was as follows: A network switch with a SPAN port was set up, the target device and the computer with the PLC software were set up running on the network, network traffic was captured from the SPAN port, the firmware update was sent. At this point a pcap file containing all of the communications the computer and target device had exchanged was captured. Since a firmware update was just sent over the network, a copy of the firmware in some form or another within that network traffic should have been captured. Upon analyzing the network traffic in Wireshark it was discovered the files were simply being transferred via File Transfer Protocol (FTP) and was unencrypted. The files were able to be extracted using Wireshark since the software supports this functionality. One of those files transferred ended up being the firmware and at this point was successfully extracted from the network traffic. This process was able to be scripted in python to extract it the firmware although at the time of writing has not been released. It's worth noting that even though users may see that the target device is being updated via FTP that doesn't mean there are going to be firmware files ready for reverse engineering. It's possible the manufacturer of the device is sending files that are encrypted via FTP.

Scenario 2: It may be observed that a firmware update is being sent over the network but using some unknown proprietary protocol. It may be possible to reverse engineer this and capture the firmware file. For example, the author was once working with a smart 3-phase power meter; similar to the power meter one may find on a residential building but with more features. A network was set up similar to the one in Scenario 1, a firmware update was sent, network traffic was captured, and then analysis was performed on the pcap file. It was discovered that a large amount of data was being sent to the meter which indicated to the author that there was a firmware file being sent but in a protocol that the author nor any tools recognized. It was possible, albeit with considerable effort, to write a script to extract the data and stitch it together to re-create the firmware file that was sent. It is worth noting doing this without a copy of firmware to start with would be exceedingly difficult. Because a copy of the firmware being sent was already on hand it was possible to identify 1:1 Byte matches in the network traffic to the firmware file.

Scenario 3: If the firmware update is being sent over an alternate means other than ethernet it may still be possible to capture it. An example of this builds off a scenario given earlier with the Paccar Davie software. It was mentioned that the software dropped the firmware files on the hard drive of the computer during a firmware update and then deleted them post update making them briefly available for copy. More recent versions of the Davie software don't work in the same way which forced another method to extract the files. During a firmware update the software sends the firmware files across the CAN bus and due to the nature of CAN, any device on the bus can listen in. Given previous understanding of how the ECM worked and how the software talked to the ECM the author was able to use CAN interface hardware and write a program to capture that file. It’s acknowledged though that this would be incredibly difficult without prior understanding of both CAN bus but also how the firmware was sent in the first place.

While it’s possible to capture network data from a firmware update over the network without initiating it this would be exceedingly difficult as you would likely end up with a massive amount of data to sift through and likely only have one shot at capturing it.

5.1.2. Invasive Means of Capturing Firmware

If non-invasive methods fail or are impractical then an escalation of the reverse engineering process is necessary. The next steps will involve invasive means of extracting firmware from the target.

5.1.2.1. External On-Board Flash

Some devices are designed with a separate flash chip on board such as the Amazon Echo previously seen in Figure 5-1. Devices can store various amounts of data and often times even firmware can be found on those chips which makes it a lucrative target to investigate. There are a couple overarching methods that data can be exfiltrated with. One method passively snoops the communications to/from the chip and the other involves actively writing/reading to the chip.

5.1.2.1.1. Passive Capture of On-Board Flash

This method involves trying to listen in on the communication between the flash chip and whatever device is attempting to read/write to the flash chip. Most flash chips communicate over a protocol called Serial Peripheral Interface (SPI). Some flash chips will communicate over I2C (sometimes called two-wire interface or TWI) but it’s rare due to speed constraints. Both protocols can be snooped but for this section only SPI will be covered. When passively capturing data from a flash chip an analyst will need some preliminary information. Information primarily needed is the pinout since users will be connecting probes to the communication lines. Unique implementations of SPI can be found with some flash chips that manufactures implement to increase the speed. At a minimum information on which pins correspond to MOSI (Master Out Slave In), MISO (Master In Slave Out) and the clock pin for SPI communications is needed. Once pins have been identified users will need to find a way to connect leads or probes to the pins either directly or by tracing out the circuit board and finding suitable landing pads. For this section the flash BIOS chip from a laptop as seen in Figure 5-3 is used as an example. A BIOS is not quite firmware but it's somewhat simple and makes for a good example.

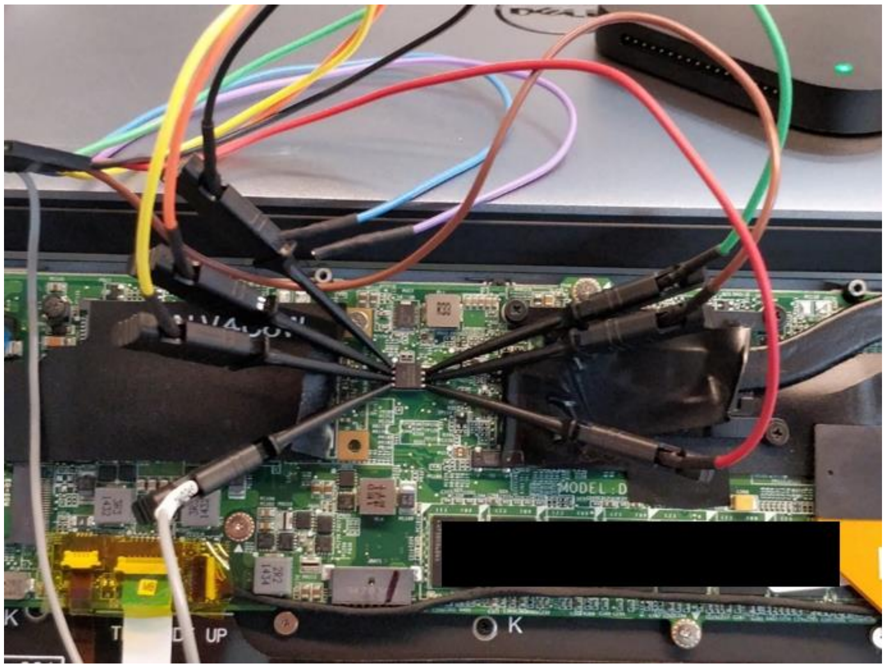


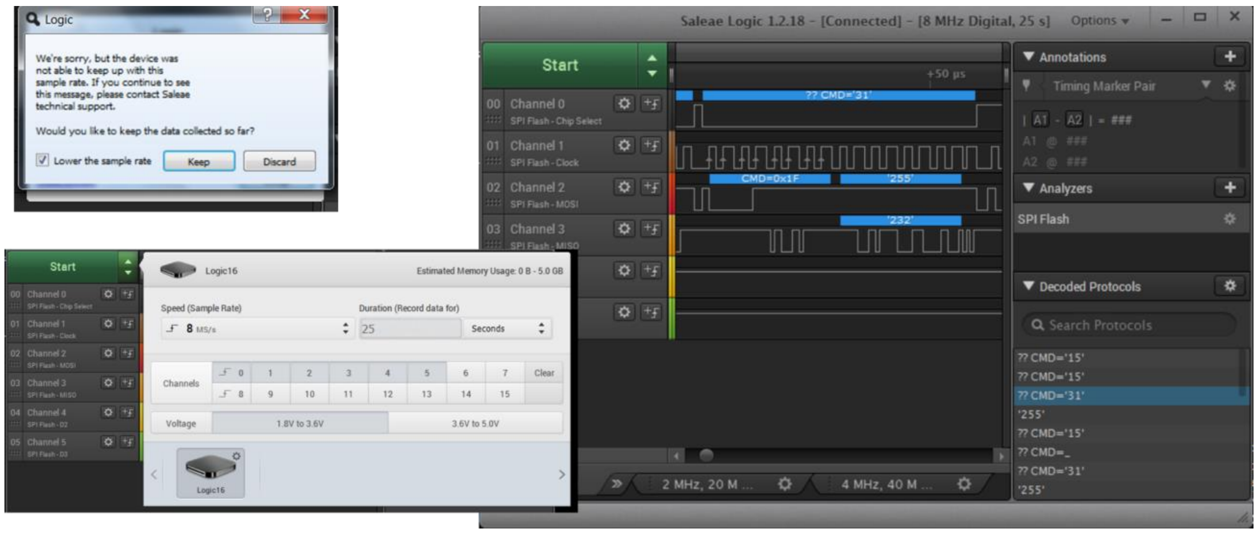
Figure 5-3: Connecting leads to the pins of a on board flash chip. The BIOS boots from this chip

To listen in on the data transfer happening between a flash chip and another chip, analysts will need some extra hardware like a logic analyzer. A logic analyzer is a tool similar to a multimeter or an oscilloscope but used for digital signals. Often, they are used to help engineers debug or figure out what is going on with their hardware. Think of them like Wireshark for digital circuits for users more familiar with that software. Often times accompanying software will be necessary to use a logic analyzer. If it is required to establish some sort of data file from what was on the flash chip, the software being used will need to natively, or through a plugin, assist in capturing data to/from flash chips. This is needed because the software in stock form only makes the data used in the protocols human readable in byte form. Stitching together 1MB (~1 million bytes) would not be feasible by hand, especially since there are other bytes in the protocol.

In Figure 5-3 above the SPI pins for the flash chip have been identified by looking in the datasheet acquired online. Those target SPI pins were then connected to a Saleae logic analyzer. Saleae is the brand the author is most familiar with and they make a logic analyzer at a price point lucrative to hobbyists, it has been covered in another section. The process for capture is as follows:

1. Start capturing data through the logic analyzer
2. Boot up PC (or device)
3. End capture
4. Analyze data

What should be happening is that during boot the PC reads data from the flash chip. Based on prior experience it is known that this is where the BIOS is located. The goal is to capture everything the PC reads from this flash chip. In Figure 5-4 below are some screen shots from an attempt to use Saleae to discover data being read from the flash chip. In the end data was not able to be read successfully. It is believed this was due to inadequate speed capabilities.


