3-21-2013

Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers

Zachary H. Basnight

Recommended Citation
Basnight, Zachary H., "Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers" (2013). *Theses and Dissertations*. 853.
https://scholar.afit.edu/etd/853

FIRMWARE COUNTERFEITING AND MODIFICATION ATTACKS ON PROGRAMMABLE LOGIC CONTROLLERS

THESIS

Zachry H. Basnight, First Lieutenant, USAF

AFIT-ENG-13-M-06

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

AFIT-ENG-13-M-06

FIRMWARE COUNTERFEITING AND MODIFICATION ATTACKS ON PROGRAMMABLE LOGIC CONTROLLERS

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Cyber Operations

Zachry H. Basnight, B.S.C.S.
First Lieutenant, USAF

March 2013

AFIT-ENG-13-M-06

FIRMWARE COUNTERFEITING AND MODIFICATION ATTACKS ON PROGRAMMABLE LOGIC CONTROLLERS

Zachry H. Basnight, B.S.C.S.
First Lieutenant, USAF

Approved:

Maj Jonathan Butts, PhD (Chairman)
8 Mar 13
Date

Thomas E. Dube
Maj Thomas Dube, PhD (Member)
8 MAR 13
Date

Juan Lopez Jr. (Member)
Ø8 MAR 2Ø13
Date

AFIT-ENG-13-M-06

Abstract

Recent attacks on industrial control systems (ICSs), like the highly publicized Stuxnet malware, have perpetuated a race to the bottom where lower level attacks have a tactical advantage. Programmable logic controller (PLC) firmware, which provides a software-driven interface between system inputs and physically manifested outputs, is readily open to modification at the user level. Current efforts to protect against firmware attacks are hindered by a lack of prerequisite research regarding details of attack development and implementation. In order to obtain a more complete understanding of the threats posed by PLC firmware counterfeiting and the feasibility of such attacks, this research explores the vulnerability of common controllers to intentional firmware modifications. After presenting a general analysis process that takes advantage of various techniques and methodologies applied to similar scenarios, this work derives the firmware update validation method used for the Allen-Bradley ControlLogix PLC. A proof of concept demonstrates how to alter a legitimate firmware update and successfully upload it to a ControlLogix L61. Possible mitigation strategies discussed include digitally signed and encrypted firmware as well as preemptive and post-mortem analysis methods to provide protection. Results of this effort facilitate future research in PLC firmware security through direct example of firmware counterfeiting.

Table of Contents

Page

Page

Page

List of Figures

Figure Page

Figure Page

List of Acronyms

Acronym	Definition
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
CISC	complex instruction set computing
CRC	cyclic redundancy check
cramfs	compressed read-only memory (ROM) file system
DCS	distributed control system
ELF	executable and linkable format
FRN	firmware revision number
IC	integrated circuit
ICE	in-circuit emulator
ICS	industrial control system
IDA	Interactive Disassembler
IDS	intrusion detection system
IEEE	Institute of Electrical and Electronics Engineers
IO	input/output
IP	internet protocol
IPS	intrusion prevention system
IT	information technology
JFFS2	Journalling Flash File System version 2
JTAG	Joint Test Action Group
LRC	longitudinal redundancy check
LZMA	Lempel-Ziv-Markov chain algorithm
MD	message digest

Acronym	Definition
NIST	National Institute of Standards and Technology
OS	operating system
PE	portable executable
PLC	programmable logic controller
RAM	random-access memory
RISC	reduced instruction set computing
ROM	read-only memory
RSA	Rivest, Shamir, Adleman
SCADA	supervisory control and data acquisition
SHA	secure hash algorithm
TAP	test access port
TCP	transmission control protocol
UART	universal asynchronous receiver/transmitter
XML	Extensible Markup Language
XOR	*exclusive or*
YAFFS	Yet Another Flash File System

FIRMWARE COUNTERFEITING AND MODIFICATION ATTACKS
ON PROGRAMMABLE LOGIC CONTROLLERS

I. Introduction

1.1 Background

Modern industrial applications necessitate the utilization of advanced automation and management networks collectively referred to as industrial control systems (ICSs). Such systems are responsible for the precise and consistent operation of many applications associated with national critical infrastructure. As ICSs become increasingly reliant on modern information technology (IT) solutions, including internet protocol (IP)-based networking and embedded computing, related security concerns also arise [55]. The progressive amalgamation of these technologies from two traditionally distinct cultures creates an apparent schism in the cyber security capabilities of IT and ICS environments. ICS cyber security implementations lag behind the sophistication of more dedicated IT solutions by comparison.

Cyber attacks on ICSs are increasing in number and scale [24]. Incidents like the 2010 Stuxnet worm exemplify this fact and provide insight into the future of cyber-based threats. Similar to traditional attacks on IT systems, ICS attacks are targeting lower level control to allow for more powerful and flexible system manipulation. The allure of ICS attacks, and the ultimate goal of such malicious manipulation, is the ability to elicit physical manifestations through cyber means. As the final link between cyber and physical components of ICSs, programmable logic controllers (PLCs) are critical in the proper operation of such systems. PLCs are embedded devices programmed to manage and control physical components responsive to system inputs and requirements. The lowest abstraction layer controlling PLC interpretation of programming, the firmware, represents a significant potential threat if compromised. Indeed, the malicious modification or counterfeiting of controller firmware allows complete control over the device and any physical system components under its purview.

1.2 Motivation

Defensive strategies to mitigate firmware threats must be established. In order to develop effective defense strategies, the threat must be thoroughly understood. Currently, little information detailing this threat is readily available. However, if an attacker is able to successfully manipulate the firmware on a PLC, they can directly control the behavior of the device to affect the control system while simultaneously masking such actions from the operator or control software. Although the risk exists, the extent to which an attacker is capable of exploiting the risk is unknown. There are currently no known examples of firmware modification attacks on PLCs [24]. Furthermore, research requiring the availability of malicious or counterfeit firmware lacks test samples for use in analysis. The creation of custom counterfeit firmware samples can aide in the development of detection and forensic analysis techniques.

1.3 Research Goals

The goal of this thesis is to determine the feasibility of firmware modification attacks on PLCs. Specifically, this research investigates and assesses the vulnerability of a common PLC to counterfeit firmware updates. This research proposes that common PLCs are vulnerable to such an attack as a result of design weaknesses associated with firmware update validation methods. This may be verified by a successful demonstration of counterfeit firmware uploaded to a common PLC.

1.4 Approach

To test a PLC's vulnerability to counterfeit firmware attacks, the firmware update validation method is derived through reverse engineering techniques. The firmware update validation method is analyzed for weaknesses that facilitate firmware counterfeiting. Weaknesses are exploited to create a counterfeit firmware sample that is uploaded and executed on a PLC. This approach is applied to a relevant test environment, consisting primarily of an Allen-Bradley ControlLogix L61 controller, to allow for a realistic assessment of the potential vulnerability on a common PLC.

The process to derive the firmware update validation method is based on a review of previous research describing techniques related to the reverse engineering of PLCs and other types of embedded devices. By combining and organizing these techniques, a general process is conceived for deriving the firmware update validation method of a PLC. The major steps in the process are: (i) firmware sample acquisition, (ii) binary analysis of firmware, (iii) firmware disassembly, and (iv) derivation of the firmware update validation method. Step (iv) is further broken down into three approaches: (a) disassembly analysis, (b) black box analysis, and (c) hardware debugging analysis.

Firmware samples are obtained directly from the vendor website as firmware update packages. The firmware binaries are extracted from these packages. Following sample acquisition, binary analysis of the firmware files determine likely image formats and identify sections of interest related to validation (e.g., header information and candidate checksum fields). Firmware disassembly requires the determination of the target processor architecture, disassembly of binary to assembly code, identification of assembly functions, determination of the firmware base address, string analysis, and rebuilding function names in the disassembly.

For derivation of the firmware update validation method, disassembly analysis consists of searching for strings or recovered function names relevant to validation. Black box analysis uses characteristics of common validation algorithms to narrow the search space as well as brute force techniques to attack the firmware update validation method. Hardware debugging is also used to physically connect to the controller. This enables direct access to the execution path of the processor and device memory including the executive loader, which is typically inaccessible to the user. These techniques result in a preliminary candidate for the firmware update validation method, which is then confirmed, analyzed, and exploited to the extent possible.

In order to provide a more complete understanding of the prerequisites for successful firmware modification, the advantages and limitations of techniques used in the derivation process are discussed. The effectiveness of each technique is considered in relation to any time or cost requirements as well as the complexity of their implementation.

1.5 Impact

This research examines and determines the feasibility of counterfeit firmware attacks on a common PLC. Demonstrating this ability helps distinguish the true nature of the threat posed by firmware counterfeiting. The described reversing process identifies prerequisite capabilities of an attacker and limitations of any potential attack. This information provides the insight necessary to develop defensive and forensic analysis techniques for firmware modification attacks. In addition to the analysis enabled by this process, direct results enable the creation of realistic counterfeit firmware samples for analysis in future research. This aids in the development of effective strategies and tactics for preventing and detecting firmware modification attacks.

1.6 Organization

Chapter 2 discusses ICS security and reviews associated work in the area of embedded firmware. Chapter 3 details the approach taken by this research and presents the process applied to derive the firmware update validation method on the test controller. Chapter 4 applies the process to a PLC and analyzes the results. Finally, Chapter 5 concludes the research by discussing significance and relevant future work.

II. Background

2.1 Industrial Control Systems

An industrial control system (ICS) comprises a set of components used for the automated management and control of an industrial process. The term may refer to a multitude of different control system schemes, devices, and implementations, which include the control of production industry processes like automotive assembly plants as well as critical infrastructure systems including the electrical power grid, water treatment systems, and chemical industry. This thesis focuses on the protection of such critical infrastructure.

As an example, supervisory control and data acquisition (SCADA) systems represent one specific type of ICS. SCADA systems are typically used in the control and management of geographically dispersed industrial systems [55]. In the structure illustrated in Figure 2.1, the human user interacts with the control system through a human machine interface (HMI). The HMI functions as the operator's main method of monitoring and altering physical components to provide external oversight of correct operation. The HMI connects to a master terminal unit (MTU) that acts as the central automated supervisory unit of the SCADA system. The MTU monitors and manages the various physical sites composing the system, but it does not directly control end nodes. This responsibility is placed on field devices. Specifically, remote terminal units (RTUs) are field devices designed to control physical aspects of the system. SCADA systems branch out over various communication channels to assorted RTUs that control and monitor actual physical objects in the system such as valves and sensors. Another type of ICS, called a distributed control system (DCS), is a type of system that focuses more specifically on the control of localized processes. Unlike SCADA systems, DCSs are typically implemented in scenarios where the entire control system is contained within the same local network.

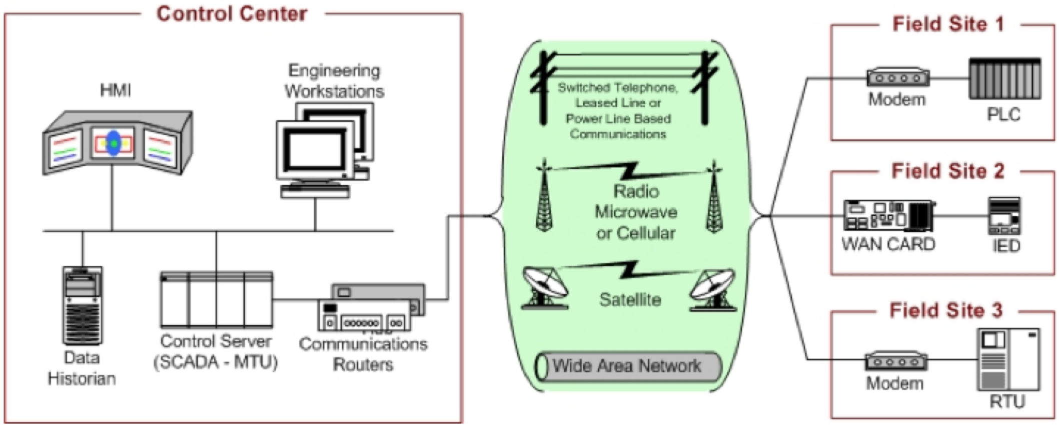


Figure 2.1: Typical structure of a SCADA system [55].

A commonly used device in ICSs is the PLC. PLCs are embedded computer systems specifically designed to control and, to an extent, independently monitor the physical system components under their supervision. PLCs are commonly used in DCSs and often times in place of RTUs in SCADA systems due to their enhanced capabilities. As the name implies, PLCs enable customized control of system components by providing a user-programmable interface between physical inputs and outputs. PLCs typically require proprietary software installed on a standard computer (usually running Microsoft Windows) to provide a method of programming the controller. Examples of such programming software include Rockwell Software's RSLogix series for managing Allen-Bradley controllers and Siemens's Simatic Step 7 for Simatic controllers. With these applications, engineers commonly utilize a graphical programming language called ladder logic to create a representation of how the controller should respond to given inputs. By creating virtual projects in the programming application, a user is able to view the logic currently running on a PLC as well as write new ladder logic to the device for execution.

2.2 Industrial Control System Security

The history of ICSs encompasses decades of steady advancement building to the complex systems present today. However, the evolution of these systems does not lend itself well to a satisfactory development of security practices and implementations. While such development continues to progress through the influence of regulation and research, it is often outpaced by advancing threats [13]. This section provides an overview of growing threats to ICSs and their significance on the future of ICS security.

2.2.1 History.

2.2.1.1 Past Incidents.

Cyber incidents related to ICSs are certainly not new; however, much has changed regarding the subject in a relatively short amount of time. A true history of ICS cyber incidents is difficult to compile considering low reporting rates due both to unrecognized incidents as well as the implications of recognized incidents on the reputation of those involved. Without any obligatory reporting procedures, only highly visible incidents are publicly reported.

The incident widely considered the first reported cyber attack against critical infrastructure took place in 1997 [1, 21]. In March, a teenager successfully hacked into a local service provider's loop carrier and disabled it. This resulted in the disruption of communications to and from the Worcester Regional Airport air traffic control tower. Also affected were communications for airport security, fire department, and weather service, as well as about 600 homes in the nearby area of Rutland, MA. The attack was executed using a dial-up modem connected to the disabled loop carrier. Three years later, in 2000, another attack took place in Queensland, Australia [54]. After three months of what was thought to be system glitches at the Maroochy Water Services plant, intentional malicious action was discovered. The actions were traced back to a disgruntled contractor who had failed to procure a new job with the Maroochy Shire Council. Over the course of the three months he had used a laptop and radio transmitter to create problems in the system and consequently released a total of around 1 million liters of sewage into local waterways.

In addition to these early attacks on critical infrastructure, the combined proliferation of computer worms and viruses over the past decades has led to unintentional effects resulting from the propagation of malware to ICSs. The first major reported incident of such a situation occurred in 2003 by means of the Sobig worm [42]. The Sobig worm managed to propagate to critical areas on CSX train systems, causing the shutdown of train signaling and dispatch services along the east coast of the United States. As a result, a cascade of delays was created, affecting many trains for hours. The same year was also marked by the Davis-Besse incident [45]. As a result of uncontrollable propagation of the Slammer worm, the safety monitoring system at the Davis-Besse Nuclear Power Station in Ohio was disabled. Fortunately, the plant was idled at the time, so no reactors were active. There also remained an analog backup safety system that remained unaffected; however, the fact that the worm could so easily impact the power station with no specific intent is concerning. Slammer was originally introduced into the unsecured network of a plant contractor. From there it was able to propagate across a T1 line that completely bypassed the plant firewall. In another 2005 case, the Zotob worm unintentionally attacked DaimlerChrysler [47]. By exploiting a buffer overflow in Microsoft's Plug and Play service, Zotob propagated onto DaimlerChrysler manufacturing plant networks and forced random system reboots [40, 48]. These disruptions shut down 13 manufacturing plants across 6 states for about an hour.

2.2.1.2 Stuxnet.

In 2010, the Stuxnet worm was discovered. This malware is now infamous as the epitome of advanced ICS threats. Reports indicate that Stuxnet was a highly targeted attack against specifically configured PLCs controlling particular ICS processes [12, 14]. The reports suspect that the initial infection vector was through the use of removable media devices. These may have been planted for unsuspecting employees to find and use, or the attackers may have breached physical security to deliver the worm [19]. Once introduced to a computer running Microsoft Windows, Stuxnet bypasses antivirus software and detects if Step 7 software is installed. If so, the worm replaces the main function library used by Step 7 with a malicious version. Stuxnet also places itself in any other removable media attached to the computer and any Step 7 project files for future propagation. When the malicious Step 7 library is loaded by a program, it runs a routine to search for specific PLC models that are connected to specific frequency converter drives used to control motor speeds. If such a PLC is found, Stuxnet injects malicious programming to the PLC that alters the motor speed, causing damage. In addition to the injection of malicious programming, the malicious Step 7 function library also masks the modified PLC code from the operator on an infected computer.

2.2.2 Future Threats.

In June of 2011, the National Institute of Standards and Technology (NIST) released the most recent version of their Guide to ICS Security [55]. The authors note that ICS technology is advancing by integrating with more traditional IT system solutions like IP-based communications and standard computers. As this occurs, ICSs become less isolated and more vulnerable to security threats. Traditional IT security solutions may be applicable to modern ICSs in some regards, but special consideration must be made in areas where traditional IT has no experience. Because of the cyber-physical link ICSs provide, issues such as human safety [21, 42, 45] and environmental [54] effects may be impacted by any gap in security.

A common trend witnessed in traditional computer malware is a race to the bottom where lower level attacks have the advantage over more overt and limited high level attacks. For example, user mode malware may be easily detectable as a file or running process and may be limited by security features present in the operating system. Alternatively, kernel mode malware is able to hide itself with rootkit functionality and may control kernel level processes inaccessible to a user mode program.

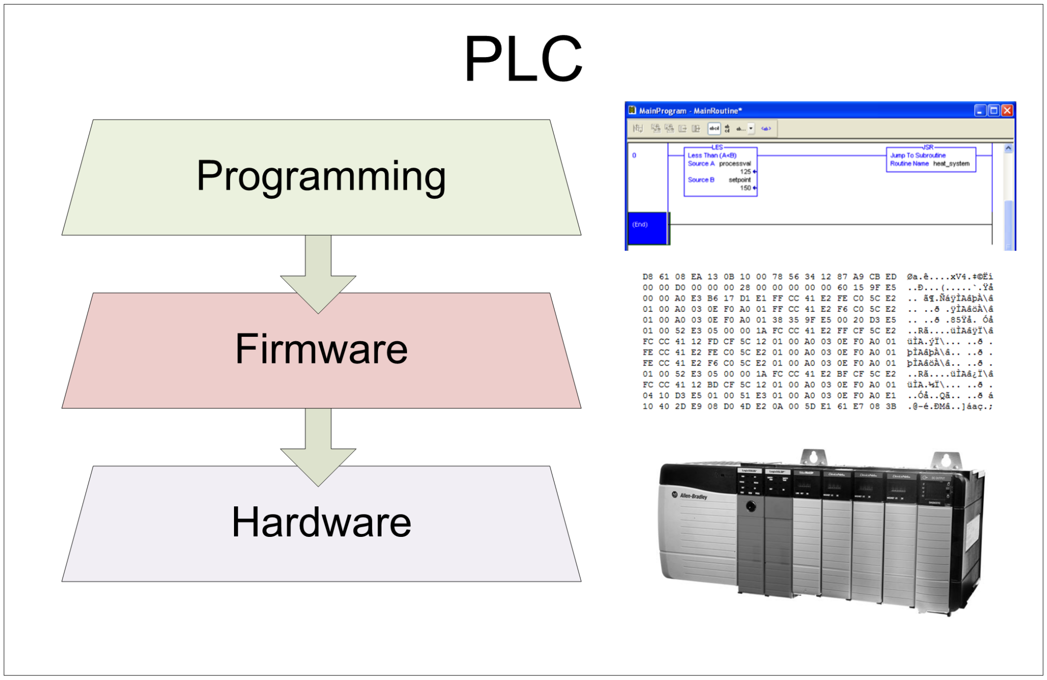
A similar trend is developing in the field of ICS security. Already, attacks like Stuxnet focus on the PLC given that it provides a link between the ICS and components affecting the physical world. At the highest level of PLC control, many current exploits focus on application layer vulnerabilities using hard-coded passwords to access control interfaces. Stuxnet takes this a step further by modifying the programming on PLCs and masking the change from operators using rootkit functionality on the Step 7 Windows machines. However, these modifications are not malicious to the PLC itself. The PLC is designed to be remotely programmable and the code loaded on the PLC by Stuxnet is valid operating code, so the device is only a conduit for the attack. Stuxnet takes advantage of the malicious Step 7 function library to hide modifications from infected Step 7 machines [19]. The PLC continues to run as it is instructed and the operator cannot observe modifications to the code; however, if the PLC is accessed from a non-infected Step 7 machine, the modifications are visible. The logical progression of attacks is to affect the PLC in such a way that no external observer can readily detect malicious modifications.

2.3 Programmable Logic Controller Security

Specific focus on cyber security from the perspective of PLCs remains a serious issue in ICSs. In a 2012 paper, McMinn *et al.* describe the existence of three operational layers on a PLC: (i) programming, (ii) firmware, and (iii) hardware [33]. This section discusses the major vectors available to attack PLCs and their related concerns following this model (see Figure 2.2).

2.3.1 Programming Layer.

The programming layer is the main channel of interaction between ICS operators and the PLC. Through this layer, a user provides the device with logic required to operate the



```
D8 61 08 EA 13 0B 10 00 78 56 34 12 87 A9 CB ED  Øa.ê....xV4.‡©Ëí
00 00 D0 00 00 00 28 00 00 00 00 00 60 15 9F E5  ..Ð...(.....`.Ÿå
00 00 A0 E3 B6 17 D1 E1 FF CC 41 E2 FE C0 5C E2  .. ã¶.ÑáÿÌAâþÀ\â
01 00 A0 03 0E F0 A0 01 FF CC 41 E2 F6 C0 5C E2  .. ..ð .ÿÌAâöÀ\â
01 00 A0 03 0E F0 A0 01 38 35 9F E5 00 20 D3 E5  .. ..ð .85Ÿå. Óå
01 00 52 E3 05 00 00 1A FC CC 41 E2 FF CF 5C E2  ..Rã....üÌAâÿÏ\â
FC CC 41 12 FD CF 5C 12 01 00 A0 03 0E F0 A0 01  üÌA.ýÏ\... ..ð .
FE CC 41 E2 FE C0 5C E2 01 00 A0 03 0E F0 A0 01  þÌAâþÀ\â.. ..ð .
FE CC 41 E2 F6 C0 5C E2 01 00 A0 03 0E F0 A0 01  þÌAâöÀ\â.. ..ð .
01 00 52 E3 05 00 00 1A FC CC 41 E2 BF CF 5C E2  ..Rã....üÌAâ¿Ï\â
FC CC 41 12 BD CF 5C 12 01 00 A0 03 0E F0 A0 01  üÌA.½Ï\... ..ð .
04 10 D3 E5 01 00 51 E3 01 00 A0 03 0E F0 A0 E1  ..Óå..Qã.. ..ð á
10 40 2D E9 08 D0 4D E2 0A 00 5D E1 61 E7 08 3B  .@-é.ÐMâ..]áaç.;
```

Figure 2.2: Operational layers of a programmable logic controller.

controller's given application. Many different languages are used at the programming layer including modified implementations of traditional languages such as C or BASIC [27]; however, the typical method used to program controllers is a graphical language called ladder logic. Ladder logic provides engineers, who may be unfamiliar with traditional programming languages, an intuitive interface to the controller. Programming software used to upload the logic to the PLC (e.g., RSLogix) compiles the graphical language to low level code for execution before uploading the code to the device. The programming uploaded to the device is analogous to a desktop application run on a traditional computer. For the PLC, the programming dictates how the controller responds to input. Because the program loaded on the controller is managed by programming software, modifications solely made to the program can be readily detected, as mentioned in the Stuxnet discussion.

2.3.2 Hardware Layer.

The hardware layer consists of the physical chips and components that make up the PLC. Among other components, this layer includes the microprocessor, volatile run-time memory, and non-volatile storage memory. Security at this layer is based on physical protection. From a malicious perspective, three main vectors are available to attack the hardware layer: physical manipulation of the hardware, software exploitation of hardware design flaws, and supply chain compromise to intentionally create vulnerabilities. Of these, physical manipulation is the least likely in an operational scenario as this would require the attacker to have intimate access to the device, possibly for extended periods of time. Such an attack implies an insider threat scenario, in which case more straightforward methods of attack like direct malicious reprogramming are possible. The second vector of attack at the hardware layer reduces to a software-based attack where physical design flaws are exploited by software running on the device. This requires exploitation through either the programming layer or the firmware as discussed in the following section.