Logic

We're sorry, but the device was not able to keep up with this sample rate. If you continue to see this message, please contact Saleae technical support.

Would you like to keep the data collected so far?

[✓] Lower the sample rate

Saleae Logic 1.2.18 – [Connected] – [8 MHz Digital, 25 s]

Speed (Sample Rate): 8 MS/s

Duration (Record data for): 25 Seconds

Voltage: 1.8V to 3.6V | 3.6V to 5.0V

Decoded Protocols:

```
?? CMD='15'
?? CMD='15'
?? CMD='31'
'255'
?? CMD='15'
?? CMD=_
?? CMD='31'
'255'
```

Keep
Discard
Start
00 Channel 0 SPI Flash - Chip Select
01 Channel 1 SPI Flash - Clock
02 Channel 2 SPI Flash - MOSI
03 Channel 3 SPI Flash - MISO
04 Channel 4 SPI Flash - D2
05 Channel 5 SPI Flash - D3
Logic16
Estimated Memory Usage: 0 B - 5.0 GB
Channels
Clear
Options
+50 µs
?? CMD='31'
CMD=0x1F
'255'
'232'
Annotations
Timing Marker Pair
Analyzers
SPI Flash
Search Protocols
2 MHz, 20 M ...
4 MHz, 40 M ...

Figure 5-4: Saleae Logic being used to 'snoop' data to/from the flash chip.

Even if passive capture works there is a glaring shortcoming. The only data discoverable will be what the master device is reading/writing to the flash chip. This can force users to try and manipulate the device through whatever interface is available to try and get the most reads/writes to the flash chip available for discovery. In the case of a firmware image being on the flash chip its possible this is stored for the sole purpose of recovery or updates. If the design architecture of the target device is as follows: download a firmware update into the flash chip, run a CRC or other security check on the firmware while it's on the flash chip, then update the processor's internal flash chip with the firmware; it won't be possible to access the firmware image with passive techniques with the exception of getting extremely lucky and snooping during a firmware update. Developing an understanding of the design architecture is difficult without prior knowledge or moving past passive capture.

5.1.2.1.2. Active Capture from On-Board Flash

Another method of establishing what is on the flash chip is to read it by introducing hardware that acts as a master. Since the flash chip is mostly a dumb chip that simply responds to commands and those commands are readily available in datasheets, this is often possible without much reverse engineering. The hardware required for this needs to be something that can actively write commands out across a SPI bus. An Arduino would likely work but for the BIOS chip example, a device called a Bus Pirate which was used. The Bus Pirate is a small single-board device used for programming, debugging, and analyzing microcontrollers and is covered in a different section. Since it is an open source project many hobbyists, tinkers, and engineers have written plugins to extend its functionality. One of those extended functionalities is to read entire flash chips.

While it may be possible to connect to the chip while its on the board it is HIGHLY DISCOURAGED. There are two reasons for this:

1. Power: The flash chip needs to be powered in order to read from it. If the chip is powered while still on the circuit board it will be likely that everything else on the board will be powered as well, or at least everything on the same voltage bus. Depending on the target this can be an enormous strain for hardware which will likely result in damage.
2. Race Conditions: Provided that the entire device is successfully powered with no damage to equipment, a race condition is likely to occur. Both the introduced

hardware and the targets hardware will be trying to behave as a SPI master and end up stepping all over each other while trying to access the flash.

Because of the risk of reading the chip while still on the board it is recommended to de-solder the chip. Depending on the form factor this can be incredibly difficult without damaging surrounding components or the circuit board. Since the BIOS flash chip on the laptop was relatively large a pair of soldering iron tweezers with wide blades as seen in Figure 5-5 was used to remove the chip.

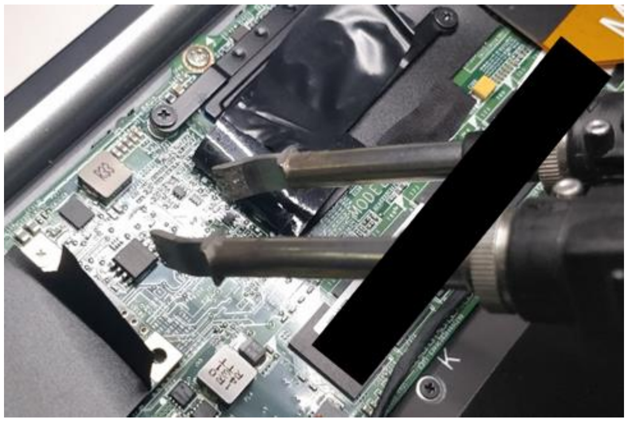

[redacted]

Figure 5-5: Using a soldering iron to de-solder the flash chip.

Once de-soldered the chip is ready to be read. The first tool selected is called a Xeltec. The Xeltec device uses various sockets to accommodate different form factors chips come in. The required socket was not on hand for this particular chip but a similar one was available seen in Figure 5-6. A read was attempted using the hardware available, but it ended up failing. It is not recommended using anything other than the correct socket at the risk of damaging the component but in the famous words “do as I say, not as I do”.

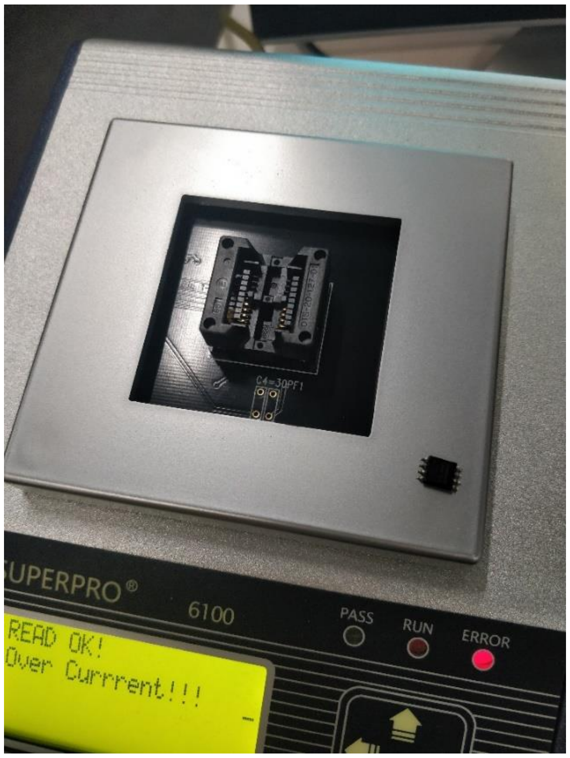


Figure 5-6: Xeltec device with the socket plugged in and the target flash chip laying on top.

This read attempt failed so new methods and tools would need to be attempted. The next suitable tool on hand was the Bus Pirate. The Bus Pirate, seen in Figure 5-7, has already been mentioned but as a refresher; it’s an open source tool at a much lower price point that serves as a great entry tool for hobbyists and tinkerers.

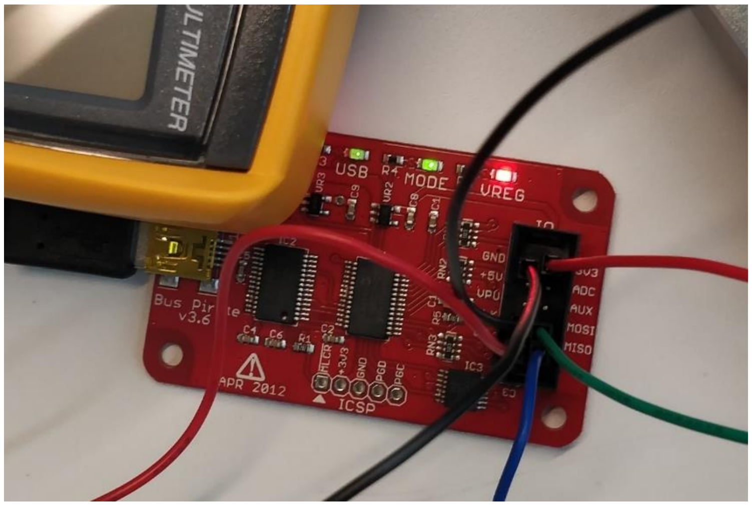


Figure 5-7: The Bus Pirate

Leads were connected between the Bus Pirate and the flash chip as seen in Figure 5-8. The Bus Pirate is then connected to a computer via USB and is able to power the flash chip natively.

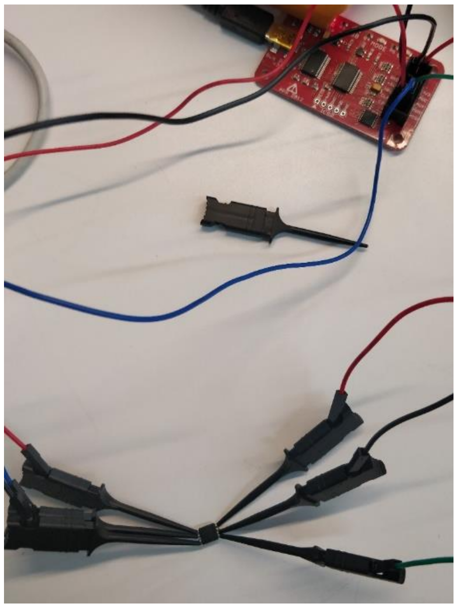
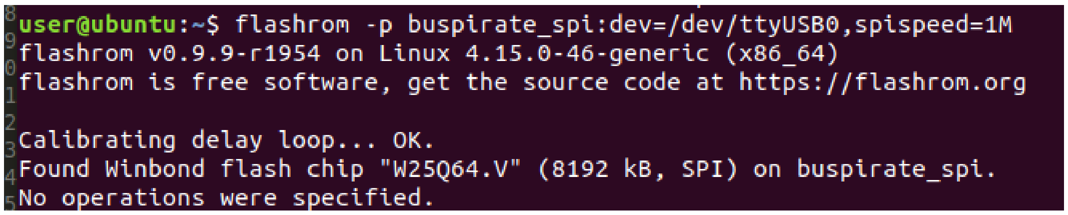


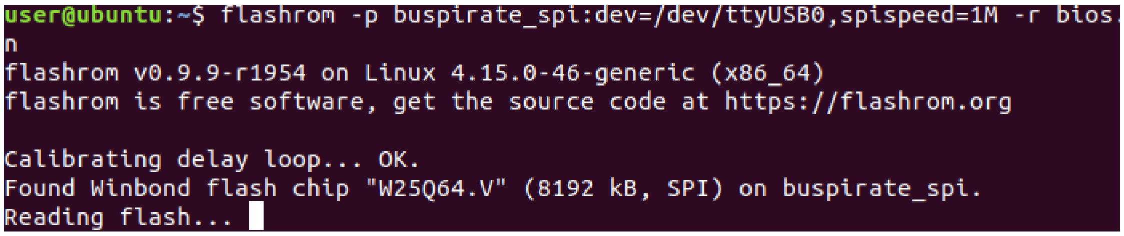
Figure 5-8: Flash chip connected to the Bus Pirate

The Bus Pirate is just a hardware interface and unable to read the flash chip by itself. Consider it like a Network Interface Card (NIC); by itself unable to communicate but software interacts through it to communicate with devices. A tool called flashrom is able to leverage bus pirate in order to read the flash chip. Screenshots from using flashrom have been captured and displayed in Figure 5-9 and 5-10.



```
user@ubuntu:~$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M
flashrom v0.9.9-r1954 on Linux 4.15.0-46-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Calibrating delay loop... OK.
Found Winbond flash chip "W25Q64.V" (8192 kB, SPI) on buspirate_spi.
No operations were specified.
```

Figure 5-9: Flashrom in use



```
user@ubuntu:~$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M -r bios.
n
flashrom v0.9.9-r1954 on Linux 4.15.0-46-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Calibrating delay loop... OK.
Found Winbond flash chip "W25Q64.V" (8192 kB, SPI) on buspirate_spi.
Reading flash...
```

Figure 5-10: Flashrom reading the chip

Flashrom worked as intended and a complete capture of the contents of the flash chip was acquired for further analysis. The flash chip is also preserved in form and function so that the original target can be put back together. The flash chip is re-soldered onto the mother board as seen in Figure 5-11.

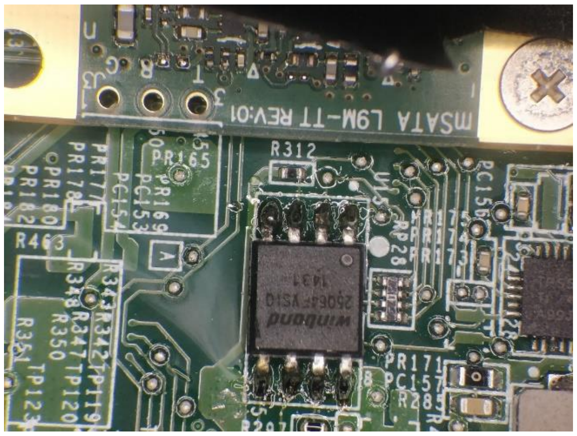


Figure 5-11: Flash chip re-soldered onto the board. The solder joints are a bit rough looking but they aren't shorted and appear to make good contact.

It’s important to ensure that the solder joints make good contact and that they don’t jump or short any pins to anything. With that in mind everything looks good and its time to power on the target to make sure it still works. The laptop is powered up and it looks like it boots up without issue as seen in Figure 5-12.

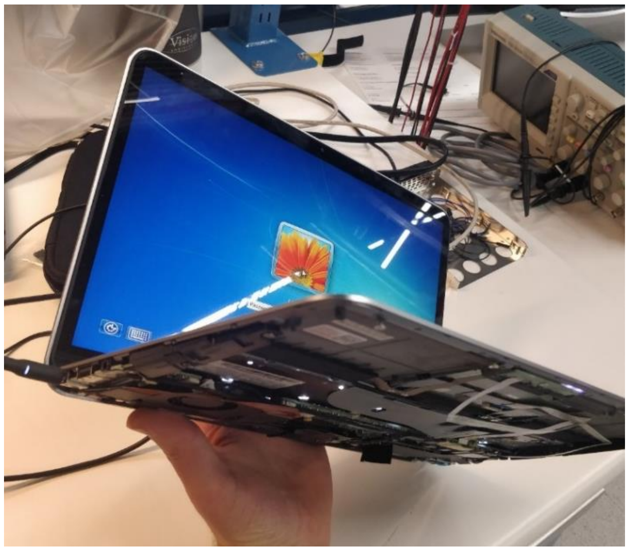


Figure 5-12: After de-soldering the flash chip, reading it, and re-soldering the flash chip the target still works.

5.1.3. Capturing Firmware Conclusion

Now that the contents have successfully ben pulled off the flash chip analysis can be performed. Analysis will be covered in the next section. The information provided in this chapter is not all inclusive and if steps fail to provide a copy of firmware other methods will need to be attempted before analysis can be performed.

CHAPTER 6: ANALYSIS OF CAPTURED FIRMWARE

Analysis of firmware is a broad term and what form analysis takes is dependent on what the user/researcher hopes to achieve. In the opinion of the author the majority of firmware analysis is motivated by the following:

- Vulnerability Discovery: The researchers wants to identify vulnerabilities within the firmware.
- What's in the Box?: The researcher wants to know what subcomponents make up the firmware image or what the intent of a suspect piece of firmware is.
- Modification: The researcher would like to modify the existing firmware for benign or malicious purposes.

Because of the breadth of firmware analysis it is difficult to develop a solid framework to cover everything. The author presents the following as starting points for any analysis to be performed:

- Firmware file type: Is it an elf, zip, Motorola S-record?
- Processor type/architecture: Is the processor an ARM, PowerPC, Intel architecture?
- Is there an OS within the firmware: Unix based, FreeRTOS, VxWorks?

Other frameworks have been attempted [43] but appear to be combinations of other existing tools stacked together and only go after low hanging fruit in the analysis process. Other tools attempt to cover the analysis process [44] but still rely on the user's creativity and are limited by their capabilities.

6.1. Analysis showcase: Investigating a Computer BIOS

For the sake of continuity, analysis will be performed on firmware extracted in section 5.1.2. This analysis is presented as an exhibition of the creativity demanded in firmware analysis. The binaries retrieved from the flash chip in section 5.1.2 are a bit of a grey area on whether they count as firmware or software since the target device is a PC. Regardless of the intended target device, the obfuscation of binaries make it a suitable example.

6.1.1. Background

In this example the author examines the contents of a flash chip from the motherboard of a PC. Based on prior experience it is known within a high level of confidence this chip contains the BIOS (Basic Input/Output System) of the computer. Many readers may be familiar with a BIOS but for those who aren't; a BIOS provides a computer with bare minimum functionalities such as display and hardware drivers like USB, hard drive, keyboard, etc. needed by a computer upon boot up and is read in from the flash chip upon power on.

More specifically the author intends to investigate the CompuTrace agent (also known as "Absolute Home & Office" or "LoJack for Laptops") that resides within the BIOS [45]. The tool is a Remote Access Tool (RAT) used for theft prevention and recovery of laptops. CompuTrace is installed into the BIOS from the manufacture and if activated with a paid subscription, can assist owners and law enforcement of disabling, tracking, and managing data from the laptop. Because CompuTrace resides in the BIOS it can persist even with a hard drive replacement and bypass traditional security tools.

Based on prior research [46], [47] CompuTrace works as follows:

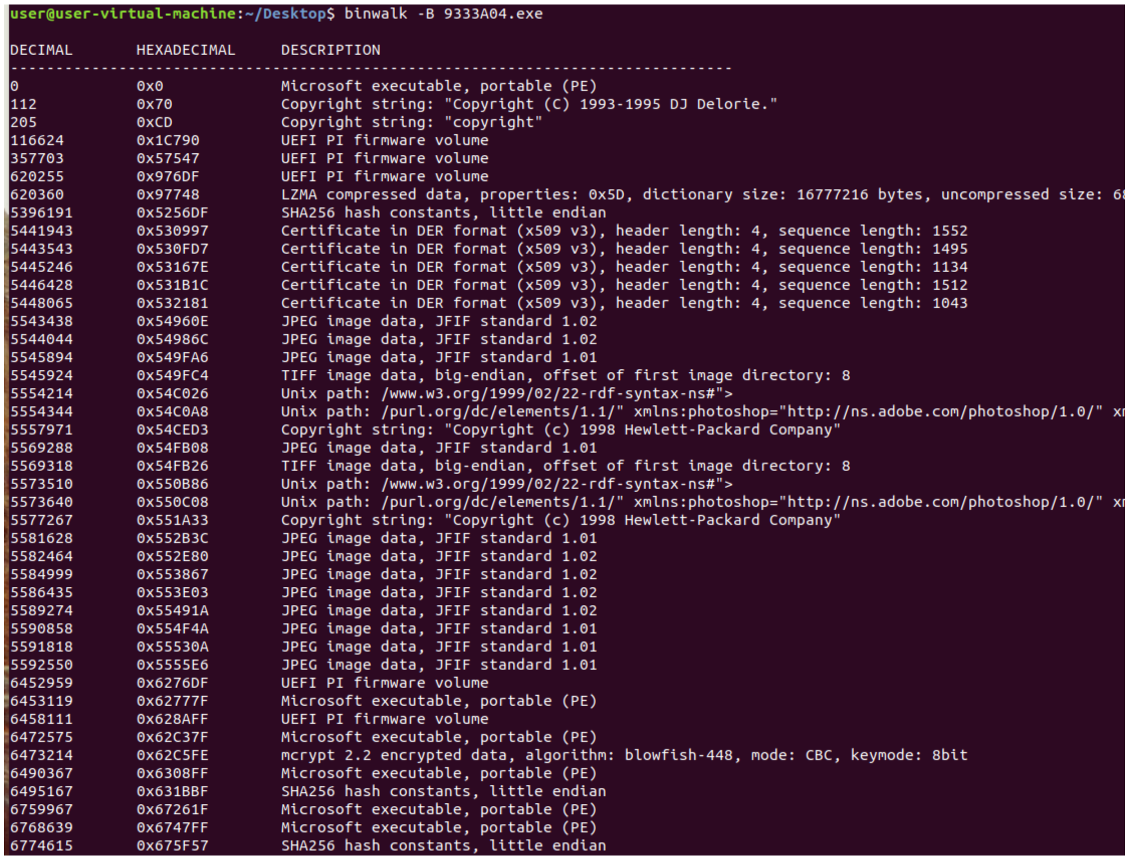
Step 1: CompuTrace code is loaded from the BIOS and scans available hard drives. It searches for a windows installation path and then for the application called autochk.exe. Once found CompuTrace injects code into autochk.exe

Step 2: On boot autochk.exe runs and drops rpcnetp.exe and registers it as a windows service.