The remaining attack vector is supply chain compromise. If an attacker can compromise the supply chain of components for a device, they may be able to influence their design and engineer vulnerabilities, providing backdoors to the system. Detection of supply chain compromise is a difficult and costly endeavor. The analysis and reverse engineering of a physical component in an attempt to detect malicious logic requires a significant amount of time. Modern microprocessors have become so complex that such methods are not feasible in many cases. In those cases where it is, the device is likely destroyed in the process. For these reasons, researchers are investigating viable alternatives using side channel methods [57]. By analyzing changes in signal metrics such as timing and power, comparisons are made to known-good reference circuits to determine if any differences are present in the test device. This method assumes that the modification of a circuit results in a noticeable change in the power requirement or timing of the circuit. However, many challenges remain in this area of research. Not only is a trusted baseline required for comparison, but the possible implementations of malicious logic are so numerous and complex that choosing appropriate detection metrics is difficult.

Presenting a comprehensive approach, Baldwin *et al.* explore the issue of supply chain trust from a strategic perspective and propose new policies to integrate "system security engineering" into the current Department of Defense acquisition life cycle [7]. This proposal follows a cradle-to-grave approach to security. Pre-development planning focuses on security from the start by analyzing possible vulnerabilities of system components. Supply chain risk management is discussed and encompasses various techniques to minimize the threat of supply chain compromise. For critical components, a full scope supply chain analysis is performed to identify all suppliers involved with component production. The concept of trusted suppliers is introduced where strict requirements are imposed for such vendors. The purchasing of components should also follow diverse redundancy and anonymous buyer practices to minimize the probability of compromise. System testing focuses specifically on critical components as they interact with the rest of the system as well as many other non-hardware based secure design practices throughout the system life cycle.

While a serious issue warranting concern, hardware layer security is ultimately immaterial in the context of current production systems. At some point, assumptions must be made about the security of the hardware. Indeed, these assumptions are already made by vendors and consumers by the fact that the devices are in use. This fact reinforces the concept that while strict physical security, thorough quality control, and a secure supply chain are important, these are complex issues requiring more specialized and long-term solutions unconcerning many field devices in current production or operation.

2.3.3 Firmware Layer.

Bridging the low level hardware layer to the upper level programming layer is the firmware layer. Firmware is the low level software run on the device to support higher level operations. For this reason, firmware is commonly referred to as the operating system (OS) of embedded devices. In a broader sense, however, firmware also includes lower level functionality like bootloader code to initialize and load the OS.

This layer in a PLC that controls the basic behavior of the device including communication with management systems and execution of the user-level program written to the device. Firmware handles all interactions between the user and the device hardware, including physical inputs and outputs. The functional analogy of OS firmware is further extended when discussing potential threats to the PLC. The traditional operation of a rootkit on a standard computer is equivalent in concept to how an attacker may take advantage of OS firmware in a controller to hide modifications. Rootkits typically exploit kernel-level processes to gain privileged access to OS functionality. Using this access, rootkits are able to modify the underlying behavior of the OS. In the same way, an attacker with access to the firmware on a PLC has potentially limitless control over the device including the ability to covertly alter device behavior in a malicious fashion.

In some embedded devices, firmware is programmed from the factory and designed such that it remains static and not reprogrammable. In such cases, firmware is relatively safe from modification attacks. However, modern embedded devices including PLCs are commonly designed with the capability to update the firmware. This ability allow vendors to patch bugs present in the firmware as well as enable new features without requiring physical updates of the hardware. Given a device with reprogrammable firmware, the procedure of performing the update is usually the responsibility of the user, typically through the application of an update software package. Since a user has access to update the firmware, however, the process facilitates an attacker's ability to upload modified firmware to a device.

2.4 Previous Works on Reversing Firmware

2.4.1 Discovering Backdoors in Ethernet Modules.

In a December 2011 web article, independent security researcher Ruben Santamarta describes a process used to discover backdoor accounts and passwords for a Modicon Quantum PLC Ethernet module [51]. While the goal of his research was not to modify the firmware, but to uncover these backdoors, the process he follows is applicable to search for validation algorithms in firmware.

To begin the reversing process, Santamarta obtains a copy of firmware for the target controller. He does so by downloading the firmware from the vendor website. An entire update package is obtained, where the actual firmware binary is contained within. Therefore, the binary image itself is found and extracted from the rest of the package. An inspection of the the update package contents reveals the location of the firmware image to extract. Santamarta advises that other files in the update package may contain useful information about the firmware and continues examining the firmware itself.

The first step is a manual inspection of the binary image using a binary file editor. This procedure reveals a file header followed by a zlib-compressed section, identified by its leading binary signature. After decompressing this section, he proceeds to identify the processor type, which is a PowerPC. While he does not go into detail about his method of processor type determination, Santamarta references a presentation by Igor Skochinsky [53] detailed in Section 2.4.3.

Following this determination, Santamarta begins the process of disassembly by loading the binary file into the Hex-Rays Interactive Disassembler (IDA), a disassembler tool. At this point, he presents a common process for reconstructing the firmware code in IDA. The process begins by collecting information from strings contained in the firmware image. A copyright string identifies the firmware as VxWorks-based OS. Next, since IDA is not able to automatically detect functions in this binary image, Santamarta describes a method to resolve functions in the disassembly by searching for common function prologue bytes. He then commands IDA to treat these addresses as functions by disassembling their code and adding them to an internal database. Santamarta provides the source code of the IDA script he uses to perform this task.

Now that IDA has disassembled and identified functions in the image using the standard prologue, Santamarta "rebases" the image by altering the code base address to that which it assumes at runtime. He suggests that the true base address may be located in the firmware header or other documentation, but determines this is not so for his case. Instead, he uses the "'[load immediate] instructions' trick" [58]. This method consists of searching the disassembly for instructions that load an immediate (i.e., absolute) address value into a register. The technique presumes that a significant number of immediate addresses refer to locations in the firmware itself, and therefore have the same base address. Candidate base addresses are then tested by rebasing the firmware and determining if the immediate addresses correctly align with target data such as strings or other functions.

After successfully rebasing the firmware, Santamarta rebuilds the function symbols, or names. He begins by searching the firmware for a symbol table. This is identified as a section of the code containing a regularly repeating data structure that includes function name strings. He discovers such a table and, using another IDA script, parses the symbol table and relabels the disassembled functions with their proper names. This enables Santamarta to use symbol names in finding sections of the firmware relevant to his goal of discovering backdoor accounts. Indeed, he finds many undocumented accounts for the Ethernet module.

2.4.2 Creating Custom Firmware for Ethernet Modules.

In a 2009 paper, Peck and Peterson demonstrate a successful upload of customized firmware to an Allen-Bradley ControlLogix Ethernet module [44]. Their goal is to show that an attacker can learn how to upload custom firmware to a field device Ethernet card using commonly available tools. They justify their choice of Ethernet card targets by describing the vulnerable state of Ethernet modules and their ease of access through remote network means. Peck and Peterson proceed to explain their method of reversing the Ethernet module firmware and subverting the card's validation algorithm.

Peck and Peterson first acquire multiple firmware samples for the target Ethernet module. These samples are downloaded from the manufacturer website. Peck and Peterson explain the importance of obtaining multiple samples for comparison as a way to identify static fields in the images. With firmware samples available, Peck and Peterson begin inspecting the binary files, looking for and identifying different segments that exist in the image (e.g., blocks of code, filesystems, or strings).

After the manual inspection, Peck and Peterson utilize a binary analysis tool called Deezee, which searches binary files for embedded zlib-compressed sections, extracts, and decompresses them [38]. Using this tool, Peck and Peterson discover a zlib-compressed section containing the symbol table for the firmware, identified by a regularly repeating pattern of addresses and string symbol names. Further analysis of the symbol table reveals function addresses listed as absolute values. Using these addresses they infer the base address of the image.

Peck and Peterson next attempt to disassemble the firmware code using IDA. However, a known processor type is required, so they initially assume that the target uses an ARM core. They are incorrect, but infer from the result of the attempted disassembly that the true processor type is PowerPC. Another attempt with this target processor type is successful and produces disassembled code. A rebase of the image is then performed using the base address derived from the symbol table. Finally, Peck and Peterson use a script to add the symbol names to the disassembly.

Peck and Peterson continue to search for the firmware validation algorithm used by the Ethernet module. By searching through the symbol names, they discover a function named nv_RamValidateChecksumsWriteFlash. An examination of this function reveals a subfunction call that performs a checksum validation calculation. Peck and Peterson reverse engineer this function to derive the entirety of the checksum validation algorithm used by the firmware. Their target device uses a 2-byte summation of the image header and a 2-byte summation of the remaining firmware image, both of which are located in the header.

With knowledge of the validation algorithm, as well as the location of the checksum values, Peck and Peterson customize the firmware with a proof-of-concept that instructs the Ethernet card to continually ping a specific IP address in addition to its normal operation. The authors note the importance of taking care not to disrupt default operation of the device. Finally, the customized firmware is uploaded to the Ethernet module using the ControlFlash firmware programming software Rockwell Software provides. While they admit that a custom flash program could be written to accomplish the task, Peck and Peterson argue that it is simpler to use the vendor supplied software.

2.4.3 General Processes for Reverse Engineering Embedded Devices.

In a presentation at the 2010 Recon security conference, Igor Skochinsky, a software developer on the Hex-Rays IDA team, provides an introduction on reverse engineering embedded firmware [53]. In Skochinsky's process, he first retrieves a firmware image. He provides various methods for doing so, the first of which is obtaining a firmware update from the vendor. Skochinsky notes that this method for acquiring firmware is straightforward and that the updates obtained may also contain firmware upload programs, filesystems, or bootloader images in addition to the main firmware, which may be relevant.

One may also take advantage of external communications with the device. While taking more time and effort, a communication program on a separate computer from the device could be reverse engineered to reveal methods of instructing the device to transmit sections of memory containing firmware code. Another technique is the use of a universal asynchronous receiver/transmitter (UART) port on the device. This method is complex, possibly involving the identification of and connection to physical UART pins on the device if no standard serial port is present. After physical connection, it may be possible to dump the firmware from the device using certain commands sent over UART, depending on the device's support for UART. Similarly, Skochinsky also discusses the use of hardware debugging tools to dump the contents of memory, as discussed in Section 2.4.4. Finally, Skochinsky mentions the tactic of reading flash memory directly from the storage device; however, this typically requires the physical removal of an embedded flash chip from the board by desoldering.

With possession of the firmware, Skochinsky next characterizes the image. This begins by identifying any filesystems embedded in the firmware. He discusses various common filesystems utilized by embedded systems, how to identify them (typically using binary signatures), and how to unpack them into a usable form and access the files within. Following this, Skochinsky's discusses how to identify the embedded operating system type. Again, he walks through various embedded operating systems and how to identify them. Identification usually involves locating copyright strings containing the developer or operating system name. The final characterization before code disassembly is identification of the processor type. Skochinsky begins by explaining the differences between major design types (e.g., reduced instruction set computing (RISC) versus complex instruction set computing (CISC)), then discusses the general attributes of several popular embedded architectures and how to identify them by signature byte patterns common to that architecture's instruction coding.

Skochinsky's final step is disassembly of the firmware code. If the firmware is contained in a structured image such as an executable and linkable format (ELF) or other OS-specific format, a disassembler like IDA may be able to automatically disassemble the code, determining much of the code structure from the wrapper. However, if the firmware format is raw binary, additional work is required to create the equivalent disassembly. The first step of this process is determining the correct base address for the binary. Skochinsky suggests initializing the base address to 0, then searching the code for hints of the true base address if the use of 0 fails to produce complete disassembly. Such hints include self-relocating algorithms that copy code using the correct base address, initialization code that uses the base address to load code from non-volatile flash memory to volatile random-access memory (RAM), jump tables that contain absolute addresses, or string table offsets to compare with the addresses of strings they point to. After successfully rebasing the code, Skochinsky discusses recovering symbol information. He explains the extraction of symbols from a Linux kernel and provides an example of the VxWorks symbol structure. Skochinsky also suggests searching for a demonstration or evaluation copy of the OS type, if it is known, to allow for comparisons between it and unknown code in the disassembly.

2.4.4 Hardware Debugging.

Hardware debugging is commonly used in the production of embedded systems as a method to test and verify system components as well as debug software at the processor level. One common hardware debugging standard is the the Institute of Electrical and Electronics Engineers (IEEE) Standard 1149.1 Standard Test Access Port and Boundary Scan Architecture [25]. This standard is often referred to as Joint Test Action Group (JTAG), after the name of the consortium that created the standard [43]. Boundary scanning with JTAG requires a specialized hardware debugger connected to the test board through special pins called test access ports (TAPs). The standard specifies certain pin signals required to control the device under test. Four signal pins and an optional fifth are the minimum defined by the standard: clock synchronization (TCK), mode select (TMS), data input (TDI), data output (TDO), and the optional reset signal (TRST). JTAG operates on a per-chip basis, so an integrated circuit (IC) on the target board must be designed to support JTAG. Typically, these chips are microprocessors. Given multiple components on a board that support JTAG boundary scanning, the same TAPs can be used to access all of them. This is known as chaining.

In a 2006 paper, Breeuwsma provides an introduction on JTAG and how to take advantage of it for forensic imaging of embedded applications [10]. Breeuwsma describes three main modes of operation for a JTAG-enabled device: normal operation mode, external test mode, and debug mode. Normal mode bypasses boundary-scan functionality to allow normal operation of the chip. External test mode provides basic JTAG functionality. In this mode, the processor core of the target is disabled. Instead, the input/output (IO) pins of the target chip are driven by values stored in the boundary-scan test register called test vectors. Using test vectors, the hardware debugger can completely control the IO signals on the target. This is useful for hardware validation and low level debugging operations including direct memory accesses.

Debug mode is an advanced operating mode allowed by JTAG to facilitate software testing and debugging. This mode requires special circuitry built into the target chip not specifically defined by the standard. Therefore, it is not uncommon for the implementation of debug mode to vary among different chips. In some cases, JTAG enabled components may lack debug circuitry altogether and only support external test mode. However, if the chip is designed with support for this mode, software executing on the chip may be debugged in real time. Depending on the target's implementation, debug mode may also enable access to memory on the target system without requiring specifically crafted test vectors as with external test mode.

Breeuwsma continues by describing techniques to identify JTAG TAPs on a device. The general process beings by searching for test pads on the target circuit board and eliminating as many as possible from the list of TAP candidates. Breeuwsma provides a list of characteristics common to JTAG TAPs that bound the search space. Such characteristics include the fact that TAP signals should remain constant while the system is running with no debugger attached. TAP traces should also not connect to non-IC components like capacitors (however, pull-up or pull-down resistors may be present). Lastly, TAP signals should not be driven by an output. After applying these rules for elimination, Breeuwsma provides additional detail on TAP characteristics to confirm those remaining candidates as TAPs.

2.4.5 Checksum Algorithms.

As discussed in Section 2.3.3, firmware on embedded devices such as PLCs is often updated by manufacturers. Firmware update processes typically include the use of validation methods to confirm that the newly uploaded image is not corrupt [44]. This section provides an overview of various types of algorithms commonly used for data validation with a specific focus on the validation of firmware updates.

A hash function is defined as a function that maps data of an arbitrary length to a fixed-length value called the hash value [34]. A checksum algorithm, then, is any type of hash function used for the purposes of validating data integrity [59]. The resulting hash value generated by a checksum algorithm is referred to as the checksum value or, succinctly, the checksum. There exist various algorithms used for calculating checksums that fall into five general categories: parity checks, modular summations, cyclic redundancy check (CRC), non-cryptographic hashes, and cryptographic hashes. Parity check algorithms, typically synonymous with longitudinal redundancy checks (LRCs) [39], calculate a checksum value by applying the *exclusive or* (XOR) operation to each n-bit word over the data set. This produces an n-bit checksum value representing the “parity” of each bit position in every word [28]. Parity algorithms have the advantage of being computationally inexpensive and straightforward implementations; however, they are less accurate in detecting data errors then other algorithm types mentioned here. Typically, the only variation between parity check implementations is the bit width of each word in the calculation.

Modular summation algorithms are based on the addition of each *n*-bit word in a data set with the next. The resulting sum is represented using a given modulus to create a checksum value of the desired bit width. Note that the term “checksum” is sometimes used by other works in reference to modular summation algorithms specifically; hence, there is common cause for confusion. This thesis, however, uses the term “checksum” in reference to the resulting value of any hash function used for the purposes of validating data integrity. Another specific type of modular summation algorithm is the Fletcher algorithm [20]. This algorithm calculates a modular summation as before, but also includes a second modular sum of those simple sums, creating two values that compose the checksum. Fletcher algorithms can be calculated with various moduli to achieve bit widths of 16, 32 or 64 bits. Each of these widths has a commonly used modulus associated with it. A further specific variation of the Fletcher algorithm is the Adler-32 algorithm which uses the specific modulus of 65,521 for both sums [17]. Modular summation algorithms allow for more accurate error detection than parity checks while remaining straightforward to implement. However, the variability of different modular summation methods is much higher than parity algorithms. Since several different schemes are based on the summation concept, determination of a specific algorithm is more difficult than for parity checks.

A CRC is a type of algorithm that uses polynomial division on a data set to produce a checksum value representing the remainder of this division. The specific process used in the CRC calculation is detailed by Ramabadran and Gaitonde [46]. In general, implementation include a series of incremental XOR operations over the data, where the result of each increment is dependent on the polynomial, the data, and the preceding result. A CRC is based on variable parameters including width of the checksum value, the static polynomial value, initial checksum value, the final XOR value applied to the resulting checksum, and whether or not input and output bytes are reflected for endianness. Therefore, the specifics of CRC algorithm implementations are highly variable. CRCs have the benefit of more accurate error detection rates than modular summations [32]. While more computationally complex than the above algorithms, CRCs remain less complex than the remaining categories.

Dedicated hash functions are categorized as either non-cryptographic or cryptographic. While, by definition, all the algorithms discussed here are hash functions, these two categories refer to functions specifically designed to minimize collisions. A collision occurs when a hash function is applied to two unique sets of data and the resulting calculations produce identical hash values. Since the previous categories of algorithms are not designed specifically to prevent this outcome, they remain vulnerable to collisions. Thus, dedicated hash functions are better suited to detect changes in input data, but usually at the cost of computational speed and complexity.

The difference then between non-cryptographic and cryptographic hash algorithms is that in addition to their sensitivity to accidental modifications of data, like previous functions, cryptographic hash algorithms are also capable of detecting intentional modifications. Specifically, cryptographic hash functions are not feasibly vulnerable to collisions or reversal. To a lesser extent than previous categories, one is still feasibly able to find collisions in non-cryptographic hash functions or reverse them to determine a data set that produces a given hash value. Cryptographic hash functions, however, are not considered vulnerable to such attacks in any feasible manner. For example, it may be technically possible to find a collision in a cryptographic hash function through brute force, but the computational resources required to do so in a reasonable amount of time are unrealistic. Many cryptographic hash functions exist, most notably the secure hash algorithm (SHA) family of algorithms [41].

Given that embedded devices are typically limited in available memory and computational ability, less complex checksum algorithms are more feasible than cryptographic hash functions for such applications [32]. However, there exists a trade off between the simplicity of an algorithm and its ability to accurately detect changes in data. For these reasons, modular summation and CRC algorithms are commonly used as validation methods for embedded systems. Their balance of computationally inexpensive calculation and reasonably accurate detection rates of unintentional errors make them popular in applications such as Ethernet, transmission control protocol (TCP), and the VxWorks embedded OS [26, 44]. As discussed by Maxino and Koopman, however, some embedded applications may also take advantage of completely proprietary checksum algorithms, especially in the case of embedded control networks [32].

2.5 Summary

Much of the nation's critical infrastructure relies on ICSs to monitor and automate control processes. As these systems evolve from traditionally isolated and specialized implementations to adopt common IT solutions, they become exposed to cyber attacks. This is witnessed as ICS cyber incidents progress from isolated attacks to unintentional impacts caused by computer malware, culminating in the highly targeted Stuxnet attack on control systems. As these attacks advance, focus is shifted from targeting high-level application systems to direct threats against PLCs. PLCs consist of a user-level programming layer, a low-level firmware layer, and a physical hardware layer. The ability of firmware to completely dictate behavior of the device with no direct operator oversight presents an attack vector accessible by legitimate means and capable of masking malicious activity, unlike the observable programming layer and inaccessible hardware layer. Previous works discuss various techniques for reverse engineering embedded devices with a specific focus on PLCs and firmware modification. Other works on the topics of hardware debugging and checksum algorithms as they relate to embedded devices augment such techniques to provide a comprehensive survey of the field.

III. Methodology

This chapter outlines the methodology by providing a problem definition and general approach, briefly describing the procedure and purpose of each step in the conceived process, and finally detailing the factors considered in assessing advantages and limitations of the composed techniques.

3.1 Problem Definition

Strategically, the intended result of this research is to further defensive measures and forensic analysis techniques targeted toward advanced ICS threats by determining the feasibility of firmware modification attacks on PLCs. Specifically, the goal of this effort is to investigate and assess the vulnerability of a common PLC to counterfeit firmware updates. The achievement of this goal provides valuable information and insight regarding the feasibility, technical requirements, and characteristic implementation of future firmware related threats to PLCs.

This research proposes that common PLCs are vulnerable to firmware modification attacks as a result of design weaknesses in firmware update validation methods. A design weakness is defined as a failure of the device to properly detect intentional modifications to firmware. An appropriately counterfeited firmware is expected to be accepted through the standard firmware update process of the controller and execute on the device in no distinguishably different manner than any given legitimate firmware, save for any modifications present in the counterfeit version.

3.2 Approach and Scope

The process to derive the firmware update validation method is based on a thorough review of previous research involving the reverse engineering of similar embedded devices (see Section 2.4). Relevant tactics and procedures are compiled into a general process to identify the firmware update validation method. The approach is applied to a production PLC that is both commercially available and commonly used in ICS implementations. Through this application, the constituent techniques of the reversing process are assessed for advantages and limitations in their ability to derive the firmware update validation method. Furthermore, feasibility of the threat posed by firmware modification attacks on PLCs is determined by a successful upload of counterfeit firmware. The reversing process provides a basis for the strategic intent of furthering defensive and analytic research on the topic.

The scope of this research is limited to PLC firmware counterfeiting. In a layered operational model, each layer is functionally independent of every layer above it and functionally dependent on every layer below it. Given the layered operational model of a PLC, the firmware layer is independent of the programming layer or any higher layer control mechanisms because the firmware dictates how those higher layer actions are interpreted. For this reason, no layers higher than the firmware layer are considered in the scope of this research. Similarly, the firmware layer is dependent on every layer below it to consistently interpret its actions. Therefore, as the only layer below the firmware, the hardware layer is considered in the scope of this research. External to the operational model, the only other considered system component is the firmware update procedure. Specifically, this research is based on legitimate firmware update procedures only. This includes any interaction with the device that the firmware interprets as a legitimate firmware update request, including the manufacturer supplied firmware update procedure or any procedure sufficiently similar to initiate a firmware update. No other external mechanisms or components are considered in the scope of this approach including, but not limited to, auxiliary controller components or the interaction of firmware updates with any external system.

3.3 Test Environment and Tools

The testing environment for this research includes an Allen-Bradley ControlLogix 1756-L61, Series B, Standard Controller module manufactured by Rockwell Automation. This hardware is accompanied by standard firmware available from the manufacturer which is applied to the device using Rockwell's standard ControlFlash update software. Since the scope of this evaluation is bounded to the PLC hardware and firmware layers with standard update procedures, this specific test environment is equivalent to the intended manufacturer configuration. The L61 controller module is accompanied in this test environment by a standard ControlLogix 1756-PA72/C power supply, 1756-A7 chassis/backplane, and 1756-ENBT Ethernet communications module to support operation and testing. However, none of these additional components are considered by the research presented here. Thus, results of this evaluation are accurate for any standard L61 controller within the given scope.

Additional tools used throughout the evaluation process include the Notepad++ standard text editor and the HxD binary file editor. The binary analysis step of the process takes advantage of the binary file difference tool Visual Binary Diff (VBinDiff) to perform a binary file comparison as well as the static binary analysis tool BinWalk for embedded file and filesystem analysis. Firmware disassembly takes advantage of the IDA tool extensively for further analysis. Brute forcing techniques discussed as part of black box testing use the CRC RevEng tool by Gregory Cook. Finally, the explored hardware debugging techniques include the use of the Advanced RISC Machine (ARM) RealView in-circuit emulator (ICE) device along with the ARM Development Studio 5 (DS-5) debugging software.

3.4 Reversing Process

An integration of related works discussed in Section 2.4 results in the general reversing process illustrated by Figure 3.1. This section details the steps in the process: (i) firmware sample acquisition, (ii) binary analysis of firmware, (iii) firmware disassembly, and (iv) derivation of the firmware update validation method. While the presented process is based on several common methods previously discussed, the nature of reverse engineering is at times as much an art as it is a science. The process externally requires intuition and experience on behalf of the investigator. The success and effectiveness of each step may rely in part on this variable aspect of the process. Nonetheless, the process serves as a roadmap to follow in deriving operation of the system.