Motivations for investigating CompuTrace is for better understanding of how it works and potentially to leverage it to drop an alternative payload for proof of concept work. This work is not novel, and malware exists in the wild that leverages CompuTrace called LoJax [48] (a play on words to LoJack).

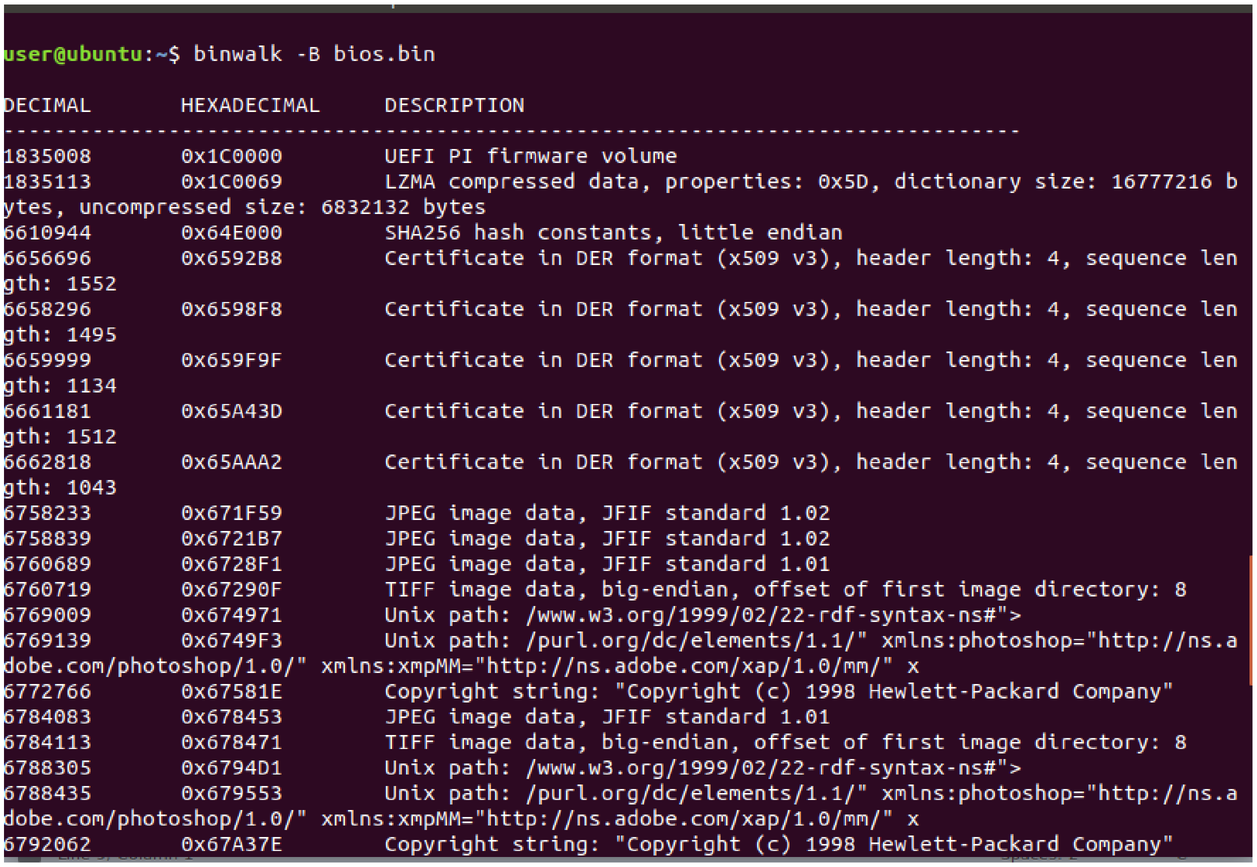
6.1.2. Analysis

This analysis picks up at the end of section 5.1.2 where the contents of the flash chip was read off. At this point a 8192KB binary file that is a 1:1 copy of the contents of the flash chip is obtained. Concurrently a BIOS update was retrieved from the manufacturer of the laptop to create a larger surface area for analysis. Upon review of BIOS version A04 update from the OEM it was discovered the file was a 18MB PE file (executable) that contained a lot more than just the BIOS. Initial analysis of the BIOS updater was brief and shelved to a later date due to being twice the size of the BIOS flash chip and the author's preference to not delve into PE file analysis. As a quick sanity check the contents of both files were analyzed with Binwalk using the -B flag. The results of this can be seen in Figure 25 and Figure 26 for the BIOS updater executable and the BIOS binary file respectively.



```
user@user-virtual-machine:~/Desktop$ binwalk -B 9333A04.exe

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             Microsoft executable, portable (PE)
112           0x70            Copyright string: "Copyright (C) 1993-1995 DJ Delorie."
205           0xCD            Copyright string: "copyright"
116624        0x1C790         UEFI PI firmware volume
357703        0x57547         UEFI PI firmware volume
620255        0x976DF         UEFI PI firmware volume
620360        0x97748         LZMA compressed data, properties: 0x5D, dictionary size: 16777216 bytes, uncompressed size: 6
5396191       0x5256DF        SHA256 hash constants, little endian
5441943       0x530997        Certificate in DER format (x509 v3), header length: 4, sequence length: 1552
5443543       0x530FD7        Certificate in DER format (x509 v3), header length: 4, sequence length: 1495
5445246       0x53167E        Certificate in DER format (x509 v3), header length: 4, sequence length: 1134
5446428       0x531B1C        Certificate in DER format (x509 v3), header length: 4, sequence length: 1512
5448065       0x532181        Certificate in DER format (x509 v3), header length: 4, sequence length: 1043
5543438       0x54960E        JPEG image data, JFIF standard 1.02
5544044       0x54986C        JPEG image data, JFIF standard 1.02
5545894       0x549FA6        JPEG image data, JFIF standard 1.01
5545924       0x549FC4        TIFF image data, big-endian, offset of first image directory: 8
5554214       0x54C026        Unix path: /www.w3.org/1999/02/22-rdf-syntax-ns#">
5554344       0x54C0A8        Unix path: /purl.org/dc/elements/1.1/" xmlns:photoshop="http://ns.adobe.com/photoshop/1.0/" x
5557971       0x54CED3        Copyright string: "Copyright (c) 1998 Hewlett-Packard Company"
5569288       0x54FB08        JPEG image data, JFIF standard 1.01
5569318       0x54FB26        TIFF image data, big-endian, offset of first image directory: 8
5573510       0x550B86        Unix path: /www.w3.org/1999/02/22-rdf-syntax-ns#">
5573640       0x550C08        Unix path: /purl.org/dc/elements/1.1/" xmlns:photoshop="http://ns.adobe.com/photoshop/1.0/" x
5577267       0x551A33        Copyright string: "Copyright (c) 1998 Hewlett-Packard Company"
5581628       0x552B3C        JPEG image data, JFIF standard 1.01
5582464       0x552E80        JPEG image data, JFIF standard 1.02
5584999       0x553867        JPEG image data, JFIF standard 1.02
5586435       0x553E03        JPEG image data, JFIF standard 1.02
5589274       0x55491A        JPEG image data, JFIF standard 1.02
5590858       0x554F4A        JPEG image data, JFIF standard 1.01
5591818       0x55530A        JPEG image data, JFIF standard 1.01
5592550       0x5555E6        JPEG image data, JFIF standard 1.01
6452959       0x6276DF        UEFI PI firmware volume
6453119       0x62777F        Microsoft executable, portable (PE)
6458111       0x628AFF        UEFI PI firmware volume
6472575       0x62C37F        Microsoft executable, portable (PE)
6473214       0x62C5FE        mcrypt 2.2 encrypted data, algorithm: blowfish-448, mode: CBC, keymode: 8bit
6490367       0x6308FF        Microsoft executable, portable (PE)
6495167       0x631BBF        SHA256 hash constants, little endian
6759967       0x67261F        Microsoft executable, portable (PE)
6768639       0x6747FF        Microsoft executable, portable (PE)
6774615       0x675F57        SHA256 hash constants, little endian
```

Figure 6-1: Analysis of the BIOS update utility executable using Binwalk with the -B flag



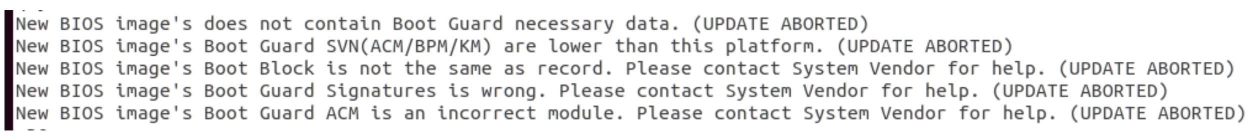
```
user@ubuntu:~$ binwalk -B bios.bin

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
1835008       0x1C0000        UEFI PI firmware volume
1835113       0x1C0069        LZMA compressed data, properties: 0x5D, dictionary size: 16777216 b
ytes, uncompressed size: 6832132 bytes
6610944       0x64E000        SHA256 hash constants, little endian
6656696       0x6592B8        Certificate in DER format (x509 v3), header length: 4, sequence len
gth: 1552
6658296       0x6598F8        Certificate in DER format (x509 v3), header length: 4, sequence len
gth: 1495
6659999       0x659F9F        Certificate in DER format (x509 v3), header length: 4, sequence len
gth: 1134
6661181       0x65A43D        Certificate in DER format (x509 v3), header length: 4, sequence len
gth: 1512
6662818       0x65AAA2        Certificate in DER format (x509 v3), header length: 4, sequence len
gth: 1043
6758233       0x671F59        JPEG image data, JFIF standard 1.02
6758839       0x6721B7        JPEG image data, JFIF standard 1.02
6760689       0x6728F1        JPEG image data, JFIF standard 1.01
6760719       0x67290F        TIFF image data, big-endian, offset of first image directory: 8
6769009       0x674971        Unix path: /www.w3.org/1999/02/22-rdf-syntax-ns#">
6769139       0x6749F3        Unix path: /purl.org/dc/elements/1.1/" xmlns:photoshop="http://ns.a
dobe.com/photoshop/1.0/" xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/" x
6772766       0x67581E        Copyright string: "Copyright (c) 1998 Hewlett-Packard Company"
6784083       0x678453        JPEG image data, JFIF standard 1.01
6784113       0x678471        TIFF image data, big-endian, offset of first image directory: 8
6788305       0x6794D1        Unix path: /www.w3.org/1999/02/22-rdf-syntax-ns#">
6788435       0x679553        Unix path: /purl.org/dc/elements/1.1/" xmlns:photoshop="http://ns.a
dobe.com/photoshop/1.0/" xmlns:xmpMM="http://ns.adobe.com/xap/1.0/mm/" x
6792062       0x67A37E        Copyright string: "Copyright (c) 1998 Hewlett-Packard Company"
```

Figure 6-2: Analysis of the binary retrieved from the flash chip using binwalk with the -B flag

The value the author obtained from this comparison was a peace of mind that the flash chip did indeed contain the BIOS. This conclusion was reached by the files being similar in that they both included the same or similar copyright strings, file paths, and number of JPEGs found.

A core principle of most reverse engineering or analysis approaches is to take the "lowest hanging fruit"; in other words, learn as much as can be learned using the easiest methods first. Using this approach, the strings command piped into a text file is a good starting point. Various strings are found but no references to CompuTrace or other potential artifacts are identified. Strings that are found are references to Hewlett-Packard which is interesting given the computer is a Dell. References to and emails from Quanta are found; upon researching Quanta it is discovered that they are a Taiwan based computer manufacturer. Other unique strings are also identified such as XXXXX- XXXXX- XXXXX- XXXXX- XXXXX (obfuscated for security purposes) which reminds the author of a Windows key although this was not confirmed. There was also a reference to OpenSSL 0.9.8l which has vulnerabilities but upon further investigation the vulnerabilities appear to be of medium consequence and its not clear how this system would be impacted by the vulnerability as this was not the goal of the analysis.

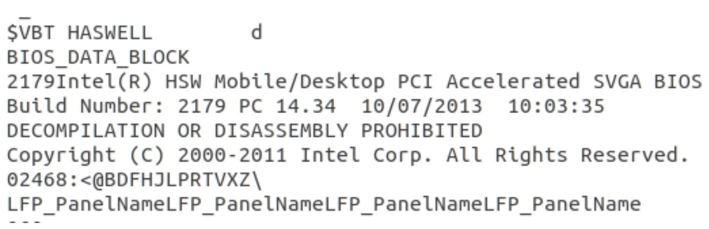
Initial analysis of the file by observing strings of interest did not prove fruitful for the author. Further analysis is now required, and the next step is also an easy one which is to attempt to extract data using Binwalk with the -e flag. Using this flag Binwalk will attempt to extract any file systems or types that can be identified. The result of this automated extraction were dozens of files. Again, strings was run on each file and the output was investigated. This results in many more strings of interest; some the author chooses to store the string and file away for later analysis if it proves to be relevant. At this stage of analysis, it is difficult to know what is useful and what is not. An example of an artifact worthy of a mental note is in Figure 6-3. These strings hint at some security functionality of the BIOS that may worth further investigation once other techniques of analysis are exhausted.



```
New BIOS image's does not contain Boot Guard necessary data. (UPDATE ABORTED)
New BIOS image's Boot Guard SVN(ACM/BPM/KM) are lower than this platform. (UPDATE ABORTED)
New BIOS image's Boot Block is not the same as record. Please contact System Vendor for help. (UPDATE ABORTED)
New BIOS image's Boot Guard Signatures is wrong. Please contact System Vendor for help. (UPDATE ABORTED)
New BIOS image's Boot Guard ACM is an incorrect module. Please contact System Vendor for help. (UPDATE ABORTED)
```

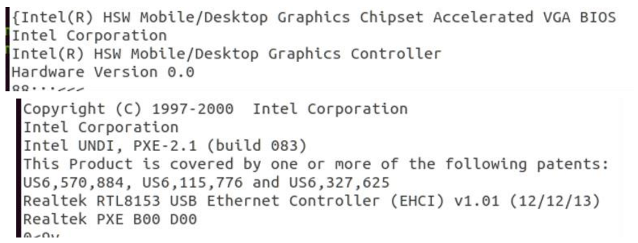
Figure 6-3: Artifact of interest extracted from the BIOS image.

Of the dozens of files generated from first extraction using Binwalk, the most interesting file and primary candidate for analysis is a file called “@1C0069”. This file is named after the hexadecimal start address Binwalk discovered the file from the original image. This file is the primary candidate of interest due to the number of strings found that are hypothesized to be relevant. Strings that the author identifies as strings of interest and indicate this is the file of the actual BIOS are as follows: Figure 6-4 references various build versions which indicate there may be multiple versions of the BIOS available. Figure 6-5 has strings referencing peripherals which makes sense if this file is the BIOS.



```
$VBT HASWELL        d
BIOS_DATA_BLOCK
2179Intel(R) HSW Mobile/Desktop PCI Accelerated SVGA BIOS
Build Number: 2179 PC 14.34  10/07/2013  10:03:35
DECOMPILATION OR DISASSEMBLY PROHIBITED
Copyright (C) 2000-2011 Intel Corp. All Rights Reserved.
02468:<@BDFHJLPRTVXZ\
LFP_PanelNameLFP_PanelNameLFP_PanelNameLFP_PanelName
```

Figure 6-4:Strings from file @1C0069 references difference builds.

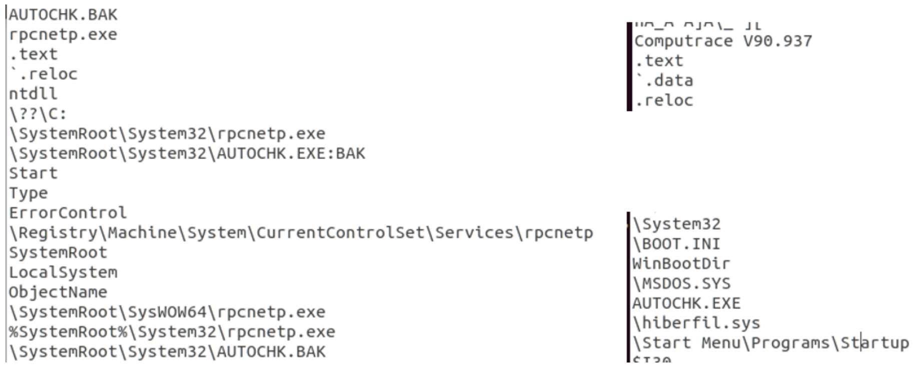


```
{Intel(R) HSW Mobile/Desktop Graphics Chipset Accelerated VGA BIOS
Intel Corporation
Intel(R) HSW Mobile/Desktop Graphics Controller
Hardware Version 0.0
```

```
Copyright (C) 1997-2000  Intel Corporation
Intel Corporation
Intel UNDI, PXE-2.1 (build 083)
This Product is covered by one or more of the following patents:
US6,570,884, US6,115,776 and US6,327,625
Realtek RTL8153 USB Ethernet Controller (EHCI) v1.01 (12/12/13)
Realtek PXE B00 D00
```

Figure 6-5: Strings references peripherals are likely to be found in a BIOS indicating this file has BIOS information in it.

These strings are relevant in that they indicate the correct file is being investigated however do not indicate CompuTrace is in the file. Towards the end of the list of strings retrieved from file “@1C0069” are strings hypothesized to be relevant to CompuTrace based on prior research. Figure 6-6 shows three snippets of strings discovered relevant to CompuTrace. Not only do strings explicitly match “Computrace V90.937” and the two artifacts “AUTOCHK.BAK” and “rpcnetp.exe” but also have windows file paths that are part of the kill chain CompuTrace uses to drop the artifacts.



```
AUTOCHK.BAK
rpcnetp.exe
.text
`.reloc
ntdll
\??\C:
\SystemRoot\System32\rpcnetp.exe
\SystemRoot\System32\AUTOCHK.EXE:BAK
Start
Type
ErrorControl
\Registry\Machine\System\CurrentControlSet\Services\rpcnetp
SystemRoot
LocalSystem
ObjectName
\SystemRoot\SysWOW64\rpcnetp.exe
%SystemRoot%\System32\rpcnetp.exe
\SystemRoot\System32\AUTOCHK.BAK
```

```
Computrace V90.937
.text
`.data
.reloc
```

```
\System32
\BOOT.INI
WinBootDir
\MSDOS.SYS
AUTOCHK.EXE
\hiberfil.sys
\Start Menu\Programs\Startup
```

Figure 6-6: Three snippets of strings discovered in file "@1C0069"

Other strings that may be of interest are “.text”, “.reloc”, and “.data”. These are strings that are commonly associated with sections of a PE (.exe) file. Based on this grouping of strings, analysts should have a strong indication that this is the target file, or at least a file of importance, in the investigation of CompuTrace.