3.4.1 Firmware Acquisition.

In order to begin reverse engineering the firmware, sample copies are first obtained. The primary method for accomplishing this is by procuring firmware updates from the vendor, which are typically available from online sources. The firmware binary images are then extracted from these update packages. Should this not be possible, alternative measures are taken such as memory acquisition through JTAG or desoldering the flash chip to directly read its contents. As many different firmware samples are obtained as possible to enable a thorough binary file analysis and comparison.

3.4.2 Binary Analysis.

This initial interaction with the firmware involves an examination of the raw binary files intent on gaining general knowledge required for a detailed inspection. Three techniques are applied: manual inspection, binary file comparison, and embedded file/filesystem analysis. A manual inspection reveals information about the general structure and contents of the binary file. In addition to static fields discussed by Peck and Peterson [44], dynamic fields of the firmware are also identified. Both are determined through the binary comparison of different firmware samples with the goal of identifying header contents and organization as well as fields in the image used for validation purposes. Embedded file and filesystem analysis is intended to detect the existence of any embedded files or filesystems in the firmware, which may contain information relevant to the firmware code operation or organization.

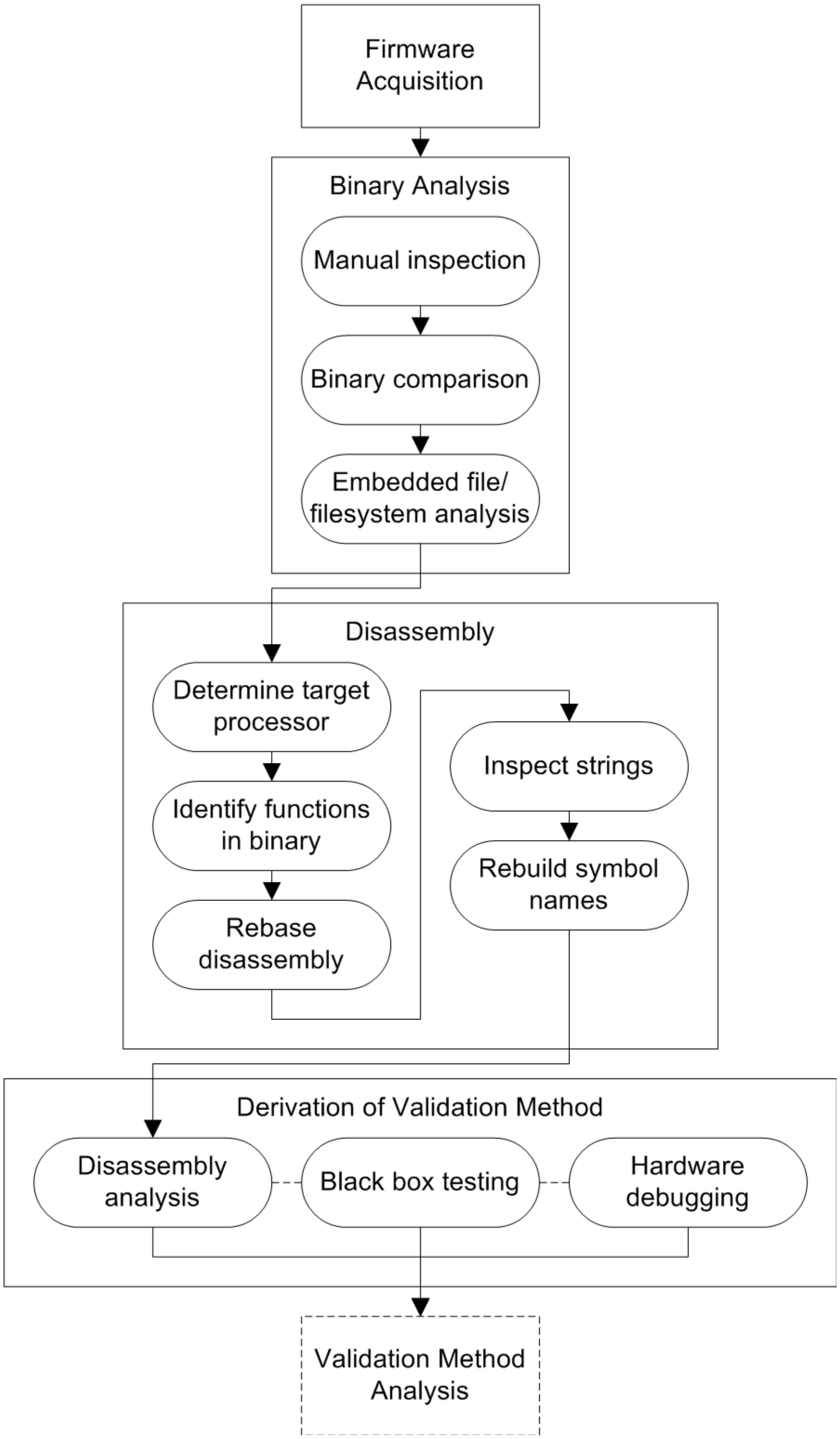


Figure 3.1: Reversing Process.

3.4.3 Disassembly.

The next step in the reversing process is disassembly of the binary code. To begin, the target processor of the binary code is identified. This can be accomplished using techniques such as Skochinsky's method of pattern matching [53]. Alternatively, a physical inspection of the target device hardware, if available, may also reveal details of the processor, as may a thorough review of official documentation for the system. Once the target processor is determined, the firmware binary is loaded into the IDA tool for disassembly. Depending on the format of the binary image, IDA may disassemble the code more accurately and completely in some cases rather than others. Assuming the firmware format is raw binary, the initial base addresses is set to 0. Following this, the disassembly function locations are fixed before proceeding further. Using Santamarta's IDA script as a template [51] and knowledge of the target processor's typical function prologue, IDA disassembles and adds the firmware function locations to its database, making code traversal and cross references in IDA more thorough and accurate. Note, however, this does not mean the functions are correctly named yet, only that they are identified as functions. The disassembled code is rebased if the initial loaded base address of 0 is incorrect. The correct base address may be discovered by applying methods described in the Section 2.4. Once the correct base address is used to disassemble the firmware, IDA matches cross references to more accurately and thoroughly traverse the code. Next, a thorough inspection of strings contained in the firmware is performed in order to identify any specifics of the firmware including OS type and any symbol names or data structure strings. The information gleaned from a string analysis is useful in learning more about how the code functions and leads into the next step: recovering symbol information. Rebuilding the firmware symbols may or may not be completely possible depending on results from the previous step, but strategies such as scripted renaming of functions based on apparent string names as discussed in Section 2.4, if successful, result in disassembled code that is more human readable to assist in analysis.

3.4.4 Derivation of Firmware Update Validation Method.

The goal of modifying the firmware relies on the ability to reverse engineer the firmware update validation method used by the PLC. The general approach consists of three phases: disassembly analysis, black box analysis, and hardware debugging methods. Disassembly analysis begins by searching symbol names for relevant titles indicative of the validation algorithm, such as those including the terms "checksum" or "CRC." Afterwards, the disassembly of such functions is analyzed to determine the validation method. Without symbol names, this process is hindered. Black box analysis uses several techniques to infer as much information as possible about the validation algorithm by narrowing down the search space, testing for the use of common algorithms, and attempting a brute force attack. Beyond this, the use of hardware debugging tools is also applied. If supported by the target device, this technique potentially allows for direct access to the device including live memory and control over execution on the processor to observe the validation as it occurs.

3.4.5 Reversing Process Considerations.

Advantages and limitations of the techniques used throughout the reversing processes are considered. A discussion is provided regarding factors including the effectiveness of each technique in contributing to the successful derivation of the validation method as well as time, cost, and complexity requirements for doing so. In the firmware acquisition step, specific considerations include the availability and ease of access to multiple firmware image samples from the manufacturer through firmware update packages or direct acquisition from the device. For the binary analysis step, each technique is individually considered. The discussion of manual analysis considers the technique's ability to extract file header and structure information. Binary file comparison is intended to identify static and dynamic sections of the firmware images and is assessed on the extent to which this is possible as well as the significance of information gained by doing so.

Discussion of the embedded file and filesystem analysis considers the success in identifying embedded files or containers as well as the accuracy and significance of any findings.

In the disassembly step, the discussion considers if the correct processor type is identified and the difficulty in doing so. Function identification is assessed by the extent to which firmware functions are successfully identified in the code. The accuracy of the determined base address is next considered, as is the rebasing technique's effectiveness in finding it. The string inspection discussion considers the number and quality of informative string values found in the firmware. To conclude this step, the accuracy of firmware functions renaming is considered.

Derivation of the firmware update validation method is divided into each component technique. The disassembly analysis is assessed on the ability to identify the validation method as implemented in the firmware and the effectiveness of being able to find relevant functions based on the details available. For black box analysis, considerations include the extent to which a likely class of the validation method is identified as well as the range of the firmware image covered by the method. Additionally, the brute force technique applied is assessed on its ability to determine the validation method and the feasibility in using this technique. Finally, the hardware debugging technique discussion considers the discovery of the validation method and the costs and effort required to do so.

3.5 Vulnerability Assessment

3.5.1 Firmware Update Validation Method Analysis.

The given candidate for the firmware update validation method is verified for correctness by applying the reversed method to all available firmware sample images. If all sample images pass the implemented candidate validation, then verification is successful. If verification fails, an iterative approach is taken to revisit the derivation process and determine the cause of the inconsistency before reattempting verification.

Following the verification of correctness, the reversed firmware update validation method is analyzed for potential design weaknesses. The existence of such a design weakness, as previously defined, means an arbitrary modification to a legitimate firmware sample image does not hinder the ability of that image to pass validation. To determine a design weakness, the derived firmware update validation method is analyzed for any functionality intent on obstructing alteration of the firmware in such a way that prevents its successful validation. For example, if the firmware is validated with a checksum and there is no functionality to prevent the successful recalculation of the checksum value given an arbitrary modification, then a design weakness is determined to exist. If such a weakness is successfully determined, a solution to take advantage of this weakness is implemented. This implemented solution, when given an arbitrarily modified firmware image, processes the image to produce a version of the modified image specifically capable of passing validation on the device.

Assuming the presence of a design weakness in the validation method and an implemented solution to take advantage of it, the solution is now tested. This testing involves determining an innocuous location in the firmware where alteration has no apparent affect on its operation. One example of such a location is an output string not used in comparison operations. After making a minor modification in this location, the implemented solution is applied to the modified image, producing a firmware version to test for general validity. This modified firmware version is next applied to the same test environment described above using the standard firmware update procedure for the device. While it cannot be guaranteed that the device will execute all possible states of the modified firmware logic, the test is considered successful if the device accepts the modified firmware as valid and continues to execute the modified firmware version with no apparent adverse effects. While it is straightforward to determine whether the device faults or not, its use of the modified firmware version may not be transparent. The latter is verified to the greatest extent possible by assessing the operational functionality of the device with respect to details unique to the uploaded firmware. If the modification can not be verified, more testing is performed by modifying alternate, more readily identifiable locations in the firmware. If the device fails to accept the testing modification, an iterative process is performed to determine the reason for the failure and, if possible, return to the testing phase when another solution is met.

While working with embedded devices and committing modifications to such low-level operational code, there exists a possibility that the device may become unstable and act unpredictably. In the event that a modified version of firmware is uploaded to the device that provokes such unpredictable behavior, the device may become locked into an unusable state. This is referred to as "bricking" the device. If a device becomes bricked, there may be no easy way to recover it. For example, the code on the device that controls firmware uploads may never be reachable in a bricked state, making it impossible to update to a known good version. To recover from such a state, a hardware debugger is used as discussed in Section 2.4.4. For this reason, during initial testing it is important to limit the extent to which modifications are performed and, if feasible, be prepared for testing with multiple physical copies of the target device.

3.5.2 Demonstration.

Given a tested and validated solution implemented to take advantage of design weaknesses in the derived firmware update validation method, a demonstration of counterfeit firmware is presented. The demonstration is intended to verify the weaknesses inherent in common controllers as well as validate the threats they pose. The demonstration first creates a valid firmware image containing customized functionality. Relevant functions in the firmware are sought by searching through the recovered symbol names. Once relevant sections of code are identified, modifications are made to achieve the desired effect. As described by Peck and Peterson [44], care should be taken to minimize the collateral damage caused by the modifications in order to improve the chance that the modified firmware still functions as expected. After the modifications are performed, the implemented solution is applied to the counterfeit firmware image, resulting in a valid custom image as shown in testing. The demonstration proceeds to upload the counterfeit firmware to the test environment device. This process is again accomplished through the use of the vendor-supplied firmware update software in the same manner as any legitimate firmware update. If the device enters a faulted state, then the counterfeit firmware failed. Otherwise, if the counterfeit firmware is confirmed as running on the device and performing its altered function correctly, then the firmware is successfully counterfeited, achieving the goals of the demonstration.

3.6 Summary

In an effort to advance detection and analysis research in the area of secure PLC firmware, this research intends to determine the feasibility of a counterfeit firmware attack on a common PLC. It is proposed that such an attack is possible due to insecurely designed firmware update validation methods. A compilation of relevant practices and procedures is described for deriving the firmware update validation method and considerations regarding its advantages and limitations are discussed. Following this, the derived method is confirmed and analyzed while a demonstration of any design weaknesses is provided, allowing for counterfeit firmware to be uploaded and executed on the test device.

IV. Reversing Process, Testing, and Demonstration

This chapter describes in detail the process followed to reverse engineer the firmware update validation method of the Allen-Bradley ControlLogix L61. After acquiring and completely disassembling the device firmware, techniques to derive the firmware update validation method are explored and assessed. Any weaknesses in the derived method are taken advantage of and verified through a demonstrated counterfeiting of firmware on the test system.

4.1 Reversing Process

4.1.1 Firmware Acquisition.

Firmware samples are first sought directly from the manufacturer. A search of the vendor website reveals numerous ControlLogix L61 OS firmware update packages available for download from Rockwell Automation. The firmware revision number (FRN)s available range from the newest, FRN 20.013, to what is presumably the oldest officially supported version for the targeted L61 Series B device, FRN 12.042. For completeness, all 19 available versions are procured.

Once the firmware updates are downloaded, the firmware binary images are extracted from the update package. In the case of ControlLogix updates, the downloaded packages are zip-compressed files containing a Windows installer (.msi extension) for the ControlFlash utility as well as a text file named CONTENTS.TXT. Using the FRN 19.011 update package as an example, opening the CONTENTS.TXT file in a text editor reveals a list of “script filenames” for each CPU module model number supported by this update. In this case, only one script file is mentioned: PN-86270.nvs. Next, the 7-Zip file compression utility is used to view the contents of the ControlFLASH.msi installer file. Located inside are several cabinet files (.cab extension). A search is performed over each

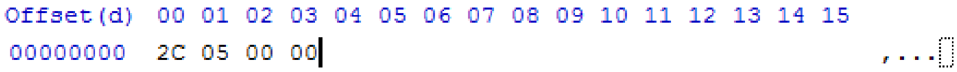
.cab file for the "PN-86270.nvs" filename listed in CONTENTS.TXT. This file is found inside the NVS.CAB file. This archive contains a series of subdirectories that, in the lowest directory, contains three files: PN-86270.nvs, PN-86270.RES, and PN-86272.bin. Since PN-86272.bin is the largest of the three and the .bin extension implies a binary file, it is a candidate for the firmware image.

It is noted that not all of the downloaded firmware updates follow the structure pattern outlined here. Several older FRN updates contain an executable setup file and an unfamiliar virtual disk-like archive. Since this only applies to 4 of the oldest updates from the total 19 available, they are excluded from the reversing process in order to focus on the newer and more immediately accessible firmware images.

4.1.2 Binary File Analysis.

4.1.2.1 Manual Inspection.

After obtaining multiple OS firmware images, a visual inspection is performed manually on a sample image. This case continues to use FRN 19.011. Note, however, that the purpose of this step is to identify general format characteristics of the firmware common to the test device. Therefore, any available version is sufficient for this step based on its inherent format required by the device. The manual inspection begins by examining the files contained in NVS.CAB. Since PN-86272.bin is the candidate firmware image, the other two files are first examined to ascertain their relevance. Opening the .nvs file in a text editor reveals a configuration script for the update (see Appendix B.1). Relevant information contained in this file includes: version number, number of updates, a list of eligible devices, starting location for the update, firmware file size, and firmware filename. This configuration file accurately identifies the firmware binary file as PN-86272.bin and confirms its size. Opening PN-86270.RES in the HxD binary file editor as illustrated in Figure 4.1 presents 4 bytes of data, which are later determined to represent ControlFlash restrictions on the firmware binary (see Section 4.3.2).



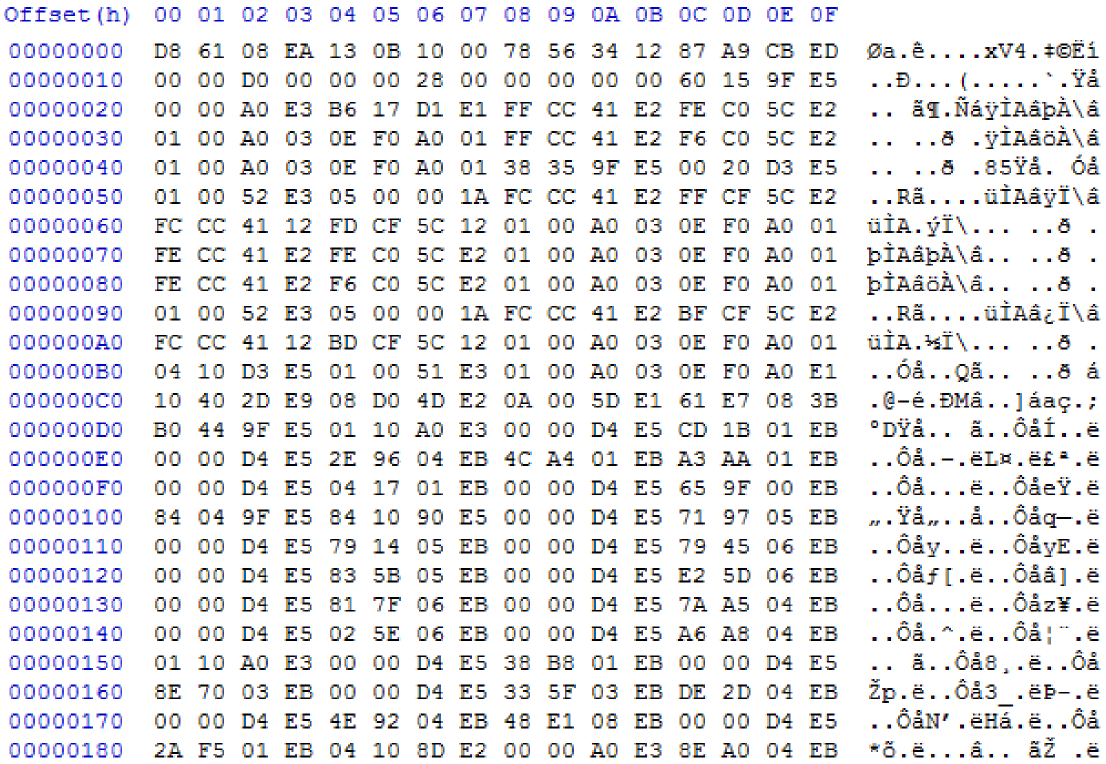
```
Offset(d)  00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00000000   2C 05 00 00                                      ,...
```

Figure 4.1: Contents of PN-86270.RES in HxD.

The firmware image file PN-86272.bin, seen in Figure 4.2, is next examined using HxD. The first goal is to assess the general structure of the binary file by identifying any encrypted or compressed section. If such sections exist, additional work involving decryption or decompression is performed before continuing the reversing process. Observations made with the sample image indicate that there exist no immediately visible compressed or encrypted sections. This is determined by a marked lack of randomness in the bytes of the image as shown in Figure 4.2. Furthermore, several American Standard Code for Information Interchange (ASCII) strings are readily visible in the binary file. Therefore, the general structure of the image is indicative of raw binary code and data. An inspection beginning with the first bytes in the file reveals possible header information. Specifically, the first 7 to 8 32-bit words of the image are of interest due to the existence of the string “xV4” and the relatively high density of 0s, especially in the 7th word, which is entirely 0.

4.1.2.2 Binary Comparison.

Having determined a possible header and discovered that the firmware binary is likely not compressed or encrypted, binary comparisons are performed on the firmware files. Using the Visual Binary Diff (VBinDiff) binary file difference utility, two firmware images are loaded at a time while VBinDiff highlights any differences that are present. In order to identify dynamic sections of the firmware, two similar firmware images are first sought. Consecutive firmware versions are expected to have many similarities, so working backwards from the most recent FRN available, every two consecutive version pairs are compared for similarities. Considering that the size of the firmware images ranges from

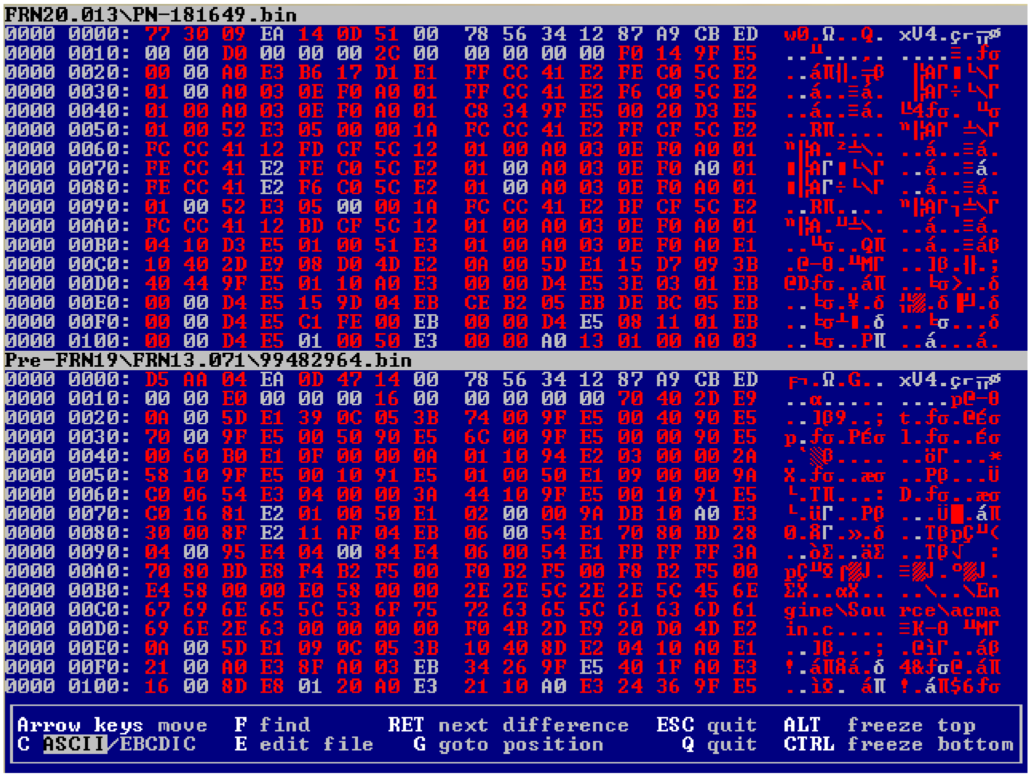


```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   D8 61 08 EA 13 0B 10 00 78 56 34 12 87 A9 CB ED   Øa.ê....xV4.‡©Ëí
00000010   00 00 D0 00 00 00 28 00 00 00 00 00 60 15 9F E5   ..Ð...(.....`.Ÿå
00000020   00 00 A0 E3 B6 17 D1 E1 FF CC 41 E2 FE C0 5C E2   .. ã¶.ÑáÿÌAâþÀ\â
00000030   01 00 A0 03 0E F0 A0 01 FF CC 41 E2 F6 C0 5C E2   .. ..ð .ÿÌAâöÀ\â
00000040   01 00 A0 03 0E F0 A0 01 38 35 9F E5 00 20 D3 E5   .. ..ð .85Ÿå. Óå
00000050   01 00 52 E3 05 00 00 1A FC CC 41 E2 FF CF 5C E2   ..Rã....üÌAâÿÏ\â
00000060   FC CC 41 12 FD CF 5C 12 01 00 A0 03 0E F0 A0 01   üÌA.ýÏ\... ..ð .
00000070   FE CC 41 E2 FE C0 5C E2 01 00 A0 03 0E F0 A0 01   þÌAâþÀ\â.. ..ð .
00000080   FE CC 41 E2 F6 C0 5C E2 01 00 A0 03 0E F0 A0 01   þÌAâöÀ\â.. ..ð .
00000090   01 00 52 E3 05 00 00 1A FC CC 41 E2 BF CF 5C E2   ..Rã....üÌAâ¿Ï\â
000000A0   FC CC 41 12 BD CF 5C 12 01 00 A0 03 0E F0 A0 01   üÌA.½Ï\... ..ð .
000000B0   04 10 D3 E5 01 00 51 E3 01 00 A0 03 0E F0 A0 E1   ..Óå..Qã.. ..ð á
000000C0   10 40 2D E9 08 D0 4D E2 0A 00 5D E1 61 E7 08 3B   .@-é.ÐMâ..]áaç.;
000000D0   B0 44 9F E5 01 10 A0 E3 00 00 D4 E5 CD 1B 01 EB   °DŸå.. ã..ÔåÍ..ë
000000E0   00 00 D4 E5 2E 96 04 EB 4C A4 01 EB A3 AA 01 EB   ..Ôå.–.ëL¤.ë£ª.ë
000000F0   00 00 D4 E5 04 17 01 EB 00 00 D4 E5 65 9F 00 EB   ..Ôå...ë..ÔåeŸ.ë
00000100   84 04 9F E5 84 10 90 E5 00 00 D4 E5 71 97 05 EB   „.Ÿå„..å..Ôåq—.ë
00000110   00 00 D4 E5 79 14 05 EB 00 00 D4 E5 79 45 06 EB   ..Ôåy..ë..ÔåyE.ë
00000120   00 00 D4 E5 83 5B 05 EB 00 00 D4 E5 E2 5D 06 EB   ..Ôåƒ[.ë..Ôåâ].ë
00000130   00 00 D4 E5 81 7F 06 EB 00 00 D4 E5 7A A5 04 EB   ..Ôå...ë..Ôåz¥.ë
00000140   00 00 D4 E5 02 5E 06 EB 00 00 D4 E5 A6 A8 04 EB   ..Ôå.^.ë..Ôå¦¨.ë
00000150   01 10 A0 E3 00 00 D4 E5 38 B8 01 EB 00 00 D4 E5   .. ã..Ôå8¸.ë..Ôå
00000160   8E 70 03 EB 00 00 D4 E5 33 5F 03 EB DE 2D 04 EB   Žp.ë..Ôå3_.ëÞ-.ë
00000170   00 00 D4 E5 4E 92 04 EB 48 E1 08 EB 00 00 D4 E5   ..ÔåN’.ëHá.ë..Ôå
00000180   2A F5 01 EB 04 10 8D E2 00 00 A0 E3 8E A0 04 EB   *õ.ë...â.. ãŽ .ë
```

Figure 4.2: Beginning of FRN19.011 binary in HxD.

roughly 2MB to 3MB, it is not feasible to identify and compare each and every difference. Instead, firmware image similarity is evaluated based on the quantity of differences visible at the beginning and end of the the files as well as the magnitude of difference in file lengths. To accurately compare the end of the files, the last bytes in the files are aligned in VBinDiff to correct for variations in length. Not only does such evaluation simplify the comparison process, but the beginning and ends of the firmware images are most likely to contain information relevant to the entire image as opposed to random code instructions contained in the middle. Examples of this are commonly witnessed in many data structure implementations as headers and footers including the portable executable (PE) file format, Ethernet frames, and VxWorks-based firmware images [26, 37, 44]. The two most similar firmware versions discovered following this method of comparison are FRN 16.081 and

FRN 16.057. Note that these two versions are the only two firmware versions available with the same length. Furthermore, the number of byte-differences between the two totals only 14 throughout their entire files.



```
FRN20.013\PN-181649.bin
0000 0000: 77 30 09 EA 14 0D 51 00  78 56 34 12 87 A9 CB ED  w0.Ω..Q. xV4.ç⌐╦φ
0000 0010: 00 00 D0 00 00 00 2C 00  00 00 00 00 F0 14 9F E5  ..╨...,. ....≡.ƒσ
0000 0020: 00 00 A0 E3 B6 17 D1 E1  FF CC 41 E2 FE C0 5C E2  ..áπ╢.╤ß  ╠AΓ■└\Γ
0000 0030: 01 00 A0 03 0E F0 A0 01  FF CC 41 E2 F6 C0 5C E2  ..á..≡á.  ╠AΓ÷└\Γ
0000 0040: 01 00 A0 03 0E F0 A0 01  C8 34 9F E5 00 20 D3 E5  ..á..≡á. ╚4ƒσ. ╙σ
0000 0050: 01 00 52 E3 05 00 00 1A  FC CC 41 E2 FF CF 5C E2  ..Rπ.... ⁿ╠AΓ ╧\Γ
0000 0060: FC CC 41 12 FD CF 5C 12  01 00 A0 03 0E F0 A0 01  ⁿ╠A.²╧\. ..á..≡á.
0000 0070: FE CC 41 E2 FE C0 5C E2  01 00 A0 03 0E F0 A0 01  ■╠AΓ■└\Γ ..á..≡á.
0000 0080: FE CC 41 E2 F6 C0 5C E2  01 00 A0 03 0E F0 A0 01  ■╠AΓ÷└\Γ ..á..≡á.
0000 0090: 01 00 52 E3 05 00 00 1A  FC CC 41 E2 BF CF 5C E2  ..Rπ.... ⁿ╠AΓ┐╧\Γ
0000 00A0: FC CC 41 12 BD CF 5C 12  01 00 A0 03 0E F0 A0 01  ⁿ╠A.╜╧\. ..á..≡á.
0000 00B0: 04 10 D3 E5 01 00 51 E3  01 00 A0 03 0E F0 A0 E1  ..╙σ..Qπ ..á..≡áß
0000 00C0: 10 40 2D E9 08 D0 4D E2  0A 00 5D E1 15 D7 09 3B  .@-Θ.╨MΓ ..]ß.╫.;
0000 00D0: 40 44 9F E5 01 10 A0 E3  00 00 D4 E5 3E 03 01 EB  @Dƒσ..áπ ..╘σ>..δ
0000 00E0: 00 00 D4 E5 15 9D 04 EB  CE B2 05 EB DE BC 05 EB  ..╘σ.¥.δ ╬▓.δ▐╝.δ
0000 00F0: 00 00 D4 E5 C1 FE 00 EB  00 00 D4 E5 08 11 01 EB  ..╘σ┴■.δ ..╘σ...δ
0000 0100: 00 00 D4 E5 01 00 50 E3  00 00 A0 13 01 00 A0 03  ..╘σ..Pπ ..á...á.
Pre-FRN19\FRN13.071\99482964.bin
0000 0000: D5 AA 04 EA 0D 47 14 00  78 56 34 12 87 A9 CB ED  ╒¬.Ω.G.. xV4.ç⌐╦φ
0000 0010: 00 00 E0 00 00 00 16 00  00 00 00 00 70 40 2D E9  ..α..... ....p@-Θ
0000 0020: 0A 00 5D E1 39 0C 05 3B  74 00 9F E5 00 40 90 E5  ..]ß9..; t.ƒσ.@Éσ
0000 0030: 70 00 9F E5 00 50 90 E5  6C 00 9F E5 00 00 90 E5  p.ƒσ.PÉσ l.ƒσ..Éσ
0000 0040: 00 60 B0 E1 0F 00 00 0A  01 10 94 E2 03 00 00 2A  .`░ß.... ..öΓ...*
0000 0050: 58 10 9F E5 00 10 91 E5  01 00 50 E1 09 00 00 9A  X.ƒσ..æσ ..Pß...Ü
0000 0060: C0 06 54 E3 04 00 00 3A  44 10 9F E5 00 10 91 E5  └.Tπ...: D.ƒσ..æσ
0000 0070: C0 16 81 E2 01 00 50 E1  02 00 00 9A DB 10 A0 E3  └.üΓ..Pß ...Ü█.áπ
0000 0080: 30 00 8F E2 11 AF 04 EB  06 00 54 E1 70 80 BD 28  0.ÅΓ.».δ ..TßpÇ╜(
0000 0090: 04 00 95 E4 04 00 84 E4  06 00 54 E1 FB FF FF 3A  ..òΣ..äΣ ..Tß√  :
0000 00A0: 70 80 BD E8 F4 B2 F5 00  F0 B2 F5 00 F8 B2 F5 00  pÇ╜Φ⌠▓⌡. ≡▓⌡.°▓⌡.
0000 00B0: E4 58 00 00 E0 58 00 00  2E 2E 5C 2E 2E 5C 45 6E  ΣX..αX.. ..\..\En
0000 00C0: 67 69 6E 65 5C 53 6F 75  72 63 65 5C 61 63 6D 61  gine\Sou rce\acma
0000 00D0: 69 6E 2E 63 00 00 00 00  F0 4B 2D E9 20 D0 4D E2  in.c.... ≡K-Θ ╨MΓ
0000 00E0: 0A 00 5D E1 09 0C 05 3B  10 40 8D E2 04 10 A0 E1  ..]ß...; .@ìΓ..áß
0000 00F0: 21 00 A0 E3 8F A0 03 EB  34 26 9F E5 40 1F A0 E3  !.áπÅá.δ 4&ƒσ@.áπ
0000 0100: 16 00 8D E8 01 20 A0 E3  21 10 A0 E3 24 36 9F E5  ..ìΦ. áπ !.áπ$6ƒσ
```

Arrow keys move   F find   RET next difference   ESC quit   ALT freeze top
C ASCII/EBCDIC   E edit file   G goto position   Q quit   CTRL freeze bottom

Figure 4.3: VBinDiff of the beginning of FRN20.013 and FRN13.071.

A comparison is also performed on two dissimilar versions to determine what static fields are present. Optimal dissimilarity is defined as the greatest possible number of byte-differences between two valid firmware images of given lengths. Given two such dissimilar images, bytes that remain unchanged between them represent static data that is likely constant across most firmware versions. For this comparison, however, optimal dissimilarity is not mandatory. To simplify comparison, the process considers that the newest and oldest available firmware versions are dissimilar to a degree sufficient in identifying static fields. For the given test environment, these versions are FRN 20.013

and FRN 13.071, respectively. Loading these two files into VBinDiff shows that much of the expected header is the same (see Figure 4.3). These static fields include the entire 3rd, 4th, and 7th words, with only one byte differing in each of the 5th and 6th words. An examination of the end of these two files first length aligns the last bytes in each file. This comparison reveals no significant similarities.

As a result of comparing the various firmware images, noticeable patterns arise. For example, the 3rd, 4th, and 7th words of the header remain constant. In addition, the second word is identified as containing the firmware version number of each image: the first byte represents the major revision number, the second is the minor revision number, and the third is the subrevision number found in the accompanying .nvs file. However, none of the header values directly relate to the file's length. Through review of the dynamic fields revealed by FRN 16.081 and FRN 16.057, the last 8 bytes of every firmware image are found to always differ in an apparently random fashion (see Figure C.2). This is strong evidence that these trailing 8 bytes represent a validation value.

4.1.2.3 Embedded File and Filesystem Analysis.

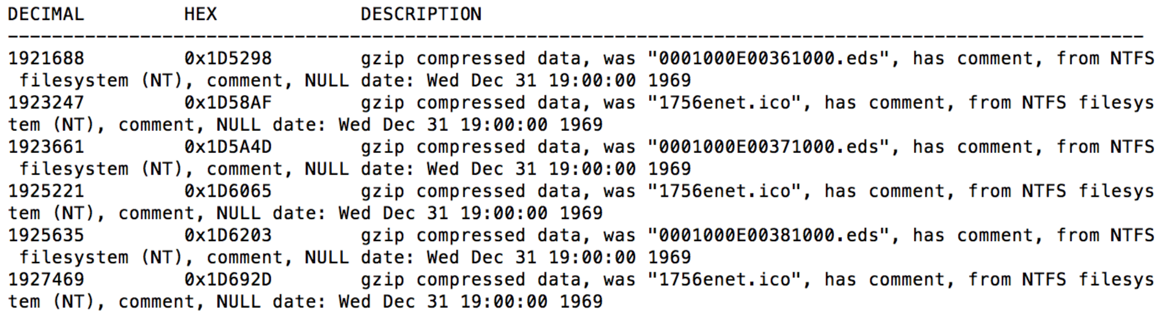
Following manual and comparative analysis, the target firmware is now analyzed for embedded files or filesystems. This is accomplished by searching the firmware binary for byte signatures matching file or filesystem types of interest. Based on previous work discussed in Section 2.4, such types commonly encountered in similar applications include zlib, gzip, or Lempel-Ziv-Markov chain algorithm (LZMA) compressed files and compressed ROM file system (cramfs), SquashFS, Journalling Flash File System version 2 (JFFS2), or Yet Another Flash File System (YAFFS) filesystems, which commonly incorporate the former compression schemes. Previous works by Santamarta and Peck utilize an automated tool to perform this type of analysis called Deezee [44, 52]. However, Deezee only searches specifically for zlib-compressed sections, so this research employs the BinWalk binary analysis tool capable of detecting all of the above [23]. From this point forward, FRN 16.081 is the firmware image targeted by this effort. This selection is made based on its similarity to another version (FRN 16.057) as well as the fact that, at over 1MB smaller than the newest firmware, there is significantly less data to analyze. The decision is made based on the assumption that this size advantage comes with no significant variation regarding implementation of the validation method or other firmware features critical to operation.

As a command line tool, Binwalk requires a firmware binary file as an argument for execution. Given the sample firmware image, Binwalk is instructed to search for the above signatures. The result of this analysis reveals 6 gzip file candidates and approximately 170 zlib file candidates. Binwalk identifies the gzip file candidates as 3 icon files and 3 .eds files (see Figure 4.4). The .eds extension signifies a configuration file used by Rockwell devices to identify themselves to control software such as RSLinx or RSLogix. Binwalk does not identify corresponding filenames for the zlib results; therefore, they are analyzed using a custom script and an open-source zlib compression utility (see Appendix E.1 and Reference [2], respectively). The script extracts all zlib containers identified by Binwalk from the firmware image and attempts to decompress them, returning only those that successfully decompress. This method determines that only 4 out of the approximately 170 original hits are valid zlib containers. A manual inspection of these remaining 4 yields no significant information. None of the resulting files are greater than 8 bytes in length. With no filesystems identified, these results indicate that, except for the identified gzip files, the firmware image is raw binary code and data.

4.1.3 Firmware Disassembly.

4.1.3.1 Processor Determination and Disassembly.

Before loading the firmware image into IDA to automate disassembly, the target processor type is determined. Using Skochinsky's presentation [53], the firmware binary as viewed in HxD is compared to different target processor code samples. This comparison



```
DECIMAL         HEX             DESCRIPTION
--------------------------------------------------------------------------------------------------------
1921688         0x1D5298        gzip compressed data, was "0001000E00361000.eds", has comment, from NTFS
 filesystem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
1923247         0x1D58AF        gzip compressed data, was "1756enet.ico", has comment, from NTFS filesys
tem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
1923661         0x1D5A4D        gzip compressed data, was "0001000E00371000.eds", has comment, from NTFS
 filesystem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
1925221         0x1D6065        gzip compressed data, was "1756enet.ico", has comment, from NTFS filesys
tem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
1925635         0x1D6203        gzip compressed data, was "0001000E00381000.eds", has comment, from NTFS
 filesystem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
1927469         0x1D692D        gzip compressed data, was "1756enet.ico", has comment, from NTFS filesys
tem (NT), comment, NULL date: Wed Dec 31 19:00:00 1969
```

Figure 4.4: List of gzip file candidates identified by Binwalk.

of instruction signatures indicates that the firmware targets an ARM processor. As an alternative to this method, a physical examination of the hardware also confirms that the target processor is an ARM (see Appendix D).

IDA provides two different ARM targets: one based on little-endian byte ordering and the other based on big-endian. The little-endian target processor is initially selected, as indicated by the Skochinsky code samples. This results in the automatic disassembly of several functions in the binary, indicating the correct determination of the target processor. For comparison, the big-endian target is also attempted, but IDA fails to disassemble any code, confirming that the target is little-endian. Figure 4.5 illustrates the initial status of the binary image disassembly in IDA, where blue represents successfully identified and disassembled functions, red represents unknown disassembly code, gray represents identified data sections, and brown represents unexplored sections of the image.

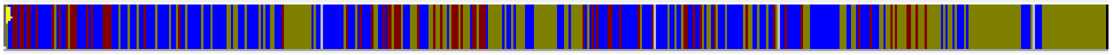


Figure 4.5: Initial IDA disassembly status.

4.1.3.2 Rebuilding Functions.

IDA is not able to completely disassemble the firmware binary and a significant portion of the binary image remains unexplored by the automated tool. Therefore, the IDA script file provided by Santamarta [51] manually instructs IDA to explore the remaining functions. In order to apply this script to the firmware, it is modified for use on the ARM architecture by searching for the appropriate ARM function prologues (see Appendix F.1). Identified functions from the code IDA natively disassembles are examined to determine their prologue signature. All such functions automatically identified by IDA begin with a *store multiple with full descending stack address mode* (STMFD) instruction that pushes current register values to the stack. The exact registers pushed to the stack vary, but the two most significant bytes of the instruction remain constant: 0xE9 0x2D. A review of the ARM Procedure Call Standard specified in the ARM Software Development Toolkit Reference Guide confirms that this instruction is a standard prologue for ARM functions [4]. Furthermore, a review of the ARM instruction encoding standard confirms that for this particular instruction, the most significant two bytes are always 0xE9 0x2D [6]. With Santamarta's IDA script modified to target this signature and corrected for endianness, the script is applied to the loaded firmware binary. As a result, the IDA status visualization bar confirms that a significantly greater majority of the binary is explored with functions identified (see Figure 4.6).

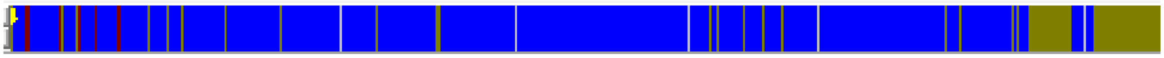


Figure 4.6: IDA status after function identification.

4.1.3.3 Determining Base Address.

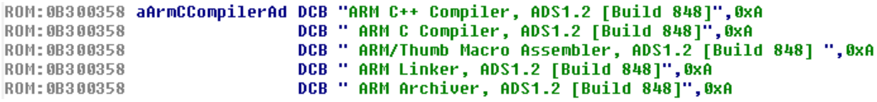
Determining the base address is the next step followed in the process. The "load immediate" technique is used initially given that there exist immediate address references in the disassembly. A search of all *load register* (LDR) instructions referencing immediate values determines that the majority of such references consist of addresses in the range of 0x10000 to 0x30000. The assumption that these addresses reference locations in the firmware implies a base address of 0. However, the firmware is currently loaded at a base of 0 and no immediate addresses reference the beginning of functions or strings. Therefore, they do not reference sections of the firmware. Based on the volume of such references, the address range is significant, but the significance is unknown.

After the complete search of immediate values present in LDR instructions, other common address bases are identified including (in descending order of approximate commonness): 0x08000000, 0x00C00000, 0x0B000000, 0x00E00000, and 0x60000000. Recall that the starting location referenced in the .nvs file accompanying the firmware is 0x0B160000. Given that this location is titled as “starting location” and falls within the range of several immediate values in the general 0x0B000000 base, it is incorporated as the true base address. In IDA, the firmware is rebased at the new address. No significant changes in the disassembled code are immediately visible. However, many instructions in the given firmware use relative addresses independent of the base address, so significant changes are not expected. For the same reason, even proceeding with an incorrect base address does not technically hamper the reversing process. The majority of instructions utilize relative addressing, so function interactions and operations remain consistent regardless of the base address. However, knowledge of the true base address is still critical in understanding the firmware as a whole. While many of the function references remain correct, working with an incorrect base address may lead to inaccurate interpretations of segments referenced by immediate addresses.

4.1.3.4 Inspecting Strings.

In order to gain as much information as possible from the firmware image, an inspection of all ASCII strings contained in the image is performed. IDA provides a subview to list all strings in the loaded binary. Alternatively, the UNIX *strings* command provides equivalent functionality, only without the support of IDA disassembly. A manual inspection of the list reveals several Extensible Markup Language (XML) strings. A review of official documentation from Rockwell Automation determines that these strings are related to the use of CompactFlash memory cards as storage for project files [50].

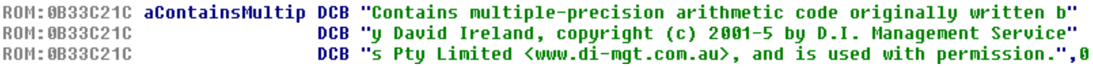
Also discovered is a set of strings indicative of ARM compiler versions used to build the firmware (see Figure 4.7). These strings indicate both that Rockwell uses standard ARM development tools as well as the time period of original firmware development. For comparison, a search of the newest firmware, FRN 20.013, for strings containing “ARM” produces one result for “ARM ADS1.2 [Build 848].” This indicates that development tool information is simplified in newer revisions, but remains present.



```
ROM:0B300358 aArmCCompilerAd DCB "ARM C++ Compiler, ADS1.2 [Build 848]",0xA
ROM:0B300358                 DCB " ARM C Compiler, ADS1.2 [Build 848]",0xA
ROM:0B300358                 DCB " ARM/Thumb Macro Assembler, ADS1.2 [Build 848] ",0xA
ROM:0B300358                 DCB " ARM Linker, ADS1.2 [Build 848]",0xA
ROM:0B300358                 DCB " ARM Archiver, ADS1.2 [Build 848]",0xA
```

Figure 4.7: ARM compiler version string.

Another string in the firmware refers to a multiple-precision math library (see Figure 4.8). A similar string in the associated .nvs file, refers to this library by the name BigDigits (see Appendix B.1). Since official documentation for BigDigits specifically describes its use in cryptographic applications, the inclusion of this string indicates possible cryptographic functionality present in the firmware [18]. This exact string remains present in FRN 20.013.

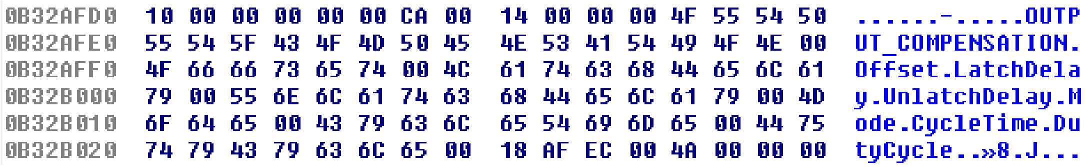


```
ROM:0B33C21C aContainsMultip DCB "Contains multiple-precision arithmetic code originally written b"
ROM:0B33C21C                 DCB "y David Ireland, copyright (c) 2001-5 by D.I. Management Service"
ROM:0B33C21C                 DCB "s Pty Limited <www.di-mgt.com.au>, and is used with permission.",0
```

Figure 4.8: BigDigits library copyright string.

Furthermore, a string containing an apparently random combination of hexadecimal characters (i.e., 0-9 and a-f) is also present in the firmware. This string is 128 characters (bytes) long, or 1024 bits, a common key length used for Rivest, Shamir, Adleman (RSA) encryption schemes. In fact, 1024 bits is the suggested key length used for RSA in corporate or medium-security environments during the time frame of the release of the ControlLogix L61. However, the NIST no longer recommends the use of 1024-bit RSA keys through 2013 and expressly prohibits its usage thereafter [8]. If this string does represent an RSA key, it is possible the BigDigits library is included in the firmware to implement the encryption scheme. Whether or not this is related to the firmware update validation method is unknown at this point in the process. Of significance, this exact string is still present in FRN 20.013.

Considering that one goal of string inspection is to identify symbol strings in concert with discovering symbol tables, the existence of apparent source filename strings in the firmware is significant. For example, one such string reads "..\\..\\Source\\acmain.c." In addition, strings representing apparent data structures are also present. These strings exist in blocks organized with a structure name followed by attributes (see Figure 4.9). The exact format of these data structures remains unknown.



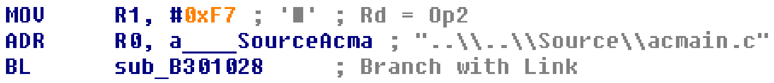
```
0B32AFD0  10 00 00 00 00 00 CA 00  14 00 00 00 4F 55 54 50  ......¯.....OUTP
0B32AFE0  55 54 5F 43 4F 4D 50 45  4E 53 41 54 49 4F 4E 00  UT_COMPENSATION.
0B32AFF0  4F 66 66 73 65 74 00 4C  61 74 63 68 44 65 6C 61  Offset.LatchDela
0B32B000  79 00 55 6E 6C 61 74 63  68 44 65 6C 61 79 00 4D  y.UnlatchDelay.M
0B32B010  6F 64 65 00 43 79 63 6C  65 54 69 6D 65 00 44 75  ode.CycleTime.Du
0B32B020  74 79 43 79 63 6C 65 00  18 AF EC 00 4A 00 00 00  tyCycle..»8.J...
```

Figure 4.9: OUTPUT_COMPENSATION data structure strings.

4.1.3.5 Rebuilding Symbols.

To rebuild symbols in the firmware, there are two basic requirements: symbol names and a way to associate those names with the functions they belong to. As determined by the string inspection, there exist a number of source code filenames in the firmware. A further inspection of these strings, however, reveals that they are not located in one common section of the binary. Instead of being located in a symbol table, the source filenames are distributed throughout the firmware image. While this discovery hampers the reversing efforts, the fact that file names are available is significant.