For the sake of brevity some details are omitted but based off the raw data, strings, and format of windows executable files it is believed multiple executable files are embedded in this one file “@1C0069”. Out of the multiple executables, two are believed to be a part of CompuTrace and are targets for further analyzing. To analyze these executables further, such as in a disassembler or debugger, they will require further extraction.

The author is not aware of any tool to automatically extract the executables and Binwalk also came up short for extracting them from the file. The executables are suspected to be of file format “Microsoft PE” which is well documented [49]. Because the file format is so well documented (see Figure 6-7) it may be possible to simply copy and paste the data of the file inside a hex viewer/editor. When copying and pasting raw data the author found it easy to identify the start of the file as it begins with the string “MZ” but finding the end of the file was a bit trickier. Since the files were ‘stacked’ and appeared as one continuous data stream, copying and pasting from the start of one file to the start of the next file (“MZ” to “MZ” flag) was done.

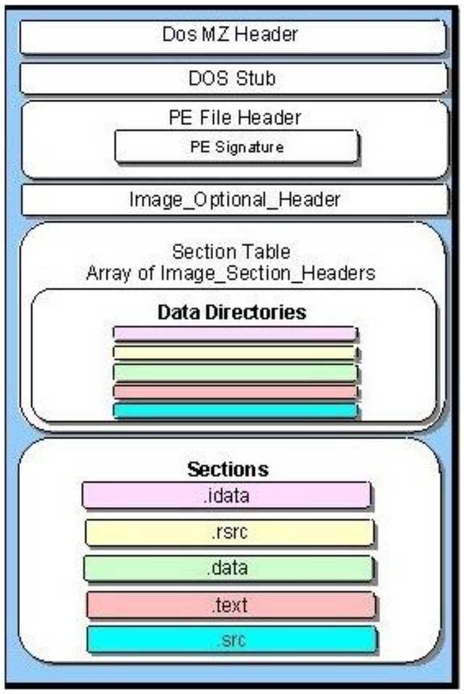
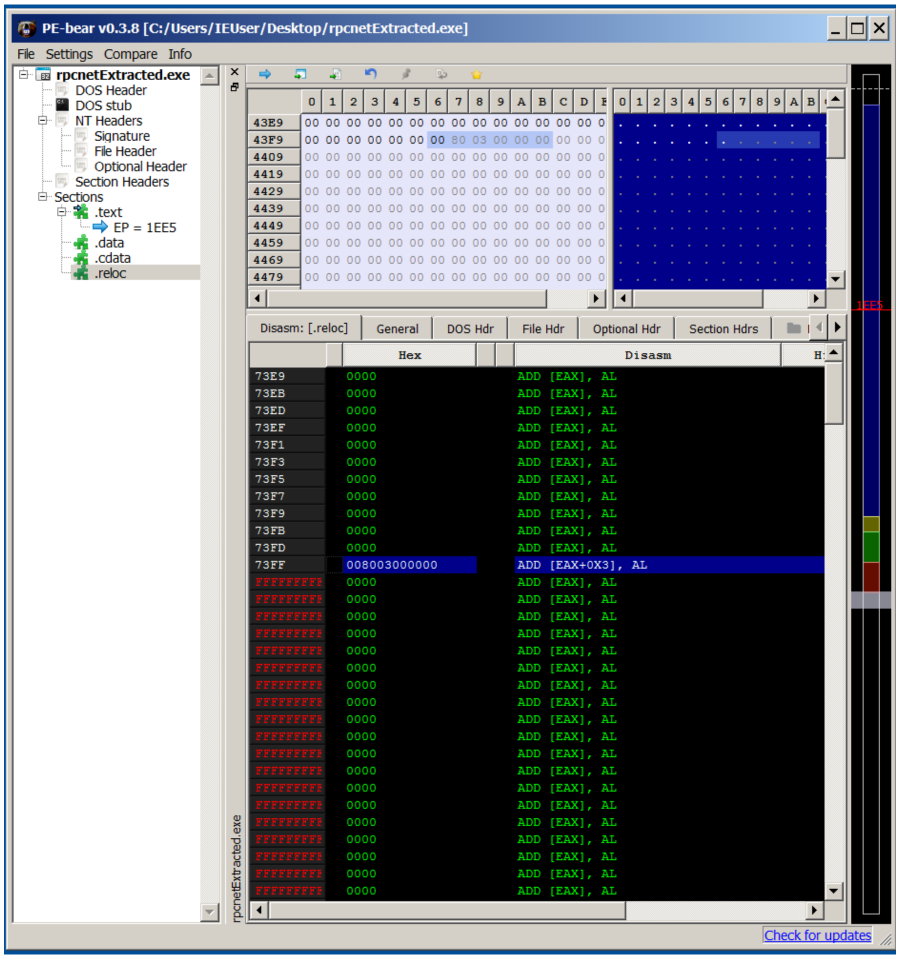


Figure 6-7: PE file format [50]

The resulting file that was copied from the start of the target file and ended at the start of the subsequent file was extracted and ready for analysis. All analysis attempts using debuggers and disassemblers available to the author (Binary Ninja, Ghydra, OllyDbg) failed and the file was not recognized or recognized with errors. At this point it is clear the file needs more refinement, or the file has been captured with faults. With no desire to dig deeper and write custom extraction software other tools were investigated. A tool called PE bear [51] was found to be useful in providing details concerning PE files. Some of that information includes start and end address for the various sections (.text, .data, .cdata, .reloc). Figure 6-8 is a screenshot of PE-bear with the target file loaded and the end address identified.


PE-bear v0.3.8 [C:/Users/IEUser/Desktop/rpcnetExtracted.exe]

File Settings Compare Info

```
rpcnetExtracted.exe
    DOS Header
    DOS stub
    NT Headers
        Signature
        File Header
        Optional Header
    Section Headers
    Sections
        .text
            EP = 1EE5
        .data
        .cdata
        .reloc
```

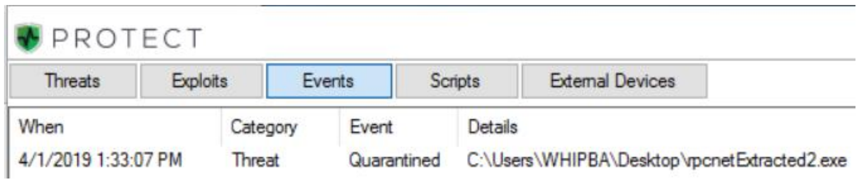
```
      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E
43E9  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
43F9  00 00 00 00 00 00 00 80 03 00 00 00 00 00 0
4409  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4419  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4429  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4439  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4449  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4459  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4469  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
4479  00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
```

Disasm: [.reloc] | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs

```
          Hex                Disasm
73E9      0000               ADD [EAX], AL
73EB      0000               ADD [EAX], AL
73ED      0000               ADD [EAX], AL
73EF      0000               ADD [EAX], AL
73F1      0000               ADD [EAX], AL
73F3      0000               ADD [EAX], AL
73F5      0000               ADD [EAX], AL
73F7      0000               ADD [EAX], AL
73F9      0000               ADD [EAX], AL
73FB      0000               ADD [EAX], AL
73FD      0000               ADD [EAX], AL
73FF      008003000000       ADD [EAX+0X3], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
FFFFFFFF  0000               ADD [EAX], AL
```

Figure 6-8: PE-bear being used to identify start and end addresses of various locations on the extracted PE file

Using PE-bear the author was able to identify the true 'end' to the executable and use that information to correctly extract the target PE file. A point of interest is that when the file was correctly extracted and saved locally on the author's PC the anti-virus running on the host machine identified the file as malicious and quarantined the file. This anti-virus report from Cylance can be seen in Figure 6-9.


PROTECT

Threats | Exploits | Events | Scripts | External Devices

When	Category	Event	Details
4/1/2019 1:33:07 PM	Threat	Quarantined	C:\Users\WHIPBA\Desktop\rpcnetExtracted2.exe

Figure 6-9: Cylance detected the extracted file as malicious and quarantined it.

Once Cylance was prevented from quarantining the target file post extraction, the file was able to be loaded into a disassembler and deemed ready for analysis. The actual analysis on this file was brief and mostly consisted of confirming other researchers reports artifacts were present. Reasons for the limited analysis conducted are discussed in the next section.

6.1.3. Analysis Conclusions

From capture, to extraction, to analysis proved to be an arduous endeavor. Readers questioning why more analysis was not done or to what effect the author hoped to achieve are not misplaced in their questioning. Further analysis was cut short due to the fact the author could not get artifacts to manifest on the host machine. The original goal of the author was to modify the CompuTrace RAT so that those changes would manifest themselves on the host machine. Without the ability to get the original and untouched version of CompuTrace to persist on the host, there was no way to prove functionally that it could be done and the author didn't want to troubleshoot why CompuTrace was not working as it allegedly was supposed to.

CHAPTER 7: SUMMARY & THE PITFALLS OF ANALYSIS TOOLKITS & FRAMEWORKS

In this thesis the author covered some common tools used in analysis and demonstrated some of the diversity of scenarios encountered when obtaining and performing analysis of firmware. It should be clear at this point the magnitude of leeway and agility researchers must use to accomplish their task. Given that so many different design paradigms, processor architects, and operating systems can be encountered on embedded devices it will remain difficult to substitute researcher's intuition and creativity with a formulated process of analysis.

7.1. Tool Shortcomings

All analysis frameworks and toolkits investigated by the author [43], [44], [52], [53], [54] would not assist in capturing firmware and would have not significantly aided in the analysis of firmware. It was found some tools investigated simply combined tools and/or argument flags in a single step which arguably doesn't quite qualify as a toolkit on its own. In other instances, the only real value being provided was a better GUI to other tools that already exist.