During disassembly, IDA automatically creates cross references between string addresses and the addresses of any instructions that reference them. Using these cross references, instructions referring to several of the source filename strings are investigated and compared. This reveals that all of the strings are used in the same manner: they are passed in as parameters to a common function. An example is shown in Figure 4.10.



```
MOV     R1, #0xF7 ; '█' ; Rd = Op2
ADR     R0, a____SourceAcma ; "..\\..\\Source\\acmain.c"
BL      sub_B301028      ; Branch with Link
```

Figure 4.10: Example symbol string usage.

A brief exploration of this common function reveals a series of calls that apparently never return. The observed behavior closely resembles the exception or assertion functionality mentioned by Skochinsky [53]. For this reason, the function is manually renamed “exception_call” in IDA. While this information does not provide a direct mapping of names to functions as a symbol table does, it is possible to infer the source filename of any function making an exception call. However, when multiple functions making exception calls are contained in the same source file, the filename arguments passed to exception_call are identical. Thus, there is not a one-to-one mapping of names to functions, but a one-to-many mapping. As a consequence of the same issue, the names available are less descriptive and specific than individual function names provided by a symbol table.

Since the source filename strings are distributed throughout the firmware, the process of naming their related functions is less straightforward, but still automatable with a script (see Appendix F.2). Every function referencing each filename string is identified using the IDA cross reference database. The script then renames the functions according to the referenced filename. Duplicates are commonly encountered in this process and handled by adding generic numerical suffixes to each function name.

4.1.4 Derivation of Firmware Update Validation Method.

4.1.4.1 Disassembly Analysis.

Reverse engineering the validation method from disassembled code begins by considering the general functionality of validation algorithms in order to determine relevant patterns and structures in the disassembled code. For instance, the validation algorithm is known to inherently perform a computation over the contents of the firmware; therefore, the code calculating the validation likely contains a loop performing an operation over a range of memory addresses. Furthermore, the specific operations performed in that loop are implied by the type of algorithm used for validation.

As Section 2.4.5 discusses, two common types of algorithms used for validation methods are the modular summation and CRC algorithms. While numerous other algorithms exist that a validation method may implement, the possible configurations are countably infinite. Therefore in the interest of pragmatism, the two candidates above are considered the primary candidates due to their popularity in related applications. A modular summation algorithm adds each *n* bit word of the target data together to create a checksum value. For such an algorithm, the main validation loop should contain an addition operation executed over each word in the image. A CRC, however, should implement a more complicated set of operations that includes the use of an XOR instruction. While the use of an XOR instruction is not guaranteed depending on the exact coded implementation, it remains a computationally inexpensive instruction and a common method of implementing the CRC calculation [46].

Given these considerations, the list of firmware functions is searched for names relevant to any checksum algorithm possibly used for validation. Once an apparently relevant function is identified, the surrounding disassembly is examined for operational flow indicative of such an algorithm. One example of a function discovered in this manner is identified by the “cmCS.c” source filename. Since the “CS” abbreviation in the file name may indicate “checksum,” the function is explored. However, an analysis of the disassembly reveals no operational flows indicative of a checksum calculation, so the function is deemed not significant. Several other strings are discovered following this process including the binary file extension “.bin,” a set of functions referencing the “Encryption.c” source filename, a large set of “up” functions including source filenames “upexec.c” and “upprog.c,” and the “ReUpLockForUpdate.c” function set. Again, these functions are examined based on possible relevance of their names, but while general information regarding their operation is gained, no specific information regarding the firmware update validation method method is discovered.

During this process, the first word in the firmware image is discovered to contain a branch instruction. IDA is instructed to treat this location as the beginning of a function, revealing initialization code at the branch target location. This initialization code is followed and analyzed for over one thousand instructions, however, no code related to the functionality of the validation method is discovered. Considering that a CRC, if present, likely uses an *exclusive or* instruction (represented as “EOR” in ARM assembly), another tactic attempted is a direct search of the disassembly for all EOR instructions. At 263 occurrences, the number of results returned by this search is too numerous to exhaustively analyze, but several of the results containing loop structures are explored. However, no code relevant to the validation method is discovered. Since the validation method remains to be located among the disassembled functions of the OS firmware, other available approaches are explored.

4.1.4.2 Black Box Testing.

For black box testing, incremental alterations of firmware are tested on the device to gain information regarding its operation. Since the true firmware update validation method is present on the device, the success of a particular tactic can only reliably be determined by modifying the firmware, uploading it to the device, and assessing the result. Anytime modifications are made at such a low level, there exists a significant risk of bricking the device. Although it may be possible to recover the device with hardware debugging methods, such ability is not guaranteed. This hazard presents a challenge to reversing efforts by restricting the modifications that can be safely attempted while minimizing the risk of harm to the device. If multiple physical sample devices exist for such testing, the threat posed is less severe, but prohibitively expensive; damage is ideally avoided.

To begin black box efforts, two pieces of information are required: the location of the checksum value in the firmware and the range of data the validation method covers. It is not guaranteed that the validation method strictly covers the range of the first byte of the binary image to the last. As determined in Section 4.1.2.2, the candidate location of the checksum value is the last 8 bytes of the image. To determine the range of bytes covered by the validation method, various bytes of the firmware image are modified such that a series of modified firmware uploads indicates whether the modified bytes are validated.

Boundary Checking. Ideally, the suspected validation boundaries are tested systematically by changing and testing each and every byte in the image. However, one caveat to this strategy is the risk of bricking the device. Certain bytes that are ideally tested throughout the image are critical to the proper operation of the firmware. Section 4.1.4.1 shows that the first 4 bytes of the firmware image represent a branch instruction to the OS initialization code. If the first byte of the firmware is modified and it passes validation, the OS should theoretically fail to initialize properly. To avoid this behavior potentially bricking the device, boundary checking modifications skip such critical bytes. Instead, the nearest non-critical bytes are targeted under the assumption that any adjacent critical bytes are included in the validation range. In this case, the first non-critical byte of the firmware is the 5th, or the first byte of the version number in the header. Starting at the 5th byte, modifications are made incrementally to each byte in the firmware. Initially, the first version byte is incremented by 1 and the modified firmware is uploaded to the device with ControlFlash. This results in a failed validation after the image is uploaded; therefore, the version number byte is included in the validation.

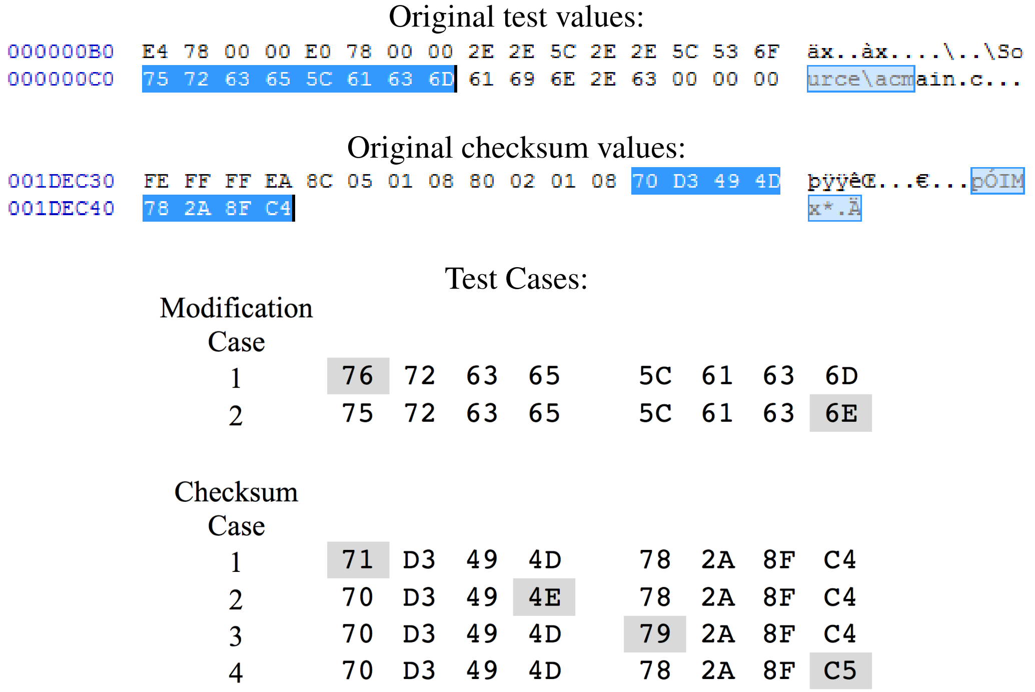
Since the process of boundary checking requires the firmware to be updated at every test point, it is not feasible to check every non-critical byte in the firmware. Not only must the criticality of every byte be determined manually, but the firmware upload process is slow. For each update, the binary is pushed to the device in a process that takes several minutes. If the upload is successful, the device proceeds to restart itself. Even if the process is automated using a custom flash program, the time required to write images to the device alone takes a prohibitive amount of time to allow a thorough check of every non-critical byte. For this reason, this process continues by checking only the likely boundaries. These include the beginning and end of the header and the beginning and end of the main code section, excluding the 8 byte checksum value. In addition, several arbitrarily chosen non-critical bytes in the middle of the firmware are also selected and tested. The results for all such checks are the same: all modifications are detected by the validation method resulting in a failed update. This evidence strongly suggests that the validation method in use is dependent on every byte of the firmware image.

Common Checksum Algorithms. Given this evidence, a number of common checksum algorithms are applied to the firmware image. The binary file used in these tests is the FRN 16.081 image with the last 8 bytes removed in order to exclude the checksum value from the calculations. HxD features the ability to calculate many common hashes over a file. The following available algorithms are applied to the test file: checksum-8 (checksum here refers to a modular summation), checksum-16, checksum-32, checksum-64, CRC-32, CRC-64, SHA-1, SHA-256, SHA-384, SHA-512, message digest (MD)-2, MD-4, and MD-5. The results of these calculations are compared to the actual checksum value of the firmware. Comparisons are made based on any observable similarities between the HxD calculated values and the true checksum value. Considering that most of the algorithms applied produce results of differing lengths from the 64-bit firmware checksum value, any apparent similarities are potentially significant. In this case, however, no similarities are observed.

Since the target processor for the firmware operates in little-endian mode, a second attempt is made to repeat this procedure after converting the test file to big-endian byte ordering. Afterwards, the same algorithms above are again applied to the new big-endian version of the test image. However, this makes no apparent difference in the comparison of similarities with the original firmware checksum value. An investigation is next initiated specifically on the two common candidate algorithms: the modular summation and the CRC. In a modular summation implementation, the checksum value is directly dependent on the value of each n-bit word before it, so a difference of 1 in any given word translates directly to a difference of 1 in the calculated checksum value. This characteristic enables a method of determining whether a modular summation algorithm is in use or not.

The target of the difference-of-1 modification is the first non-critical byte in firmware that occurs in the least significant position of a word. With a standard modular summation algorithm, a difference of 1 in such a byte results is a difference of 1 in the corresponding least significant byte of the checksum value. Since endianness and word size of the checksum value are unknown, two target bytes are chosen to be modified: the first byte and last byte of a 64-bit aligned word. The first byte addresses the little-endian 32 and

64-bit possibilities while the last byte addresses the big-endian 32 and 64-bit possibilities. One at a time, these values are modified by an increment of 1. For each of these two cases, 4 different bytes in the checksum value are tested: the first and last bytes of each 32-bit word. Again, this covers the possibilities that the true checksum value is either 32-bit or 64-bit and big or little-endian. Additionally, in the case that the true checksum value is only 32-bits, this checks both possible word positions for the 32-bit value. For each of these possibilities, the checksum value byte is also incremented by 1. Figure 4.11 illustrates the 8 total test cases present, 2 firmware modification cases for each of 4 checksum value modification cases. Each of these test cases is uploaded to the device using ControlFlash. All 8 firmware test cases fail validation; therefore, it is concluded that the checksum method is not a simple modular summation.



Original test values:

```
000000B0   E4 78 00 00 E0 78 00 00 2E 2E 5C 2E 2E 5C 53 6F   äx..àx....\..\So
000000C0   75 72 63 65 5C 61 63 6D 61 69 6E 2E 63 00 00 00   urce\acmain.c...
```

Original checksum values:

```
001DEC30   FE FF FF EA 8C 05 01 08 80 02 01 08 70 D3 49 4D   þÿÿêŒ...€...pÓIM
001DEC40   78 2A 8F C4                                       x*.Ä
```

Test Cases:

Modification Case								
1	76	72	63	65	5C	61	63	6D
2	75	72	63	65	5C	61	63	6E

Checksum Case								
1	71	D3	49	4D	78	2A	8F	C4
2	70	D3	49	4E	78	2A	8F	C4
3	70	D3	49	4D	79	2A	8F	C4
4	70	D3	49	4D	78	2A	8F	C5

Figure 4.11: Modular summation test cases with changes highlighted.

Brute Forcing CRC. As Section 2.4.5 describes, a CRC is based on a number of customizable parameters including bit width, polynomial value, initial checksum value, final XOR value, and endianness. Gregory Cook provides a command line tool for calculating and determining CRCs with arbitrary parameters [15]. This tool, called CRC RevEng or RevEng, is applied to determine if a CRC algorithm configuration is used for validation. RevEng includes a database of common CRC models. In total, there are 64 different default CRC configurations including nine 32-bit models and three 64-bit models. The search functionality in RevEng checks all known models of the specified width against multiple specified sample files. If a match is not found in one of the default models, a brute force search is initiated through all possible model configurations. Sample input files are required to be in the format of message data followed immediately by the checksum value of the specified width. With minimum knowledge of the true model's parameters, the model width and and at least three input samples are specified such that at least two of the samples are identical in length and at least two differ in length.

To test the ControlLogix L61 firmware for a CRC algorithm, seven unique test cases are established. All cases are based on the same three firmware samples: FRN 16.081, FRN 16.057, and FRN 16.022. These satisfy the minimum sample criteria since FRN 16.081 and FRN 16.057 are the same length while FRN 16.022 differs in length. The first case supposes a 64-bit width and little-endian byte ordering. The exact manner in which RevEng or the device validation method handle little-endian byte ordering for 64-bit width models is unknown, but hypothetically RevEng may operate based on 64-bit little endian words while the device checksum operates on two 32-bit little endian words, or vice versa. For the former case, the files are left unmodified since their format already follows the data-plus-checksum format required. However, to ensure the data is treated properly for the latter, new firmware sample files are created where each two–32-bit word pair in the original samples are swapped. This ensures that at least one of the two variations is processed by RevEng in the same manner as the device checksum. The next test case, 64-bit width big-endian, also uses the unmodified firmware files since big-endian does not change the effective word ordering.

The remaining test cases all assume a 32-bit width. Since the checksum value at the end of the image is 64-bits, these cases assume only one of the two 32-bit words is a CRC value; the other must represent something else. As such, there are two possibilities for which 32-bit word represents the CRC value and for each of those, two possibilities for endianness. The three sample firmware image files are appropriately modified by removing the first 32-bit word of the checksum value for the first case and the second 32-bit word for the second case. Endianness has no effect on the sample files themselves for the 32-bit width, only when executing RevEng must it be specified. These four combinations in addition to the three above constitute the seven cases tested (see Figure 4.12).

```
reveng -w 64 -l -s -f PN-66834.bin PN-66830.bin PN-70325.bin
reveng -w 64 -l -s -f PN-66834_word_swapped.bin PN-66830_word_swapped.bin PN-70325_word_swapped.bin
reveng -w 64 -s -f PN-66834.bin PN-66830.bin PN-70325.bin
reveng -w 32 -l -s -f PN-66834_first.bin PN-66830_first.bin PN-70325_first.bin
reveng -w 32 -s -f PN-66834_first.bin PN-66830_first.bin PN-70325_first.bin
reveng -w 32 -l -s -f PN-66834_second.bin PN-66830_second.bin PN-70325_second.bin
reveng -w 32 -s -f PN-66834_second.bin PN-66830_second.bin PN-70325_second.bin
```

Figure 4.12: Terminal commands for RevEng search cases.

Since the default models are checked first, RevEng returns the matching model almost immediately if a match is found. However, none of the test cases match any default model, so RevEng continues to run through its brute force test. RevEng does not include functionality to check the search progress, so after 24 hours of constant runtime, the processes are terminated with no results. Since RevEng is open source, the source code is modified to include progress indication by printing the current attempted polynomial value upon entering the Control+Z key combination. RevEng tests polynomials incrementally, so this value provides a direct indication of the program's progress. RevEng is run again on one of the test cases to obtain a rough approximation of the rate of search progression. The search is run for 15 minutes on an Intel Core i7 processor running at 2.3 GHz. The Control+Z key combination is pressed at 5, 10, and 15 minutes. Averaging the returned polynomial values over the given time intervals results in a linear progression of approximately 10,000 polynomials checked per minute. At this rate, a search for a 64-bit polynomial with 2^{64} combinations, would take approximately 3.5 billion years. A 32-bit polynomial search with 2^{32} combinations is shorter, but would still take approximately 10 months. These time requirements alone are impractical before taking into consideration the additional calculations required to determine initial checksum and final XOR values. Although RevEng is currently not parallelized, even a highly parallelized version running on more advanced hardware would take a significant amount of time. Depending on time and cost constraints, a 32-bit parallelized search may be possible, but a 64-bit search is not.

4.1.4.3 Hardware Debugging.

In this section, the alternative of hardware debugging is explored. The underlying operation of the device hardware and firmware is first considered. General information about how the device handles firmware is gained from the attempted firmware updates through this process. Initially, the device is received from the manufacturer loaded with only a base firmware (FRN 1.010). This base firmware provides basic functionality to allow for the device to be updated with a true OS firmware, allowing proper operation [49].

When updating with ControlFlash, the device enters a special state while firmware is pushed to it. Upon receipt of the entire firmware image, the device immediately validates it. If the image passes validation, the device automatically resets itself and boots into the new OS. However, if the image fails validation, the device enters an error, or faulted, state where it requires a physical reset by cycling the power. When the device restarts, it reverts back to the base firmware (FRN 1.010). This same behavior is observed whether updating from the base firmware to an OS version or from one OS version to another. Even if an OS is already installed, a bad update reverts the device to the base firmware. This implies that flashing a new OS image overwrites the previous OS before validation is attempted. Since the determination on whether to boot into the OS or the base firmware is made on a hard reset, there must exist loader code on the device with the ability to determine which image to execute. This may be accomplished either via checking a "valid" flag stored in non-volatile memory or a direct validation of the OS image on each startup.

The existence of such a loader with the ability to validate firmware on its own provides another possible target to reverse engineer the validation method. Furthermore, it is possible that in order to update the firmware, the OS running in memory makes the equivalent of a system call to the loader or base firmware to perform the actual validation. Indeed, the loader or base firmware may completely handle the firmware update process. The validation method used on the OS may not even be present in the OS firmware image. Regardless, either the loader or base firmware contain the validation method, otherwise an OS firmware update is not possible in the first place. Since the underlying binary of these code segments is not openly available through updates like the OS, hardware debugging techniques are used to access them directly on the device.

Locating JTAG Ports. Since JTAG is the common standard for hardware debugging interfaces, the device is physically examined for possible JTAG TAPs. After physically disassembling the device to access the circuit board, unknown and unused connectors are targeted first for investigation. For the ControlLogix L61, there are no unused connectors present on the board, so the search moves on to empty solder pads and test points. A number of unused solder pads are present, but in order to identify them as JTAG TAPs, their corresponding signals must be verified. Since the device is ARM-based, a search is conducted for standard ARM JTAG pinouts. Two common ARM JTAG configurations available are 20-pin and 14-pin layouts. One apparent empty connector pad on the controller board also has 14 pins, so it is a prime candidate for a JTAG interface.

An initial visual inspection finds no signs that any of the solder pads of the candidate connector trace to any non-IC components other than a resistor. A multimeter is now used to test the pinout signals of the candidate. Ground and power signals are the most straightforward to identify, so the pins are tested against ground and power while their layout is compared to the 14-pin ARM JTAG reference pinout. All 8 combined power and ground pins correspond between the reference pinout and the candidate connector pads, providing initial evidence that the empty connector is for JTAG. The other candidate pins should also be confirmed using Breeuwsma's process [10], but in practice this is difficult. When the controller is fully assembled for operation as required by the remainder of Breeuwsma's method, the candidate TAPs are inaccessible to manual probing. To access them, lead wires must be soldered directly to the candidate points.

Further manual analysis is performed first in an attempt to more easily confirm the candidate pads as TAPs. Reliably tracing pins on a multi-layer circuit board is challenging without expensive equipment or tedious point-testing and luck. However, a visual inspection of the candidate pads determines that the pad coinciding with the system reset pin (nSRST) of the 14-pin ARM JTAG reference pinout is connected to a pull-up resistor. Since nSRST is active low, this is an appropriate configuration. Unless the nSRST pin is grounded, the pull-up resistor pulls the nSRST signal high by default; thus, the system is defaulted to not reset. This evidence further supports that the pinout is a 14-pin ARM JTAG connector and implies that an attempted connection to the device through JTAG is safe for the hardware. To connect physically, a connector is fabricated and soldered to the candidate TAP connector pads. Through this physical link, an attempt is made to connect to the device using the hardware debugger. The primary purpose of this is to verify that the TAP signal pins are correctly identified.

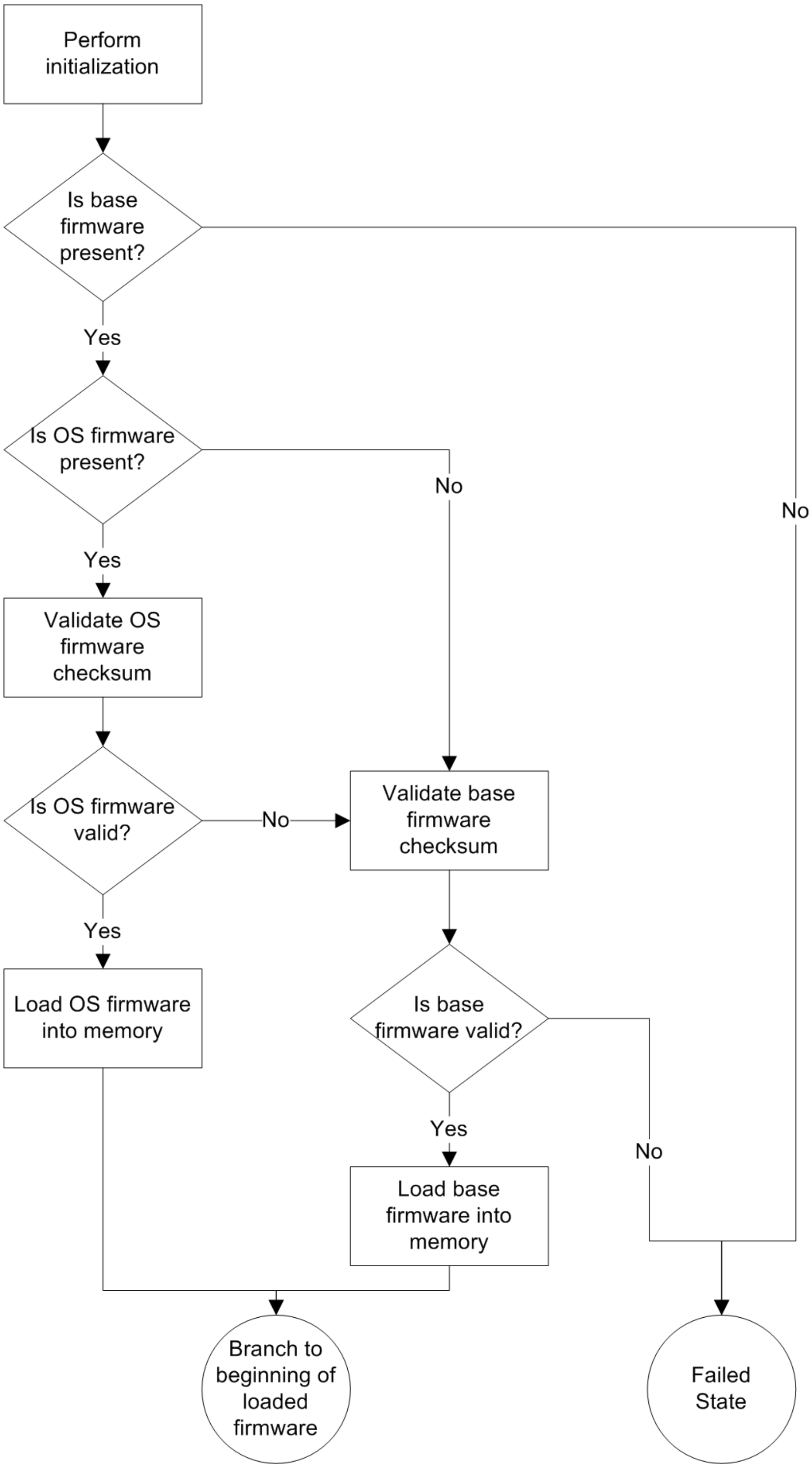
Debugger Configuration. The ARM RealView ICE device together with the ARM Development Studio 5 (DS-5) software are used to debug the controller. Using the soldered connector, the ControlLogix L61 is connected to the ICE hardware. Initial configuration is required with DS-5 regarding the specifics of the target processor. The ControlLogix L61 uses a custom ARM processor with exceeding rare documentation. However, the provided RealView ICE configuration utility is able to automatically detect the target core of an attached ARM processor. In this case, the automatic configuration identifies the core as an ARM7TDMI. The successful detection of the processor core confirms that the connector in use is a JTAG interface to the device and the TAPs have been correctly identified. After initial configuration, the DS-5 target database is updated to include the new target configuration. This is accomplished with a command line utility provided by DS-5. After updating the database, DS-5 is configured to connect with the target device by creating a new debugger instance and setting the target type as the controller's newly added entry to the configuration database.

With the debugger configured, the first attempts are made to connect to the target. The connections succeed, but the device cannot be stopped for debugging. When a stop command is given, the controller enters a faulted state and becomes unresponsive, requiring a manual restart. An attempt to connect immediately after a manual reset determines that execution on the controller can only be stopped before the OS finishes booting. This means the debug connection must be made as soon as the device is powered on before the OS has time to boot. The cause of this anomaly remains unconfirmed, but a possible explanation is that halting the processor while the OS is running triggers a fail-safe response from a watchdog timer. While the processor is halted, the watchdog timer is not properly fed, so a fault is triggered when the timer expires in an attempt to prevent unsafe operation of the system.

Memory Image Acquisition and Analysis. Once connected to the debugger and halted, execution on the device is stopped in its pre-boot state. At this point, the debugger is used to perform a memory dump on the device. Lacking any information of the proper mapping of device memory, as much of the 32-bit address space is dumped as the debugger can access. A manual analysis of the acquired memory dump in HxD reveals that much of the dumped address space repeats itself. After accounting for duplicated memory, several sections of binary code are identified by searching for common ARM instruction patterns. The OS firmware is found among these code segments in two distinct locations: one starting at the address 0x0B1A0000 and again at address 0x00D00000. Another large code segment is found at address 0x0B020000. The beginning of this segment contains a header of the same format as the OS firmware. Inspection of the version number field reveals that this segment is FRN 1.010, the base firmware.

In addition to these firmware images, two other short and unfamiliar code segments are found: one at address 0x0A000000 and another at address 0x80000000. The two unknown code segments are extracted from the dump and the same reversing process described in Sections 4.1.2 and 4.1.3 is applied to them. This produces disassembled code in IDA for functional analysis. Analysis of the 0x0A000000 segment reveals multiple source filename strings akin to the OS firmware. Among the strings are the names “ExecLoader.s” and “hw_setup.s.” Reverse engineering of the function referencing “ExecLoader.s” in IDA confirms that, based on its functionality, this particular segment is the executive loader. After performing hardware initialization, ExecLoader follows a process to validate and load the firmware into memory before handing over execution.

In this process, illustrated in Figure 4.13, ExecLoader begins by determining what, if any, firmware is present. To accomplish this, ExecLoader first verifies the existence of base firmware. The base address of the base firmware is hard-coded as 0x0B020000. Using this base address, ExecLoader compares the 3rd and 4th words of the presumed base firmware



Perform initialization
Is base firmware present?
Yes
No
Is OS firmware present?
Yes
No
Validate OS firmware checksum
Is OS firmware valid?
No
Yes
Validate base firmware checksum
Load OS firmware into memory
Is base firmware valid?
Yes
No
Load base firmware into memory
Branch to beginning of loaded firmware
Failed State

Figure 4.13: Flow chart of ExecLoader.s.

header. If these two values are bitwise inverses of each other, ExecLoader accepts that a base firmware is present. If no base firmware is detected, execution on the device is terminated in an infinite loop. However, if the presence of base firmware is confirmed, ExecLoader continues by determining if OS firmware is present. Given the OS firmware base address, ExecLoader performs the same check above on the 3rd and 4th words of the presumed OS firmware header. Again, if these two values are bitwise inverses, ExecLoader accepts that an OS firmware is present.

ExecLoader now validates the firmware. If an OS firmware is present, the validation is performed on it; otherwise, ExecLoader defaults to validating the base firmware. The algorithm used for validation is a variation of a modular summation. In the event an OS firmware is present but does not pass validation, ExecLoader proceeds to validate the base firmware. If no firmware passes validation, execution is terminated in an infinite loop. After successful validation of the appropriate firmware, ExecLoader loads the firmware from flash memory to RAM. To determine the address in RAM where the firmware is loaded, ExecLoader references the 5th word of the firmware header. The firmware is copied to this address. Afterwards, execution jumps from ExecLoader to the base address of the loaded firmware. At this point in the process, a candidate for the firmware update validation method is known.

4.2 Firmware Update Validation Method Analysis

4.2.1 Verification of Correctness.

The validation method checksum algorithm derived by the reversing process is now tested for correctness. The validation algorithm is recreated in C as a command line program called `ab_cksum`. This program is able to validate existing firmware images and is applied to all available firmware versions. Each of the 15 firmware versions from FRN 20.013 to FRN 13.071 is passed to the `ab_cksum` program. The program applies the derived validation algorithm to the firmware binary files. All 15 firmware versions tested in this manner successfully pass the `ab_cksum` validation, confirming correctness of the algorithm.

4.2.2 Design Analysis.

Given the success of initial testing, the design of the derived validation algorithm is analyzed for weaknesses that allow for counterfeit firmware to pass validation. The design of the modular summation algorithm implemented by the firmware update validation method is based on the notion that in order to be considered a valid image, the modular summation of its contents must equal a given value. This design has several inherent security weaknesses from the perspective of integrity and authenticity. By design, a modular summation algorithm is meant only to validate data integrity, not authenticity. Furthermore, the integrity of the data is only protected against accidental alteration, and this is not completely guaranteed. Therefore, taking advantage of a modular summation algorithm to violate data integrity and authenticity is trivial. In the specific implementation of the ControlLogix L61, the key requirement to pass validation is to match the constant value used by the algorithm for comparison. As long as the modular summation of the firmware results in this constant value, the firmware is considered valid by the controller.

A solution taking advantage of this weakness is implemented in `ab_cksum`. The goal of the solution is to ensure that the modular summation algorithm, when applied to a given firmware image, results in the correct constant validation value by modifying only the checksum value field in the firmware. This is accomplished by applying the algorithm to the firmware image, but excluding the checksum value field from the calculation. The resulting value is then subtracted from the constant validation value, and the result is placed in the checksum value field.

The validation design weakness is tested on the ControlLogix L61 controller using FRN 16.081 as the baseline image. To minimize the risk of damaging the test device, the first attempted modification does not alter the checksum value field. Instead, a modification resulting in a collision of checksum values is manually generated in the firmware image. The resulting checksum value is not changed, as validated by `ab_cksum`. To test this solution, the firmware is uploaded to the controller using ControlFlash. After the upload is complete, the controller restarts itself, unlike previous invalid updates, but then fails to boot. Even after a manual reset, the device is completely unresponsive.

4.2.3 Refinement.

With the test device in an inoperable state, JTAG hardware debugging is employed to recover the device. After connecting to the device through JTAG as previously described, the debugger reveals that code execution on the device terminates in an infinite loop. However, the OS firmware is still loaded into memory from flash. The DS-5 debugging software is used to verify that the firmware loaded into memory is indeed the modified test version by locating the modified bytes in RAM. This demonstrates the executive loader is successfully validating the OS image and loading it into memory, but that the OS apparently fails to correctly initialize.

The location of the error is determined by iteratively setting hardware breakpoints in the debugger, restarting the system, and observing if the breakpoints are hit or not. Following this process, the location of the error is reduced to one particular unidentified function in the OS firmware. Referring to the disassembled firmware in IDA, this function is examined and determined to contain another checksum algorithm: a CRC. This evidence establishes that the validation performed by the executive loader is interpreted correctly, but that a secondary validation in the OS initialization continues to hinder modification.

Iteratively repeating the testing process described above, the `ab_cksum` utility is revised to include the ability to validate firmware based on the discovered CRC validation algorithm. The correctness of the updated utility is verified by passing all available firmware versions to `ab_cksum`. After applying both the modular summation and CRC algorithms to the firmware images, `ab_cksum` reports that they all successfully pass validation. This confirms the correctness of the revised utility.

A design analysis of the CRC algorithm determines that it suffers from the same security weakness as the modular summation algorithm. Although a CRC is able to more accurately detect accidental alterations of data than a modular summation, the CRC is not designed to detect intentional modifications. As long as the CRC calculation implemented in the ControlLogix L61 results in a constant value, the firmware is considered valid. This solution is implemented in `ab_cksum` such that, given a firmware image, the utility performs the derived CRC validation and places the result in the CRC checksum field.

In the current case where the loaded firmware does not pass CRC validation, the device enters a faulted state. The device is first recovered to a usable state before testing continues. Since the modified OS firmware currently loaded on the device is able to pass the ExecLoader validation but fails its own CRC validation, external intervention from the hardware debugger is required for recovery. After resetting the device and connecting through JTAG, the debugger is used to divert execution in the executive loader and directly load the base firmware instead of the OS firmware. This is accomplished by modifying the processor program counter when execution in ExecLoader reaches the point of detecting OS firmware, effectively acting as though no OS firmware is present. With the base firmware temporarily running on the device, a known good OS image is uploaded to the device in the standard manner, recovering the device to a fully operational state.

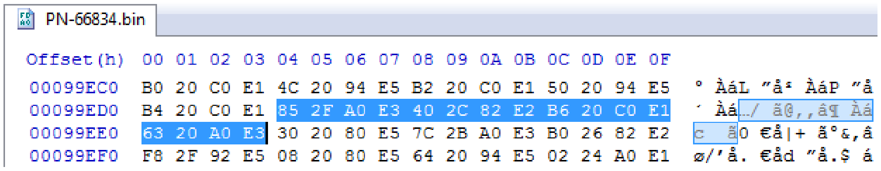
To test the refined `ab_cksum` solution, a one-byte modification is made to the test firmware in a non-critical location. The checksum values are recalculated using `ab_cksum` to replace the original values. The firmware is uploaded to the device using the standard ControlFlash method. After the update is complete, the controller restarts and successfully boots completely into its standard operating state. This is confirmed by the successful communication of the device with RSLinx device configuration software. Thus, the validation method of the device is successfully bypassed.

4.3 Demonstration

4.3.1 Firmware Modification.

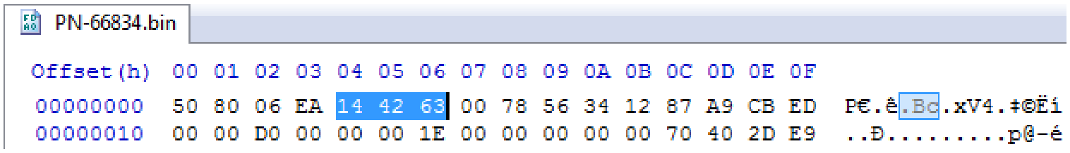
The process for firmware modification begins by reversing the firmware binary to obtain named functions in the disassembly in exactly the same manner as previously described. Based on the goal of the modification, the disassembly is searched for any potentially relevant strings or function names. The following case study demonstrates the concept. The goal in this case is to counterfeit the firmware version number FRN 16.081 to appear as FRN 20.066.099, a value higher than any currently available version number from Rockwell.

First, the firmware is searched for any locations that reference the version number. IDA detects no code references to the version number bytes in the header, but there must exist a reference to version number in the code since the device reports a version number to ControlFlash. Since FRN 16.081 only differs from FRN 16.057 by 14 bytes, the binary files of these two versions are compared. An inspection of the disassembly surrounding differences in the versions determines that one such difference exists in a function returning a hard-coded version number. Figure 4.14 highlights 4 ARM instructions that use immediate values to calculate and store this version number. The immediate values of these instructions are appropriately modified in HxD to represent the target version number 20.066.099. The version number bytes in the firmware header are also appropriately modified (see Figure 4.15), where 0x14 = 20, 0x42 = 66, and 0x63 = 99. Using the `ab_cksum` validation utility, the correct checksum values are calculated and updated in the new firmware binary file. The utility then revalidates the binary to confirm that the checksum values are correctly updated.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00099EC0  B0 20 C0 E1 4C 20 94 E5 B2 20 C0 E1 50 20 94 E5  ° ÀáL ”å² ÀáP ”å
00099ED0  B4 20 C0 E1 85 2F A0 E3 40 2C 82 E2 B6 20 C0 E1  ´ Àá…/ ã@,‚â¶ Àá
00099EE0  63 20 A0 E3 30 20 80 E5 7C 2B A0 E3 B0 26 82 E2  c  ã0 €å|+ ã°&‚â
00099EF0  F8 2F 92 E5 08 20 80 E5 64 20 94 E5 02 24 A0 E1  ø/’å. €åd ”å.$ á
```

Figure 4.14: Modification of FRN 16.081 version number in function to 20.066.099.



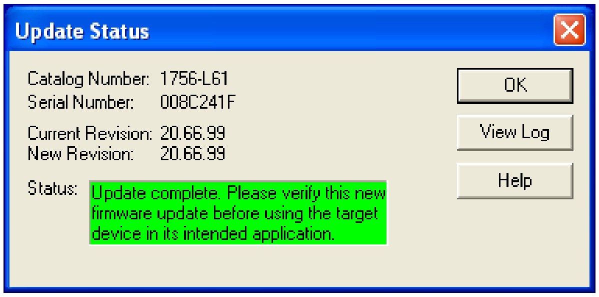
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  50 80 06 EA 14 42 63 00 78 56 34 12 87 A9 CB ED  P€.ê.Bc.xV4.‡©Ëí
00000010  00 00 D0 00 00 00 1E 00 00 00 00 00 70 40 2D E9  ..Ð.........p@-é
```

Figure 4.15: Modification of FRN 16.081 version number in header to 20.066.099.

4.3.2 Device Exploitation.

In order to use ControlFlash to upload the counterfeit firmware, adjustments are made to the .nvs configuration file. The firmware version number is specified several times in the configuration file, so these values are modified to reflect the counterfeited version number. The modified firmware now appears in the ControlFlash firmware catalog list as version 20.66.99. Rockwell is inconsistent with whether subrevision numbers include a leading 0 or not, so 20.66.99 is equivalent to 20.066.099. This demonstration continues to use the same test environment present throughout this research, which includes the ControlLogix L61 controller. The device is first updated with the legitimate FRN 20.013 firmware from Rockwell in order to test the effectiveness of a counterfeit update on the newest major revision. ControlFlash is then used to update the device with the counterfeit FRN 20.066. The update successfully completes, the device restarts, and the new 20.66.99 revision number is reported to ControlFlash indicating a successful update as shown in Figure 4.16.

While ControlFlash is used in this case, a custom utility could also be developed to perform the same operation. When attempting more complex firmware modifications,



Update Status

Catalog Number: 1756-L61
Serial Number: 008C241F

Current Revision: 20.66.99
New Revision: 20.66.99

Status: Update complete. Please verify this new firmware update before using the target device in its intended application.

Figure 4.16: Successful firmware update to FRN 16.081 with spoofed 20.66.99 version number.

ControlFlash poses certain complications. In oder to process the catalog of available updates, ControlFlash references a restriction file (.RES extension) in addition to the configuration script and firmware image. This .RES file is used to determine if its associated firmware has any usage restrictions. In addition, the .RES file contains yet another type of checksum value for the firmware image. This particular algorithm is exceptionally prone to collisions, so minor modifications are not detected, but extensive modifications including changes in image length require the value to be recalculated. Since .RES file processing is handled exclusively by ControlFlash, a custom update utility avoids this validation. Given the minor modifications demonstrated in the case above, however, the .RES file value does not require recalculation.

4.4 Discussion

In order to develop an understanding of the requirements for successful firmware modification, this section discusses the prerequisites, advantages, and limitations of the various techniques utilized throughout the reversing process. The discussion includes issues related to time, cost, and complexity requirements of the various techniques as well as their applicability and any problems encountered while applying the process to the ControlLogix L61.

For the ControlLogxix L61, firmware update packages are readily available from the vendor through Rockwell Automation's website. Furthermore, it is straightforward to access the firmware image inside these update packages. The combination of these circumstances provide a significant advantage in obtaining a wide range of firmware samples with little time, cost, or effort. However, the prerequisite to the advantages of technique is a target device with upgradable firmware where the manufacturer makes such updates openly available. Without such availability, the acquisition process is complicated significantly.

The general format of a given firmware image is technically observable by viewing the image in a binary file editor. However, manual detection of these structural features is dependent on the particular firmware being analyzed and the personal experience of the investigator. For example, an encrypted or obfuscated firmware image would impose significant time limitations on the investigation. Although, if successful this technique may provide the advantageous identification of potential image header and general structure information.

Binary comparison of firmware is accurately capable of identifying dynamic and static fields in the image. The advantage of binary comparison is the critical identification of a candidate checksum value field as well as addition information regarding the image header. This comes at no significant time, cost, or complexity requirements. However, the technique is limited by the availability of multiple firmware sample images, where its effectiveness and accuracy is directly proportional to the number of samples available. During investigation of the ControlLogix L61 firmware, initial comparison of the ends of firmware images corrected for length by aligning the last bytes of the two compared images. However, additional analysis conducted after the reversing process and demonstration reveals that this approach is not the necessarily the most effective due to potential variations in the image trailer length. Data initially identified as unrelated at the ends of FRN20.013

and FRN 13.071 is actually nearly identical when offset by 8 bytes. This difference is a result of the fact that FRN 13.071 lacks a CRC checksum value. Additional comparison with FRN 15.060 reveals that the CRC checksum validation is present, so this functionality is apparently added after FRN 13.071. Note that no FRN 14 samples are available from Rockwell.

The signature-based analysis for embedded files and filesystems is limited by the availability of a raw image that is not encrypted, compressed, or obfuscated. Additionally, the Binwalk tool used in this case results in a number of false positives that require additional time to examine. Still, it is a relatively straightforward process to eliminate them as candidates, which is also aided by automation. Observations from this technique indicate the accuracy of Binwalk varies depending on the target signature type. While the gzip results are highly accurate, the zlib results are as equally inaccurate. The information gained from this step indicates that the firmware contains mostly raw binary code and data, which is significant. Had more significant files been embedded in the firmware, such as a compressed symbol table as described by Peck and Peterson [44], the advantage of this technique would significantly increase.

The significant advantage of processor identification is that it enables automated disassembly using a tool such as IDA. In this case, the technique of instruction signature comparison correctly identifies the processor architecture with no significant time or complexity requirements. However, one limitation is the direct cost incurred by using the IDA tool since it is a licensed software product. The alternative method of physical analysis may also be limited by cost requirements since a physical device must be available and physically disassembled.

The technique of searching for prologue signature provides the critical advantage of identifying the locations of functions that follow the conventional prologue. These functions are then available for further analysis. However, not all functions are guaranteed to do so. In the former case, as with the majority of functions in the test environment, function identification is accomplished with no significant time, cost, or complexity requirements beyond the cost of IDA. Considering the majority of functions follow the standard prologue convention, this technique is highly effective in providing a more complete representation of the firmware.

The “load immediate” technique is technically capable of determining the correct base address, but primarily amounts to an educated version of a brute force attack. It does not incur any direct cost, but requires a significant amount of time to accomplish. The process of identifying and testing immediate address values is straightforward, but the complexity of implementation is introduced by the need to repeatedly test various possible solutions. The success of this step requires disassembled firmware and the existence of multiple immediate address references inside the scope of the firmware image. The advantage of success is a proper base address that facilitates reverse engineering of functions by providing accurate address references. While incremental progress is made using the “load immediate” technique with the ControlLogix L61, the “starting location” value in the firmware’s associated .nvs configuration file apparently indicates a base address value. The use of the starting location value 0x0B160000 as the base address corroborates other references to memory in the 0x0B000000 range, but the true base address in RAM is later confirmed as 0x00D00000 (see Section 4.1.4.3). An analysis of all relevant address references in the firmware must be performed regardless of any implied base address.

The technique of string examination may potentially determine a significant amount of information regarding details of the firmware implementation. Also of significance with the ControlLogix L61 is the lack of any copyright strings explicitly identifying the OS used by the firmware. This eliminates several OS implementations as candidates, including VxWorks, since they are known to contain copyright strings. As the number of strings present in the image increase, however, so does the required amount of time to examine them. Execution of this technique is not significantly difficult, where the majority of the required effort is spent on researching the significance of discovered strings. The sample firmware image used for this technique need not necessarily be disassembled for the inspection, but must inherently contain strings of informational value. Despite these limitations, the resulting information enables a more detailed understanding of the firmware, including the identification of function names, although some of the information is not necessarily relevant to deriving the validation method.

The ability to rebuild symbol names in the firmware is heavily dependent on the availability of function or source file names in the firmware from the string inspection. For the ControlLogix L61 firmware, the latter is available, providing successfully renamed functions. However, a lack of such information severely limits the success of the overall process. Further limitations of this technique include the ability to determine how the available name strings relate to functions in the disassembly as well as writing a script to automate the rebuilding. Despite these limitations, the successful application of this technique provides the significant advantage of named functions. The availability of function names enables the investigator to focus reverse engineering efforts on specifically targeted functions.

From a technical perspective, the technique of disassembly analysis is capable of successfully determining the location and operation of any validation method present in the firmware. In practice, however, time constraints limit the amount of progress that can possibly be made. Additionally, while there are no direct cost requirements, the process of determining candidate functions and reversing their functionality is significantly complex and dependent on knowledge and experience. The ability of this technique to find relevant functions in the disassembly is reliant on the effectiveness of the function naming process. An analysis of all symbol or source filename strings is critical to understanding the disassembled code. Furthermore, a complete list of names does not guarantee that the investigator can identify relevant names. Multiple searches of the function name list are suggested to minimize this possibility. For example, initial review of the ControlLogix L61 source filenames failed to identify the name of a function related to the firmware update validation method. One apparently obscure name is later determined to represent a function that writes firmware updates to flash memory and performs the modular summation checksum validation in the OS. This discovery confirms that disassembly analysis of the OS firmware alone would derive the validation method given enough time and experience. If successful, the derived validation method enables analysis and testing on the security of its design. In this particular investigation, time and complexity constraints prevent this technique from determining the validation method. However, it remains effective as a general technique to reverse specifically targeted areas of firmware code as demonstrated in Section 4.1.4.3.

The presented technique of boundary checking is technically capable of determining the range of data included in the validation method. Time and complexity requirements incurred by critical bytes limit the ability to comprehensively check every firmware byte. In addition, unknown variables, such as the fact that the ControlLogix L61 algorithms operate over memory beyond the image, complicate matters. This case specifically makes boundary checking, already complicated by concerns of damaging the device, ineffective without such knowledge. The technique to check for common classes of firmware update validation method algorithms can potentially determine if any of the specifically tested algorithms are in use, but is complicated by variations on the basic cases tested. The detection of modular summation class algorithms should have tested inverse relationships using the difference of 1 technique. For example, the inclusion of cases to check for changes by -1 would correctly detect that a summation algorithm is in use for the ControlLogix L61 validation; however, this would still result in bricking the device given the additional CRC validation. Regardless, determining whether the true firmware update validation method is a class of the tested cases or not provides advantageous insight either way. Without the limitations of cost and time, a brute force technique is technically capable of determining the validation method given a known class. However, with no guarantee that the firmware update validation method implements a CRC, the additional complexity of having an unknown class of firmware update validation method creates a countably infinite number of possibilities, making a brute force attack futile. Furthermore, the brute force test demonstrates that the time requirement alone precludes this approach from being truly effective. Overall, the black box approach is effective at boundary checking and testing common validation methods, but beyond these common configurations, black box testing can not effectively derive the true validation method.

The significant advantage provided by hardware debugging tools reinforces the requirement of obtaining a test device. In addition to hardware debugging, a physical reference device is required for testing modifications and allows for physical examination of the device circuit board. Immediate cost requirements include the expenses of the physical device under examination as well as any tools needed to aid in identification of the ports. Time and complexity requirements for the ControlLogix L61 are minimal, but devices may implement JTAG differently, if at all, introducing a wide variance. Locating JTAG test ports is only possible if the device hardware is designed with such support. With the ControlLogix L61, JTAG TAPs are successfully identified, enabling further hardware debugging. For configuration and memory acquisition, cost requirements include the hardware and software required for debugging with JTAG, which is significant. Time and complexity requirements for acquiring and analyzing memory images are also significant. Although the memory acquisition is straightforward, the analysis and reverse engineering of the acquired image is more time consuming and complex. Once connected to the device, the image acquisition and analysis technique takes advantage of hardware debugging capabilities and applies previously discussed reversing techniques on the newly acquired memory. An additional concern of hardware debugging determined with the ControlLogix L61 is the existence of watchdog timers that may be effected by halting the OS. Watchdog timers may effectively limit the abilities of a debugger. Overall, the use of hardware debugging tools facilitates the derivation of both checksum algorithms for the ControlLogix L61 and also provides access to otherwise inaccessible information such as the initialization and executive loading processes. While the use of hardware debugging is not strictly required to derive the ControlLogix L61 firmware update validation method, it remains an effective technique for doing so. In addition, the availability of hardware debugging tools is critical in recovering a bricked test device.