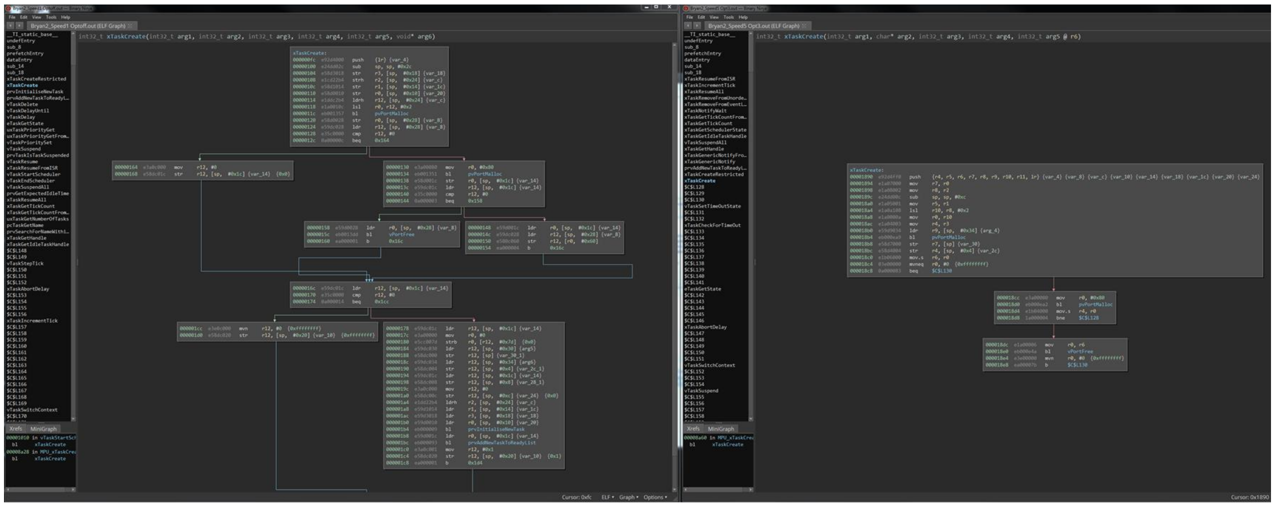
Given the broad field of firmware analysis it is unlikely to have a single tool to cover all aspects of the process. Even when the scope is narrowed to smaller goals such as full extraction or file identification it is unlikely one tool can cover every scenario. While hypothetically a toolkit may be able to extract firmware files for 88% of those encountered it begs the question "is it useful?" when using other tools in varying steps can accomplish the same and more.

CHAPTER 8: FUTURE WORK: MACHINE LEARNING APPLIED TO FIRMWARE ANALYSIS

Various aspects regarding analysis of embedded devices have been covered in this thesis. One of those areas, and one of the most difficult, provenance discovery may be improved using machine learning. Various methodologies and techniques have been tried or proposed [55], [56], [57], [58], [59], [60], [61] such as binary to binary comparison, code to binary comparison, and binary to code comparison to try and solve the question of what subcomponents make up an embedded firmware file. While the other methods claim to achieve decent results, the author believes many of these techniques will not scale well. This belief is further hardened based on professional experience dealing with commercial software companies claiming to achieve high levels of accuracy in provenance discovery only to have performances fall short of promises.

8.1. Handling Large Data

One of the major difficulties in provenance discovery is the diversity of binaries that can be encountered. Even when the original code is the exact same the binaries can be different depending on processor architecture, optimization flags, compilers used, and other compiler flags used. Below in Figure 8-1 an example of this is observed in the disassembly of two binaries the author compiled from the same FreeRTOS code for an ARM architecture with the only change an optimization flag to the compiler. Not only is the number of instructions drastically different but the control flow blocks look completely different.



```
int32_t xTaskCreate(int32_t arg1, int32_t arg2, int32_t arg3, int32_t arg4, int32_t arg5, void* arg6)

xTaskCreate:
000000fc  e92d4000  push   {lr} {var_4}
00000100  e24dd02c  sub    sp, sp, #0x2c
00000104  e58d3018  str    r3, [sp,  #0x18] {var_18}
00000108  e1cd22b4  strh   r2, [sp,  #0x24] {var_c}
0000010c  e58d1014  str    r1, [sp,  #0x14] {var_1c}
00000110  e58d0010  str    r0, [sp,  #0x10] {var_20}
00000114  e1ddc2b4  ldrh   r12, [sp,  #0x24] {var_c}
00000118  e1a0010c  lsl    r0, r12, #0x2
0000011c  eb001357  bl     pvPortMalloc
00000120  e58d0028  str    r0, [sp,  #0x28] {var_8}
00000124  e59dc028  ldr    r12, [sp,  #0x28] {var_8}
00000128  e35c0000  cmp    r12, #0
0000012c  0a00000c  beq    0x164

00000164  e3a0c000  mov    r12, #0
00000168  e58dc01c  str    r12, [sp,  #0x1c] {var_14}  {0x0}

00000130  e3a00080  mov    r0, #0x80
00000134  eb001351  bl     pvPortMalloc
00000138  e58d001c  str    r0, [sp,  #0x1c] {var_14}
0000013c  e59dc01c  ldr    r12, [sp,  #0x1c] {var_14}
00000140  e35c0000  cmp    r12, #0
00000144  0a000003  beq    0x158

00000158  e59d0028  ldr    r0, [sp,  #0x28] {var_8}
0000015c  eb0013dd  bl     vPortFree
00000160  ea000001  b      0x16c

00000148  e59d001c  ldr    r0, [sp,  #0x1c] {var_14}
0000014c  e59dc028  ldr    r12, [sp,  #0x28] {var_8}
00000150  e580c060  str    r12, [r0,  #0x60]
00000154  ea000004  b      0x16c

0000016c  e59dc01c  ldr    r12, [sp,  #0x1c] {var_14}
00000170  e35c0000  cmp    r12, #0
00000174  0a000014  beq    0x1cc

000001cc  e3e0c000  mvn    r12, #0  {0xffffffff}
000001d0  e58dc020  str    r12, [sp,  #0x20] {var_10}  {0xffffffff}

00000178  e59dc01c  ldr    r12, [sp,  #0x1c] {var_14}
0000017c  e3a00000  mov    r0, #0
00000180  e5cc007d  strb   r0, [r12,  #0x7d]  {0x0}
00000184  e59dc030  ldr    r12, [sp,  #0x30] {arg5}
00000188  e58dc000  str    r12, [sp] {var_30_1}
0000018c  e59dc034  ldr    r12, [sp,  #0x34] {arg6}
00000190  e58dc004  str    r12, [sp,  #0x4] {var_2c_1}
00000194  e59dc01c  ldr    r12, [sp,  #0x1c] {var_14}
00000198  e58dc008  str    r12, [sp,  #0x8] {var_28_1}
0000019c  e3a0c000  mov    r12, #0
000001a0  e58dc00c  str    r12, [sp,  #0xc] {var_24}  {0x0}
000001a4  e1dd22b4  ldrh   r2, [sp,  #0x24] {var_c}
000001a8  e59d1014  ldr    r1, [sp,  #0x14] {var_1c}
000001ac  e59d3018  ldr    r3, [sp,  #0x18] {var_18}
000001b0  e59d0010  ldr    r0, [sp,  #0x10] {var_20}
000001b4  eb000009  bl     prvInitialiseNewTask
000001b8  e59d001c  ldr    r0, [sp,  #0x1c] {var_14}
000001bc  eb000093  bl     prvAddNewTaskToReadyList
000001c0  e3a0c001  mov    r12, #0x1
000001c4  e58dc020  str    r12, [sp,  #0x20] {var_10}  {0x1}
000001c8  ea000001  b      0x1d4
```

```
int32_t xTaskCreate(int32_t arg1, char* arg2, int32_t arg3, int32_t arg4, int32_t arg5 @ r6)

xTaskCreate:
00001890  e92d4ff0  push   {r4, r5, r6, r7, r8, r9, r10, r11, lr} {var_4} {var_8} {var_c} {var_10} {var_14} {var_18} {var_1c} {var_20} {var_24}
00001894  e1a07000  mov    r7, r0
00001898  e1a08002  mov    r8, r2
0000189c  e24dd00c  sub    sp, sp, #0xc
000018a0  e1a05001  mov    r5, r1
000018a4  e1a0a108  lsl    r10, r8, #0x2
000018a8  e1a0000a  mov    r0, r10
000018ac  e1a04003  mov    r4, r3
000018b0  e59d9034  ldr    r9, [sp,  #0x34] {arg_4}
000018b4  eb000ea9  bl     pvPortMalloc
000018b8  e58d7000  str    r7, [sp] {var_30}
000018bc  e58d4004  str    r4, [sp,  #0x4] {var_2c}
000018c0  e1b06000  mov.s  r6, r0
000018c4  03e00000  mvneq  r0, #0  {0xffffffff}
000018c8  0a000083  beq    $C$L130

000018cc  e3a00080  mov    r0, #0x80
000018d0  eb000ea2  bl     pvPortMalloc
000018d4  e1b04000  mov.s  r4, r0
000018d8  1a000004  bne    $C$L128

000018dc  e1a00006  mov    r0, r6
000018e0  eb000e4a  bl     vPortFree
000018e4  e3e00000  mvn    r0, #0  {0xffffffff}
000018e8  ea00007b  b      $C$L130
```

Figure 8-1: Different results seen in disassembly from the same source code obtained by compiling with different optimization flags

An area of research that may be able to conquer the amount of different binaries that would result from a large number of libraries being compiled for a multitude of targets and compiler flags is machine learning. Some work has already been done in this area of machine learning for provenance discovery [62] and shows promising results however the author has not been able obtain source code or sufficient details to reproduce the results. Based on professional and academic experience in this area machine learning is able to categorize data sets with a high magnitude of raw data. The author believe the challenge in leveraging this technology to library identification will be in how to convert a binary into a set of features that can be consumed by a machine learning algorithm while still maintaining information about code flow as well as accommodating multiple processor architectures which may have different instruction sets.