4.5 Summary

This section demonstrates the feasibility of a counterfeit firmware attack on a common PLC. This is accomplished by following a general process based on various reverse engineering techniques to derive the firmware update validation method. After confirming the correctness of the derived method, its design is analyzed for design weaknesses enabling the intentional modification and counterfeiting of the firmware. The applied reversing process is effective in deriving the validation method in this case. While the technique of disassembly analysis is limited by the complexity of the firmware binary, and the effectiveness of black box analysis is limited to detecting common validation methods, hardware debugging provides a significant advantage in the reversing process. The successful demonstration provides a realistic example of a counterfeit firmware update by exploiting a weakness in the design of the firmware update validation method.

V. Conclusions and Future Work

5.1 Conclusions

Industrial control systems are responsible for the management and automation of an ever increasing number of processes in national critical infrastructure. Embedded computing devices called PLCs offer immediate access to physical process elements in these applications and are critical in ensuring their proper operation. In the escalation of attacks on these devices, tools and techniques exhibit growing sophistication with an emphasis on subverting the system at incrementally lower levels. Hardware components of the devices representing the lowest level remain uniquely inaccessible to attackers without physical compromise at either the production or operational stages. Meanwhile, efforts focused on user programming at the highest level fail to adequately avert isolated detection, relying primarily on assistance from an insider threat or additional supervisory malware. The intermediate firmware level provides a compromise between accessibility, functionality, and autonomy while simultaneously exploiting the current deficiencies in detection methods.

This thesis determines the feasibility of firmware modification attacks on PLCs through the investigation and assessment of a common PLC to counterfeit firmware updates. A review of related works in the field of embedded devices reverse engineering provides various techniques that are integrated into a general reversing process to derive the firmware update validation method of PLCs. This process consists of four steps: (i) firmware acquisition, (ii) binary analysis of firmware, (iii) firmware disassembly, and (iv) derivation of the firmware update validation method. Step (iv) is presented as three approaches: (a) disassembly analysis, (b) black box analysis, and (c) hardware debugging analysis.

The reversing process is applied to a test environment, consisting of an Allen-Bradley ControlLogix L61. In this environment, the process is able to successfully derive the firmware update validation method. Firmware acquisition, binary analysis, and disassembly reveal general firmware information and prepare the firmware for step (iv). Disassembly analysis is a technique capable of deriving the firmware update validation method, but limited in effectiveness by the availability of descriptive disassembly, time available, and experience of the attacker. Black box analysis is potentially effective in reducing the search space of candidate algorithm classes, but a full brute force attack is not feasible. This technique is limited by time and the complexity of the firmware update validation method in use. Hardware debugging analysis provides the advantages of direct access to the device and augmenting disassembly analysis for successful derivation of the firmware update validation method of the test device. This approach is limited by the availability of hardware debugging support on the device and costs incurred by the special equipment required.

After deriving the firmware update validation method, the candidate algorithm is tested for correctness against available sample firmware images. The algorithm is then analyzed for design weaknesses that allow for intentionally modified firmware to pass validation. Such a weakness identified in the ControlLogix firmware is exploited and demonstrated with an example of counterfeit firmware. After spoofing the version number, an old firmware version is uploaded to the test system and successfully validated, reporting a new version number. This research confirms that firmware update validation methods in common use suffer from design weaknesses that facilitate firmware modification attack. Thus, this thesis verifies the feasibility of attacks targeting PLCs through the use of counterfeit firmware updates.

5.2 Significance

Unlike programming flaws such as a buffer overflow vulnerability that might allow for arbitrary code injection, the firmware update process is not manipulated or abused to achieve the same result. While a buffer overflow attack exploits a patchable vulnerability, firmware validation is a design level feature. The underlying software on the device intentionally inaccessible by the user, the executive loader, uses a modular summation algorithm to validate the firmware image. This design is intended to prevent the execution of corrupt firmware in the event of accidental modification. However, given the ability to upload firmware to the device, there exists no protection against intentional firmware modification.

This research demonstrates an example of a minor modification with a significant impact if used with malicious intent. By spoofing the version number, an older firmware version can be counterfeited to appear as a new version. Given known vulnerabilities in previous firmware versions, an attacker may exploit an old vulnerability in the firmware. As far as the operator and the control software for the device are aware, the version number reported by the PLC indicates that the new and secure firmware is correctly installed.

This research successfully verifies the feasibility of counterfeit firmware attacks on a common PLC. Considerations of the reversing process also provide necessary insight into the requirements and limitations of successfully counterfeiting firmware. This knowledge enables the development of defensive and forensic analysis techniques for firmware modification attacks. As a direct result of this research, realistic counterfeit firmware can be developed to assist researchers in detection and analysis techniques. Overall, this research demonstrates that, although the process and requirements to counterfeit firmware are not trivial and there exist limitations dependent on the device and the examiner, the threat posed to ICS security is credible and requires further attention.

5.3 Future Work

5.3.1 Direct Extensions.

As an extension to this work, the reversing process described may be considered as a basis for the vulnerability analysis of other PLC validation methods. Although the ControlLogix L61 used in the test environment is common, there remain numerous alternative controllers present in ICS applications. In order to develop an accurate awareness of general ICS vulnerabilities to this specific threat, further assessments should be applied to as many common PLCs as possible. It may also be possible to automate portions of the process to aid in this assessment. Indeed, the process described here may aide in the development of an automated assessment tool. In addition, the scope of the assessment should also be expanded to include external factors not considered in this work, such as the vulnerability of PLCs to remote counterfeit firmware updates. Such a vulnerability, if feasibly exploitable, presents an expanded attack surface for firmware attacks. Beyond the process itself, additional applications include the creation of counterfeit firmware samples. Through the process and information provided by this research, various samples of counterfeit firmware may be created to facilitate analysis and testing of realistic samples for the purposes of detection and forensic capabilities.

5.3.2 Preventative Measures.

Effective mitigation strategies for the threat of firmware attacks are reliant upon device manufacturers to follow secure design practices. The most apparent requirement for a secure firmware update method is a secure validation algorithm. A validation method able to detect intentional changes is a significant step towards more secure controllers. One such process is conceptualized in the form of digital signatures. By digitally signing legitimate firmware images, the vendor provides a means to validate that the firmware is authentic and unaltered. Digital signatures use asymmetric cryptography to create a digest, or signature, of the message (i.e., the firmware) using a private key held securely by the manufacturer

[34]. In this scheme, a controller presented with a firmware update uses the corresponding public key to validate that the signature is generated by the manufacturer's private key.

In order for any type of secure validation method to be effective, another requirement is that the manufacturer implement the method external to the OS firmware. The current implementation of the CRC algorithm in the ControlLogix emphasizes this point. Since the CRC algorithm is solely contained in the OS firmware, it is ineffective from a security perspective. Passing the modular summation validation present in the executive loader is sufficient to load and execute the firmware. Direct alteration of the firmware binary enables the removal of any validation functionality present in the OS.

In addition to implementing a secure validation method, alternative options should be considered for controlling access to the firmware. This may be implemented, for example, by requiring a valid serial number to access firmware update packages on vendor websites. This limits firmware access to those who possess a controller or have access to a valid serial number. Another approach is to obfuscate or encrypt firmware images. Although obscurity alone is not a sound solution to design security and such tactics create overhead in the executive loader, they may still inhibit attackers' ability to reverse engineer the firmware or implement meaningful modifications.

While the strategies proposed thus far have the potential to defend against firmware counterfeiting from a software perspective, hardware debugging tools provide low level access to the device enabling the manipulation of this software. Using a standard interface like JTAG, an attacker may obtain access to the executive loader and observe the details of any validation or obfuscation algorithms. Permanently disabling or locking out hardware debugging support for the controller after production should be considered.

Due to complexities arising from a redesign of the validation method, in addition to possible resistance by industry customers, adoption rates of direct design solutions may be low. For this reason, alternative solutions should also be explored in an effort to provide mitigation with comprehensive protection. Possible solutions include external validation tools capable of detecting counterfeit firmware. Such solutions may be implemented as either standalone or network level protection in a control system. As part of a comprehensive security plan, such detection and defense mechanisms may be integrated into an existing intrusion detection system (IDS) or intrusion prevention system (IPS). By monitoring network activity, such defensive measures may detect and possibly prevent unauthorized firmware updates to devices. Further research in these areas may provide solutions with adequate protection that are deployable in current environments.

5.3.3 Detection and Forensic Analysis.

The research and development of preemptive detection, firmware acquisition, and forensic analysis techniques for cases of counterfeit firmware is crucial to provide adequate protection for exposed critical infrastructure control systems. Detection methods are required to identify modified firmware in operational systems where cost and production considerations take priority over secure configurations, possibly leaving devices vulnerable to attack. Firmware acquisition techniques allow for both detection and analysis based on a complete or partial firmware image obtained directly from the device. Forensic analysis techniques applied to the acquired image facilitate assessment of its operational status and possible indicators of attack.

5.3.3.1 Indirect Methods.

Indirect methods like black-box testing and side-channel analysis may be used to infer the internal characteristics of a PLC for the purposes of detection and forensic analysis. Black-box testing involves systematic manipulation of device inputs while measuring outputs to infer information regarding the logic operating on the device. This may potentially enable the detection of malicious logic on the device with little to no knowledge of the underlying software. Black-box techniques can also be applied to a known clean device in order to create a fingerprint for future comparisons. Common metrics applied to black-box testing include the system's conformance to specifications, error recovery ability, defined security responses, performance, and configurability [31].

The goal of side-channel analysis is to infer information about the system based on measurements of external factors referred to as the side channels. By measuring a side channel, it may be possible to infer operational conditions inside the system. Some possible side channels worth considering include power [30], timing [29], temperature, and electromagnetic emanations [3]. Black-box and side-channel analysis may potentially facilitate detection of firmware deviations while minimizing direct impacts to the device.

5.3.3.2 Direct Methods.

Direct detection and analysis techniques fall into two categories: software-based and hardware-based. Software backdoors in a system may provide an attacker control over the system by taking advantage of access mechanisms intended to provide low-level access to system developers. Such features left over from development or improperly secured may allow read access to firmware memory. Recent research presenting hard coded passwords left in the device firmware by developers are an example [9, 51, 60]. In addition to built in functionality, the exploitation of a vulnerability on the device may also allow access to the firmware in memory. As demonstrated on the iPhone [22], an exploit that gains execution control of the device may allow the injection of instructions to output the contents of memory containing firmware. Software-based methods are potentially advantageous since a firmware image may be obtained remotely without physical access to the device. It may even be possible to obtain the image from a live system to minimize or avoid any operational interruptions. Potential disadvantages of these methods include their inherent interaction with the device, which may adversely affect forensic fidelity or completeness.

Hardware-based methods rely largely on the techniques presented in this research. A hardware debugging interface to the target device through a standard such as JTAG allows for direct memory reads to dump the firmware. Since JTAG is an optional standard, not every device necessarily supports it. In the worst case, there may be no hardware debugging standard implemented at all. However, the processes described here may be used to obtain a firmware image if hardware debugging is supported. Barring the success of an image capture using JTAG, an alternative is to perform independent chip analysis. This method involves physically removing the flash chip containing the firmware from the circuit board in order to read it directly [11]. These hardware-based methods offer the potential benefit of providing a complete image of the memory while maintaining a high level of forensic integrity. One potential drawback, however, is their dependency on physical access to the system. In the case of independent chip analysis, the system must be dismantled, possibly resulting in permanent destruction.

5.4 Summary

This research demonstrates that counterfeit firmware attacks on a common PLC are feasible. This vulnerability is a result of insecure design of the firmware update validation method. The described process provides insight into the advantages and limitations of particular attack techniques, and the counterfeit firmware provides a realistic example for use in future research. Other common PLCs should also be assessed for similar vulnerabilities to firmware modification. Preventative measures such as securely designed validation methods or external validation mechanisms must be considered and implemented. Finally, future work must also include the development of detection and forensic analysis capabilities, possibly based on black box and side channel analysis or the direct analysis of firmware on the device.

Appendix A: ControlLogix Firmware Operation Flowcharts

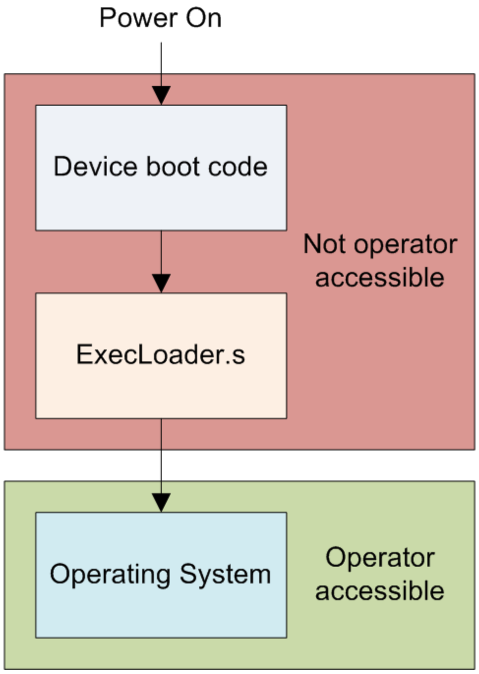


Figure A.1: Overview of ControlLogix L61 operation.

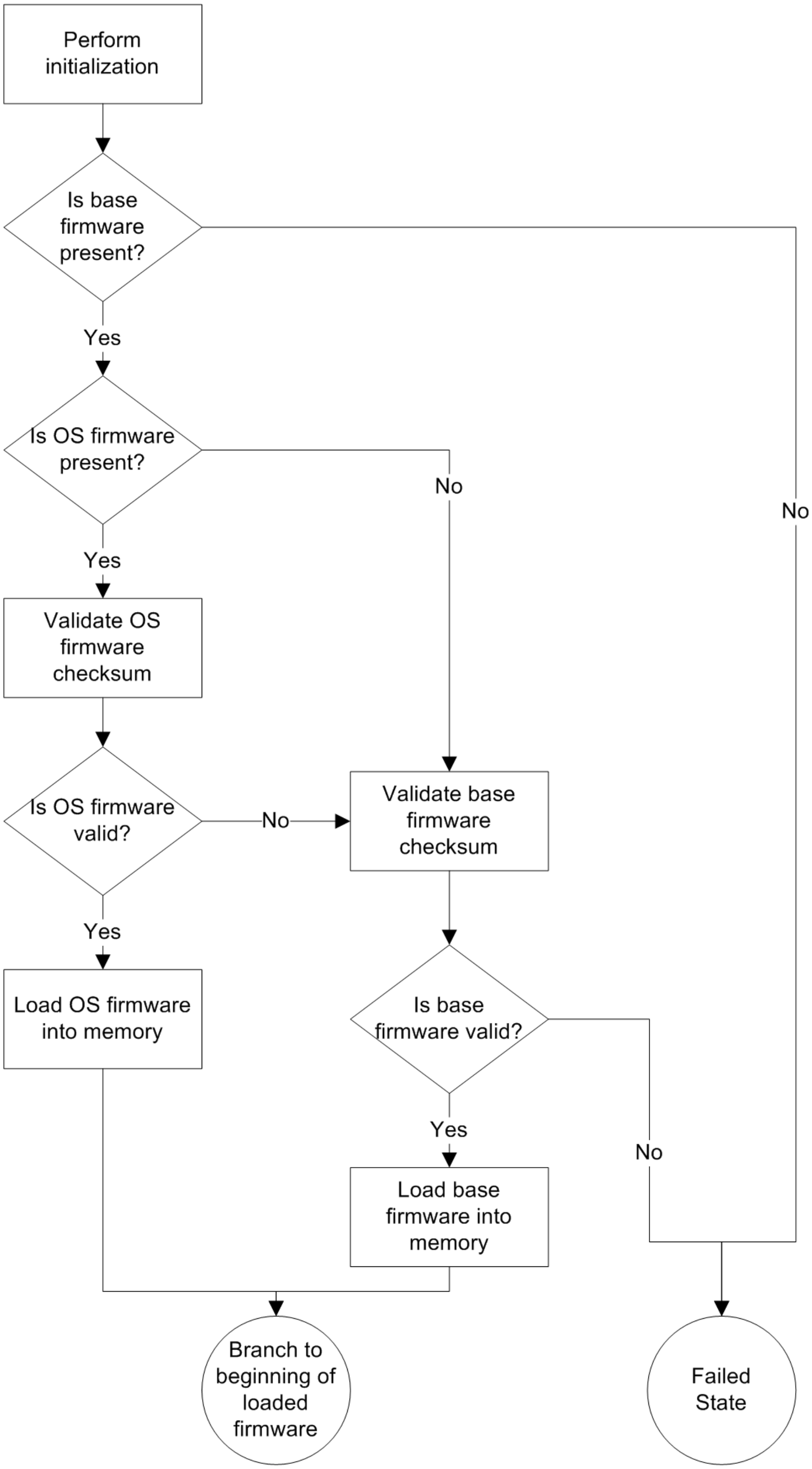

Perform initialization
Is base firmware present?
Yes
Is OS firmware present?
No
No
Yes
Validate OS firmware checksum
Is OS firmware valid?
No
Validate base firmware checksum
Yes
Load OS firmware into memory
Is base firmware valid?
Yes
No
Load base firmware into memory
Branch to beginning of loaded firmware
Failed State

Figure A.2: Flow chart of ExecLoader.s.

Appendix B: Contents of Firmware Update Package

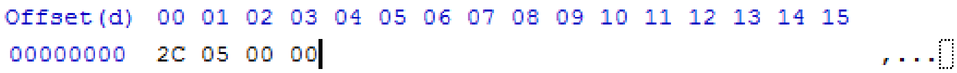
B.1 PN-86270.nvs

```
[Device]
NewRevision = 19.11.16
DialogNewRevision = 19.11.16
NumberUpdates = 1
ConnectionType = UNCONNECTED
NumberIdentities = 87

            Vendor  Product  Product   Major     Minor    Catalog    HW Major  HW Minor
              Id     Type     Code    Revision  Revision  Revisions  Revision  Revision

Identity1   =   1,    0x0e,   0x36,        1,      0,     1756-L61,      1,      0
Identity2   =   1,    0x0e,   0x37,        1,      0,     1756-L62,      1,      0
Identity3   =   1,    0x0e,   0x38,        1,      0,     1756-L63,      1,      0
Identity4   =   1,    0x0e,   0x38,       10,      0,     1756-L63,      1,      0
Identity5   =   1,    0x0e,   0x38,       11,      0,     1756-L63,      1,      0
Identity6   =   1,    0x0e,   0x36,       12,      0,     1756-L61,      1,      0

...[lines removed for brevity]...

Identity79  =   1,    0x0e,   0x36,       19,      0,     1756-L61,      1,      0
Identity80  =   1,    0x0e,   0x37,       19,      0,     1756-L62,      1,      0
Identity81  =   1,    0x0e,   0x38,       19,      0,     1756-L63,      1,      0
Identity82  =   1,    0x0e,   0x36,       19,      0,     1756-L61,      2,      0
Identity83  =   1,    0x0e,   0x37,       19,      0,     1756-L62,      2,      0
Identity84  =   1,    0x0e,   0x38,       19,      0,     1756-L63,      2,      0
Identity85  =   1,    0x0e,   0x36,       19,      0,     1756-L61,      3,      0
Identity86  =   1,    0x0e,   0x37,       19,      0,     1756-L62,      3,      0
Identity87  =   1,    0x0e,   0x38,       19,      0,     1756-L63,      3,      0


[Update1]
NVSInstance = 3
MajorRevision = 19
MinorRevision = 11
MaxTimeoutSeconds = 60
StartingLocation = 0xb160000
FileSize = 2546132
DataFileName = PN-86272.bin
UpdateReset = 1
AutoResetOnError = 0
FirstTransferDelay = 0
ErrorInstructions = Manually Reset module

[About Info]

/******************** COPYRIGHT AND LICENCE NOTICE ********************

"Contains BIGDIGITS multiple-precision arithmetic code originally
written by David Ireland, copyright (c) 2001-5 by D.I. Management
Services Pty Limited <www.di-mgt.com.au>, and is used with
permission."

*************** END OF COPYRIGHT AND LICENCE NOTICE ******************/
```

B.2 CONTENTS.TXT

```
ControlFlash Firmware Upgrade Kit Contents
Created: 09/27/10 08:24:01

Catalog Number    Revision    Script Filename
1756-L61          19.11.16    C:\FIRMWARE\1756-L6X\V19\19.11\PN-86270.nvs
1756-L62          19.11.16    C:\FIRMWARE\1756-L6X\V19\19.11\PN-86270.nvs
1756-L63          19.11.16    C:\FIRMWARE\1756-L6X\V19\19.11\PN-86270.nvs
```

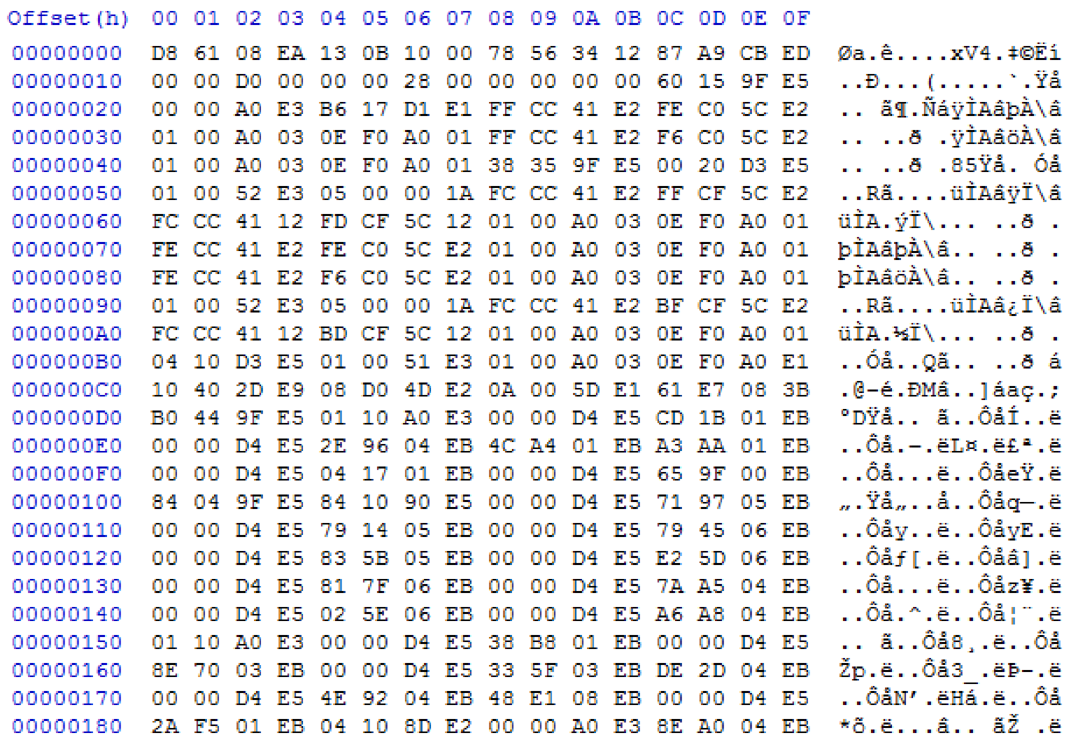
B.3 PN-86270.RES



```
Offset(d) 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
00000000  2C 05 00 00                                      ,...
```

Figure B.1: Contents of PN-86270.RES in HxD.