CHAPTER 9: REFERENCES

[1] "Interactive: The Top Programming Languages 2016," IEEE Spectrum, 2016. [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016#index/2016/1/1/1/0/1/50/1/50/1/50/1/30/1/30/1/30/1/20/1/20/1/5/1/5/1/20/1/100/.

[2] M. Hill, J. Masters, P. Ranganathan, P. Turner and J. Hennessy, "On the Spectre and Meltdown Processor Security Vulnerabiliteis," *IEEE Micro*, vol. 39, no. 2, pp. 9-19, 2019.

[3] "Is it Necessary for a software engineer to learn about CPU architecture?," Quora, 2016. [Online]. Available: https://www.quora.com/Is-it-necessary-for-a-software-engineer-to-learn-about-CPU-architecture.

[4] Balajee, "What are skilks needed to be a successful firmware engineer," Quora, 23 Nov 2015. [Online]. Available: https://www.quora.com/What-are-skills-needed-to-be-a-successful-firmware-engineer.

[5] P. Nallari, "What to Look for When Hiring an Embedded Systems Software Engineer," EASi, 15 Sep 2016. [Online]. Available: https://www.easi.com/en/insights/articles/what-to-look-for-when-hiring-an-embedded-systems-software-engineer.

[6] STM, "ARM Cortex-M4 STM32F405xx datasheet," September 2016. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32f407ig.pdf.

[7] STM, "Application Note DMA controller," June 2016. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/27/46/7c/ea/2d/91/40/a9/DM00046011.pdf/files/DM00046011.pdf/jcr:content/translations/en.DM00046011.pdf.

[8] J. Ganssle, Embedded Systems; World Class Designs, Newnes, 2007.

[9] M. Barr, Programming Embedded Systtems in C and C++, O'Reilly Media, 2009.

[10] STM, "STM32CubeMX," [Online]. Available: https://www.st.com/en/development-tools/stm32cubemx.html.

[11] C. Hosmer, "Forensic Searching and Indexing Using Python," *Python Forensics,* 2014.

[12] N. A. Hassan and R. Hijazi, "Data Hiding Under Windows OS File Structure," *Data Hiding Teqhniques in Windows OS,* 2017.

[13] "TWI Bus," i2c-bus.org, [Online]. Available: https://www.i2c-bus.org/twi-bus/.

[14] I2C Info, "I2C Info - I2C Bus, Interface and Protocol," [Online]. Available: https://i2c.info/.

[15] M. Grusin, "Serial Peripheral Interface (SPI)," SparkFun, [Online]. Available: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all.

[16] G. Kessler, "CK's File Signatures Table," GaryKessler, December 2019. [Online]. Available: https://www.garykessler.net/library/file_sigs.html.

[17] A. Frantzis, "Bless- GitHub - README," 2018. [Online]. Available: https://github.com/bwrsandman/Bless.

[18] "HexWorkshop," BreakPoint Software, [Online]. Available: http://www.hexworkshop.com/.

[19] Hex-Rays, "IDA PRO," Hex-Rays, [Online]. Available: https://www.hex-rays.com/products/ida/.

[20] Hex-Rays, "Hex-Rays Online Store," [Online]. Available: https://www.hex-rays.com/cgi-bin/quote.cgi.

[21] Binary Ninja, "Binary Ninja Purchase Page," [Online]. Available: https://binary.ninja/purchase/.

[22] Binary Ninja, "Download Binary Ninja Demo," [Online]. Available: https://binary.ninja/demo/.

[23] Ghidra, "Ghidra," NSA, [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra/.

[24] N. L. H., "The NSA Makes Ghidra, a Powerful Cybersecurity Tool, Open Source," *WIRED*, 2019.

[25] c3n3k, "IDA Educational vs Ghidra for learning malware analysis," Reddit, 6 2019. [Online]. Available: https://www.reddit.com/r/Malware/comments/bal8v2/ida_educational_vs_ghidra_for_learning_malware/.

[26] A. Das, "NSA has Open Sourced its Reverse Engineerint Tool Ghidra," itsfoss, 6 March 2019. [Online]. Available: https://itsfoss.com/nsa-ghidra-open-source/.

[27] STMicroelectronics, "Home," STM, [Online]. Available: https://www.st.com/content/st_com/en.html.

[28] IAR Systems, "Debugging and trace proves," [Online]. Available: https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/in-circuit-debugging-probes/.

[29] STM, "ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32," [Online]. Available: https://www.st.com/en/development-tools/st-link-v2.html.

[30] Lauterbach, "Lauterbach Development Tools," [Online]. Available: https://www.lauterbach.com/frames.html?home.html.

[31] IEEE, "1149.1-2013 IEEE Standard for Test Access Port and Boundary-Scan Architecture," IEEE Standards Association, [Online]. Available: https://standards.ieee.org/standard/1149_1-2013.html.

[32] D. McClellan, "JTAG Explained (finally!): Why "IOT", Software Security Engineers, and Manufacturers Should Care," Senrio, 28 9 2016. [Online]. Available: https://blog.senr.io/blog/jtag-explained.

[33] M. Ding, "Swerial Wire Debug (SWD)," Silicon Labs, 21 10 2014. [Online]. Available: https://www.silabs.com/community/mcu/32-bit/knowledge-base.entry.html/2014/10/21/serial_wire_debugs-qKCT.

[34] N. Oberli, "SWD - ARM's Alternative to JTAG," Kudelski Security Research, 16 May 2019. [Online]. Available: https://research.kudelskisecurity.com/2019/05/16/swd-arms-alternative-to-jtag/.

[35] STMicroelectronics UM0470 User Manual, "STM8 WIM communication protocol and debug module," August 2016.

[36] NXP, "BDM_ICE," [Online]. Available: https://www.nxp.com/files-static/training_pdf/27642_HCS08_BDM_ICE_WBT.pdf.

[37] C. Heffner, "Binwalk Package Description," Kali Tools, [Online]. Available: https://tools.kali.org/forensics/binwalk.

[38] "CyberChef - The Cyber Swiss Army Knife," [Online]. Available: https://gchq.github.io/CyberChef/.

[39] P. Miller, "SRecord 1.64," [Online]. Available: http://srecord.sourceforge.net/.

[40] W. Denk, "u-boot," DENX Software Engineering, [Online]. Available: https://github.com/u-boot/u-boot.

[41] CVE Deails, "BusyBox Vulnerability Statistics".

[42] P. D. Kit, "Truck Diagnostic Solutions," SpecDiag, [Online]. Available: http://specdiag.com/paccar.html.

[43] rkornmeyer, "Firmware Analysis Framework (FAF)," GitHub, Aril 2014. [Online]. Available: https://github.com/rkornmeyer/FAF.

[44] weidenba, "Firmware Analysis and Comparison Tool (FACT)," GitHub, March 2020. [Online]. Available: https://github.com/fkie-cad/FACT_core.

[45] "Absolute Software - Computrace Agent," UCLA Software Central, [Online]. Available: https://softwarecentral.ucla.edu/absolute.

[46] M. Hao, "Tracking and Analysis of the LoJack/CompuTrace Incident," NSFocus, 9 December 2019. [Online]. Available: https://nsfocusglobal.com/tracking-and-analysis-of-the-lojackcomputrace-incident/.

[47] V. Kamluk, "Absolute Computrace Revisited," Secure List, 12 February 2014. [Online]. Available: https://securelist.com/absolute-computrace-revisited/58278/.

[48] M. Archambault, "'LoJax' rootkit malware can infect UEFI, a core computer interface," Digital Trends, 27 September 2018. [Online]. Available: https://www.digitaltrends.com/computing/lojax-uefi-rootkit-infect-machines/.

[49] WikiBooks, "x86 Disassembly/Windows Executable Files," 8 Janurary 2020. [Online]. Available: https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files.

[50] Revers3r, "Malware Researcher's Handbook (Demystifying PE File)," [Online]. Available: https://resources.infosecinstitute.com/2-malware-researchers-handbook-demystifying-pe-file/#article.

[51] Hasherezade, "PE-bear," Github, 25 Jan 2019. [Online]. Available: https://github.com/hasherezade/pe-bear-releases/releases/tag/0.3.9.5.

[52] T. S. G. Solutions, "BianryAnalysisTool (BAT)," NLnet, [Online]. Available: http://www.binaryanalysis.org/old/home.

[53] armijnhemel, "binaryanalysis-ng," github, [Online]. Available: https://github.com/armijnhemel/binaryanalysis-ng.

[54] Linux Security Expert, "Manticore," [Online]. Available: https://linuxsecurity.expert/tools/manticore/.

[55] C. Ragkhitwetsagul, J. Krinke and D. Clark, "A comparison of code similarity analysers," *Empir Software Eng*, 2018.

[56] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering,* vol. 33, no. 9, p. 577, 2007.

[57] C. K. Roy and J. R. Cordy, "A Mutation/Injection-based Automatic Framework for Evaluating Code Clone Detection Tools," 2009.

[58] J. Hage, P. Rademaker and N. v. Vugt, "A comparison of plagiarism detection tools," *Technical Report UU-CS-2010-015,* 2010.

[59] E. Burd and J. Bailey, "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *The Research Institute in Software Evolution, University of Durham,* 2010.

[60] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl and S. Demeyer, "Comparison of Similarity Metrics for Refactoring Detection," 2011.

[61] J. Svajlenko and C. K. Roy, "BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench," *IEEE International Conference on Software Maintenance and Evolution,* 2016.

[62] D. Miyani, Z. Huang and D. Lie, "BinPro: A Tool for Binary Source Code Provenance," *University of Toronto,* 2017.

[63] L. Teschler, "Teardown: Inside Amazon's Echo Dot," MicroControllerTips, 5 January 2018. [Online]. Available: https://www.microcontrollertips.com/teardown-inside-amazons-echo-dot/.