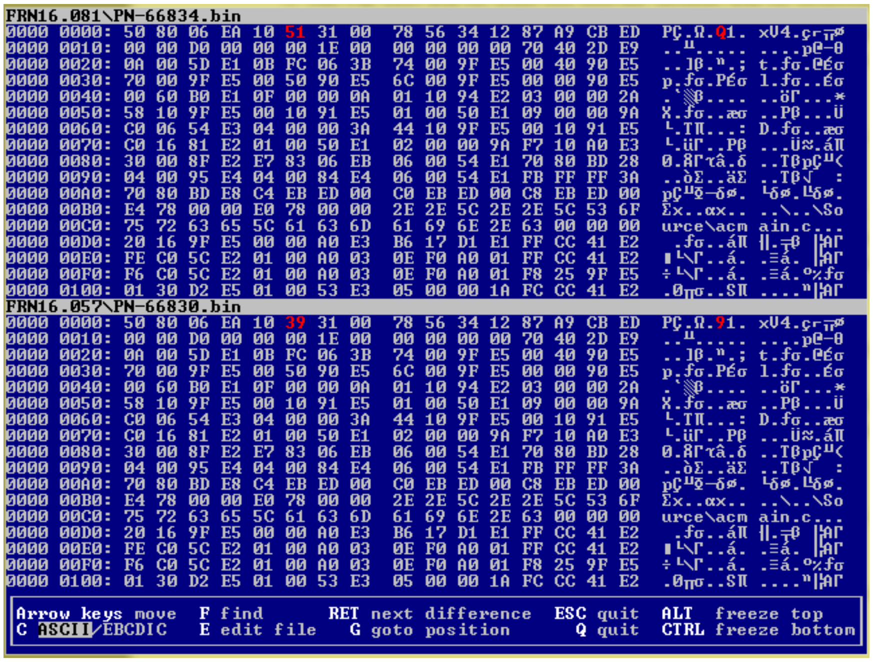
B.4 PN-86272.bin



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  D8 61 08 EA 13 0B 10 00 78 56 34 12 87 A9 CB ED  Øa.ê....xV4.‡©Ëí
00000010  00 00 D0 00 00 00 28 00 00 00 00 00 60 15 9F E5  ..Ð...(.....`.Ÿå
00000020  00 00 A0 E3 B6 17 D1 E1 FF CC 41 E2 FE C0 5C E2  .. ã¶.ÑáÿÌAâþÀ\â
00000030  01 00 A0 03 0E F0 A0 01 FF CC 41 E2 F6 C0 5C E2  .. ..ð .ÿÌAâöÀ\â
00000040  01 00 A0 03 0E F0 A0 01 38 35 9F E5 00 20 D3 E5  .. ..ð .85Ÿå. Óå
00000050  01 00 52 E3 05 00 00 1A FC CC 41 E2 FF CF 5C E2  ..Rã....üÌAâÿÏ\â
00000060  FC CC 41 12 FD CF 5C 12 01 00 A0 03 0E F0 A0 01  üÌA.ýÏ\... ..ð .
00000070  FE CC 41 E2 FE C0 5C E2 01 00 A0 03 0E F0 A0 01  þÌAâþÀ\â.. ..ð .
00000080  FE CC 41 E2 F6 C0 5C E2 01 00 A0 03 0E F0 A0 01  þÌAâöÀ\â.. ..ð .
00000090  01 00 52 E3 05 00 00 1A FC CC 41 E2 BF CF 5C E2  ..Rã....üÌAâ¿Ï\â
000000A0  FC CC 41 12 BD CF 5C 12 01 00 A0 03 0E F0 A0 01  üÌA.½Ï\... ..ð .
000000B0  04 10 D3 E5 01 00 51 E3 01 00 A0 03 0E F0 A0 E1  ..Óå..Qã.. ..ð á
000000C0  10 40 2D E9 08 D0 4D E2 0A 00 5D E1 61 E7 08 3B  .@-é.ÐMâ..]áaç.;
000000D0  B0 44 9F E5 01 10 A0 E3 00 00 D4 E5 CD 1B 01 EB  °DŸå.. ã..ÔåÍ..ë
000000E0  00 00 D4 E5 2E 96 04 EB 4C A4 01 EB A3 AA 01 EB  ..Ôå.–.ëL¤.ë£ª.ë
000000F0  00 00 D4 E5 04 17 01 EB 00 00 D4 E5 65 9F 00 EB  ..Ôå...ë..ÔåeŸ.ë
00000100  84 04 9F E5 84 10 90 E5 00 00 D4 E5 71 97 05 EB  „.Ÿå„..å..Ôåq—.ë
00000110  00 00 D4 E5 79 14 05 EB 00 00 D4 E5 79 45 06 EB  ..Ôåy..ë..ÔåyE.ë
00000120  00 00 D4 E5 83 5B 05 EB 00 00 D4 E5 E2 5D 06 EB  ..Ôåƒ[.ë..Ôåâ].ë
00000130  00 00 D4 E5 81 7F 06 EB 00 00 D4 E5 7A A5 04 EB  ..Ôå...ë..Ôåz¥.ë
00000140  00 00 D4 E5 02 5E 06 EB 00 00 D4 E5 A6 A8 04 EB  ..Ôå.^.ë..Ôå¦¨.ë
00000150  01 10 A0 E3 00 00 D4 E5 38 B8 01 EB 00 00 D4 E5  .. ã..Ôå8¸.ë..Ôå
00000160  8E 70 03 EB 00 00 D4 E5 33 5F 03 EB DE 2D 04 EB  Žp.ë..Ôå3_.ëÞ-.ë
00000170  00 00 D4 E5 4E 92 04 EB 48 E1 08 EB 00 00 D4 E5  ..ÔåN’.ëHá.ë..Ôå
00000180  2A F5 01 EB 04 10 8D E2 00 00 A0 E3 8E A0 04 EB  *õ.ë...â.. ãŽ .ë
```

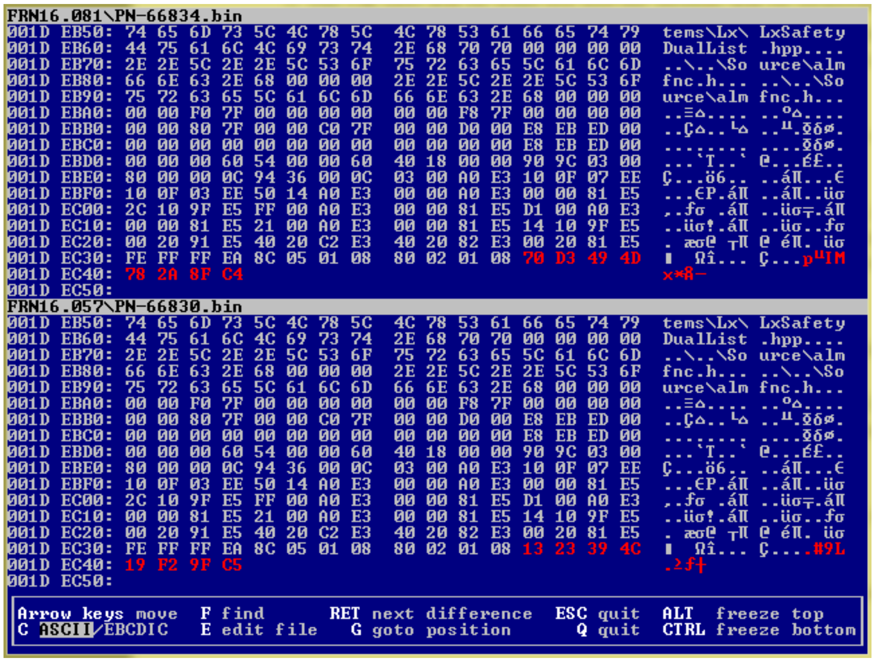
Figure B.2: Beginning of FRN19.011 binary in HxD.

Appendix C: VBinDiff Examples



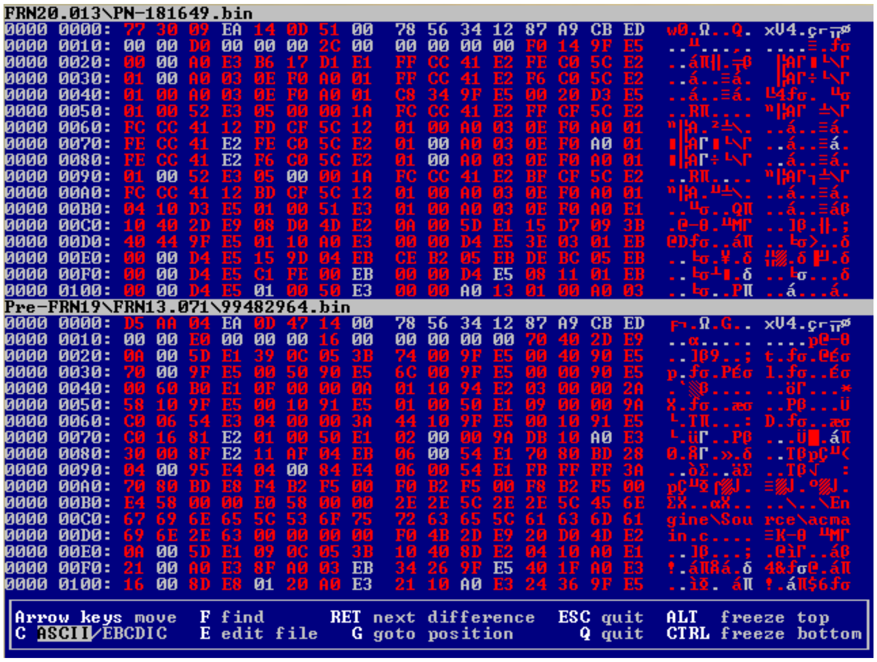
```
FRN16.081\PN-66834.bin
0000 0000: 50 80 06 EA 10 51 31 00   78 56 34 12 87 A9 CB ED   PÇ.Ω.Q1. xV4.ç⌐╦φ
0000 0010: 00 00 D0 00 00 00 1E 00   00 00 00 00 70 40 2D E9   ..╨..... ....p@-Θ
0000 0020: 0A 00 5D E1 0B FC 06 3B   74 00 9F E5 00 40 90 E5   ..]ß.ⁿ.; t.ƒσ.@Éσ
0000 0030: 70 00 9F E5 00 50 90 E5   6C 00 9F E5 00 00 90 E5   p.ƒσ.PÉσ l.ƒσ..Éσ
0000 0040: 00 60 B0 E1 0F 00 00 0A   01 10 94 E2 03 00 00 2A   .`░ß.... ..öΓ...*
0000 0050: 58 10 9F E5 00 10 91 E5   01 00 50 E1 09 00 00 9A   X.ƒσ..æσ ..Pß...Ü
0000 0060: C0 06 54 E3 04 00 00 3A   44 10 9F E5 00 10 91 E5   └.Tπ...: D.ƒσ..æσ
0000 0070: C0 16 81 E2 01 00 50 E1   02 00 00 9A F7 10 A0 E3   └.üΓ..Pß ...Ü≈.áπ
0000 0080: 30 00 8F E2 E7 83 06 EB   06 00 54 E1 70 80 BD 28   0.ÅΓτâ.δ ..TßpÇ╜(
0000 0090: 04 00 95 E4 04 00 84 E4   06 00 54 E1 FB FF FF 3A   ..òΣ..äΣ ..Tß√  :
0000 00A0: 70 80 BD E8 C4 EB ED 00   C0 EB ED 00 C8 EB ED 00   pÇ╜Φ─δφ. └δφ.╚δφ.
0000 00B0: E4 78 00 00 E0 78 00 00   2E 2E 5C 2E 2E 5C 53 6F   Σx..αx.. ..\..\So
0000 00C0: 75 72 63 65 5C 61 63 6D   61 69 6E 2E 63 00 00 00   urce\acm ain.c...
0000 00D0: 20 16 9F E5 00 00 A0 E3   B6 17 D1 E1 FF CC 41 E2    .ƒσ..áπ ╢.╤ß ╠AΓ
0000 00E0: FE C0 5C E2 01 00 A0 03   0E F0 A0 01 FF CC 41 E2   ■└\Γ..á. .≡á. ╠AΓ
0000 00F0: F6 C0 5C E2 01 00 A0 03   0E F0 A0 01 F8 25 9F E5   ÷└\Γ..á. .≡á.°%ƒσ
0000 0100: 01 30 D2 E5 01 00 53 E3   05 00 00 1A FC CC 41 E2   .0╥σ..Sπ ....ⁿ╠AΓ
FRN16.057\PN-66830.bin
0000 0000: 50 80 06 EA 10 39 31 00   78 56 34 12 87 A9 CB ED   PÇ.Ω.91. xV4.ç⌐╦φ
0000 0010: 00 00 D0 00 00 00 1E 00   00 00 00 00 70 40 2D E9   ..╨..... ....p@-Θ
0000 0020: 0A 00 5D E1 0B FC 06 3B   74 00 9F E5 00 40 90 E5   ..]ß.ⁿ.; t.ƒσ.@Éσ
0000 0030: 70 00 9F E5 00 50 90 E5   6C 00 9F E5 00 00 90 E5   p.ƒσ.PÉσ l.ƒσ..Éσ
0000 0040: 00 60 B0 E1 0F 00 00 0A   01 10 94 E2 03 00 00 2A   .`░ß.... ..öΓ...*
0000 0050: 58 10 9F E5 00 10 91 E5   01 00 50 E1 09 00 00 9A   X.ƒσ..æσ ..Pß...Ü
0000 0060: C0 06 54 E3 04 00 00 3A   44 10 9F E5 00 10 91 E5   └.Tπ...: D.ƒσ..æσ
0000 0070: C0 16 81 E2 01 00 50 E1   02 00 00 9A F7 10 A0 E3   └.üΓ..Pß ...Ü≈.áπ
0000 0080: 30 00 8F E2 E7 83 06 EB   06 00 54 E1 70 80 BD 28   0.ÅΓτâ.δ ..TßpÇ╜(
0000 0090: 04 00 95 E4 04 00 84 E4   06 00 54 E1 FB FF FF 3A   ..òΣ..äΣ ..Tß√  :
0000 00A0: 70 80 BD E8 C4 EB ED 00   C0 EB ED 00 C8 EB ED 00   pÇ╜Φ─δφ. └δφ.╚δφ.
0000 00B0: E4 78 00 00 E0 78 00 00   2E 2E 5C 2E 2E 5C 53 6F   Σx..αx.. ..\..\So
0000 00C0: 75 72 63 65 5C 61 63 6D   61 69 6E 2E 63 00 00 00   urce\acm ain.c...
0000 00D0: 20 16 9F E5 00 00 A0 E3   B6 17 D1 E1 FF CC 41 E2    .ƒσ..áπ ╢.╤ß ╠AΓ
0000 00E0: FE C0 5C E2 01 00 A0 03   0E F0 A0 01 FF CC 41 E2   ■└\Γ..á. .≡á. ╠AΓ
0000 00F0: F6 C0 5C E2 01 00 A0 03   0E F0 A0 01 F8 25 9F E5   ÷└\Γ..á. .≡á.°%ƒσ
0000 0100: 01 30 D2 E5 01 00 53 E3   05 00 00 1A FC CC 41 E2   .0╥σ..Sπ ....ⁿ╠AΓ

 Arrow keys move   F find      RET next difference   ESC quit   ALT  freeze top
 C ASCII/EBCDIC    E edit file   G goto position       Q quit   CTRL freeze bottom
```

Figure C.1: VBinDiff of FRN16.081 and FRN16.057 beginning.



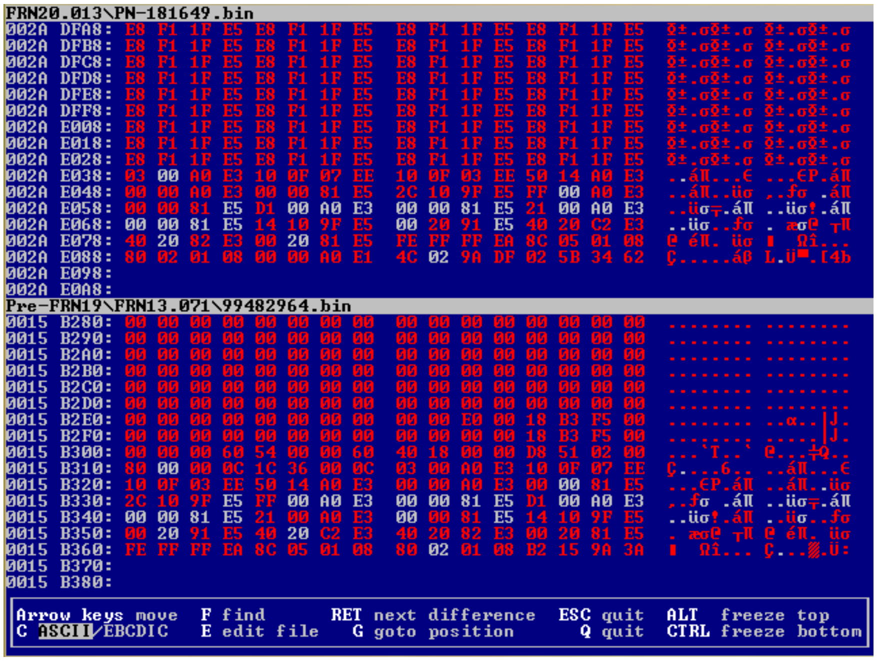
```
FRN16.081\PN-66834.bin
001D EB50: 74 65 6D 73 5C 4C 78 5C   4C 78 53 61 66 65 74 79   tems\Lx\ LxSafety
001D EB60: 44 75 61 6C 4C 69 73 74   2E 68 70 70 00 00 00 00   DualList .hpp....
001D EB70: 2E 2E 5C 2E 2E 5C 53 6F   75 72 63 65 5C 61 6C 6D   ..\..\So urce\alm
001D EB80: 66 6E 63 2E 68 00 00 00   2E 2E 5C 2E 2E 5C 53 6F   fnc.h... ..\..\So
001D EB90: 75 72 63 65 5C 61 6C 6D   66 6E 63 2E 68 00 00 00   urce\alm fnc.h...
001D EBA0: 00 00 F0 7F 00 00 00 00   00 00 F8 7F 00 00 00 00   ..≡⌂.... ..°⌂....
001D EBB0: 00 00 80 7F 00 00 C0 7F   00 00 D0 00 E8 EB ED 00   ..Ç⌂..└⌂ ..╨.Φδφ.
001D EBC0: 00 00 00 00 00 00 00 00   00 00 00 00 E8 EB ED 00   ........ ....Φδφ.
001D EBD0: 00 00 00 60 54 00 00 60   40 18 00 00 90 9C 03 00   ...`T..` @...É£..
001D EBE0: 80 00 00 0C 94 36 00 0C   03 00 A0 E3 10 0F 07 EE   Ç...ö6.. ..áπ...ε
001D EBF0: 10 0F 03 EE 50 14 A0 E3   00 00 A0 E3 00 00 81 E5   ...εP.áπ ..áπ..üσ
001D EC00: 2C 10 9F E5 FF 00 A0 E3   00 00 81 E5 D1 00 A0 E3   ,.ƒσ .áπ ..üσ╤.áπ
001D EC10: 00 00 81 E5 21 00 A0 E3   00 00 81 E5 14 10 9F E5   ..üσ!.áπ ..üσ..ƒσ
001D EC20: 00 20 91 E5 40 20 C2 E3   40 20 82 E3 00 20 81 E5   . æσ@ ┬π @ éπ. üσ
001D EC30: FE FF FF EA 8C 05 01 08   80 02 01 08 70 D3 49 4D   ■  Ωî... Ç...p╙IM
001D EC40: 78 2A 8F C4                                        x*Å─
001D EC50:
FRN16.057\PN-66830.bin
001D EB50: 74 65 6D 73 5C 4C 78 5C   4C 78 53 61 66 65 74 79   tems\Lx\ LxSafety
001D EB60: 44 75 61 6C 4C 69 73 74   2E 68 70 70 00 00 00 00   DualList .hpp....
001D EB70: 2E 2E 5C 2E 2E 5C 53 6F   75 72 63 65 5C 61 6C 6D   ..\..\So urce\alm
001D EB80: 66 6E 63 2E 68 00 00 00   2E 2E 5C 2E 2E 5C 53 6F   fnc.h... ..\..\So
001D EB90: 75 72 63 65 5C 61 6C 6D   66 6E 63 2E 68 00 00 00   urce\alm fnc.h...
001D EBA0: 00 00 F0 7F 00 00 00 00   00 00 F8 7F 00 00 00 00   ..≡⌂.... ..°⌂....
001D EBB0: 00 00 80 7F 00 00 C0 7F   00 00 D0 00 E8 EB ED 00   ..Ç⌂..└⌂ ..╨.Φδφ.
001D EBC0: 00 00 00 00 00 00 00 00   00 00 00 00 E8 EB ED 00   ........ ....Φδφ.
001D EBD0: 00 00 00 60 54 00 00 60   40 18 00 00 90 9C 03 00   ...`T..` @...É£..
001D EBE0: 80 00 00 0C 94 36 00 0C   03 00 A0 E3 10 0F 07 EE   Ç...ö6.. ..áπ...ε
001D EBF0: 10 0F 03 EE 50 14 A0 E3   00 00 A0 E3 00 00 81 E5   ...εP.áπ ..áπ..üσ
001D EC00: 2C 10 9F E5 FF 00 A0 E3   00 00 81 E5 D1 00 A0 E3   ,.ƒσ .áπ ..üσ╤.áπ
001D EC10: 00 00 81 E5 21 00 A0 E3   00 00 81 E5 14 10 9F E5   ..üσ!.áπ ..üσ..ƒσ
001D EC20: 00 20 91 E5 40 20 C2 E3   40 20 82 E3 00 20 81 E5   . æσ@ ┬π @ éπ. üσ
001D EC30: FE FF FF EA 8C 05 01 08   80 02 01 08 13 23 39 4C   ■  Ωî... Ç....#9L
001D EC40: 19 F2 9F C5                                        .≥ƒ┼
001D EC50:

 Arrow keys move   F find      RET next difference   ESC quit   ALT  freeze top
 C ASCII/EBCDIC    E edit file   G goto position       Q quit   CTRL freeze bottom
```

Figure C.2: VBinDiff of FRN16.081 and FRN16.057 end.



```
FRN20.013\PN-181649.bin
0000 0000: 77 30 09 EA 14 0D 51 00  78 56 34 12 87 A9 CB ED  w0.Ω..Q. xV4.ç⌐╦φ
0000 0010: 00 00 D0 00 00 00 2C 00  00 00 00 00 F0 14 9F E5  ..╨...,. ....≡.ƒσ
0000 0020: 00 00 A0 E3 B6 17 D1 E1  FF CC 41 E2 FE C0 5C E2  ..áπ╢.╤ß  ╠AΓ■└\Γ
0000 0030: 01 00 A0 03 0E F0 A0 01  FF CC 41 E2 F6 C0 5C E2  ..á..≡á.  ╠AΓ÷└\Γ
0000 0040: 01 00 A0 03 0E F0 A0 01  C8 34 9F E5 00 20 D3 E5  ..á..≡á. ╚4ƒσ. ╙σ
0000 0050: 01 00 52 E3 05 00 00 1A  FC CC 41 E2 FF CF 5C E2  ..Rπ.... ⁿ╠AΓ ╧\Γ
0000 0060: FC CC 41 12 FD CF 5C 12  01 00 A0 03 0E F0 A0 01  ⁿ╠A.²╧\. ..á..≡á.
0000 0070: FE CC 41 E2 FE C0 5C E2  01 00 A0 03 0E F0 A0 01  ■╠AΓ■└\Γ ..á..≡á.
0000 0080: FE CC 41 E2 F6 C0 5C E2  01 00 A0 03 0E F0 A0 01  ■╠AΓ÷└\Γ ..á..≡á.
0000 0090: 01 00 52 E3 05 00 00 1A  FC CC 41 E2 BF CF 5C E2  ..Rπ.... ⁿ╠AΓ┐╧\Γ
0000 00A0: FC CC 41 12 BD CF 5C 12  01 00 A0 03 0E F0 A0 01  ⁿ╠A.╜╧\. ..á..≡á.
0000 00B0: 04 10 D3 E5 01 00 51 E3  01 00 A0 03 0E F0 A0 E1  ..╙σ..Qπ ..á..≡áß
0000 00C0: 10 40 2D E9 08 D0 4D E2  0A 00 5D E1 15 D7 09 3B  .@-Θ.╨MΓ ..]ß.╫.;
0000 00D0: 40 44 9F E5 01 10 A0 E3  00 00 D4 E5 3E 03 01 EB  @Dƒσ..áπ ..╘σ>..δ
0000 00E0: 00 00 D4 E5 15 9D 04 EB  CE B2 05 EB DE BC 05 EB  ..╘σ.¥.δ ╬▓.δ▐╝.δ
0000 00F0: 00 00 D4 E5 C1 FE 00 EB  00 00 D4 E5 08 11 01 EB  ..╘σ┴■.δ ..╘σ...δ
0000 0100: 00 00 D4 E5 01 00 50 E3  00 00 A0 13 01 00 A0 03  ..╘σ..Pπ ..á...á.
Pre-FRN19\FRN13.071\99482964.bin
0000 0000: D5 AA 04 EA 0D 47 14 00  78 56 34 12 87 A9 CB ED  ╒¬.Ω.G.. xV4.ç⌐╦φ
0000 0010: 00 00 E0 00 00 00 16 00  00 00 00 00 70 40 2D E9  ..α..... ....p@-Θ
0000 0020: 0A 00 5D E1 39 0C 05 3B  74 00 9F E5 00 40 90 E5  ..]ß9..; t.ƒσ.@Éσ
0000 0030: 70 00 9F E5 00 50 90 E5  6C 00 9F E5 00 00 90 E5  p.ƒσ.PÉσ l.ƒσ..Éσ
0000 0040: 00 60 B0 E1 0F 00 00 0A  01 10 94 E2 03 00 00 2A  .`░ß.... ..öΓ...*
0000 0050: 58 10 9F E5 00 10 91 E5  01 00 50 E1 09 00 00 9A  X.ƒσ..æσ ..Pß...Ü
0000 0060: C0 06 54 E3 04 00 00 3A  44 10 9F E5 00 10 91 E5  └.Tπ...: D.ƒσ..æσ
0000 0070: C0 16 81 E2 01 00 50 E1  02 00 00 9A DB 10 A0 E3  └.üΓ..Pß ...Ü█.áπ
0000 0080: 30 00 8F E2 11 AF 04 EB  06 00 54 E1 70 80 BD 28  0.ÅΓ.».δ ..TßpÇ╜(
0000 0090: 04 00 95 E4 04 00 84 E4  06 00 54 E1 FB FF FF 3A  ..òΣ..äΣ ..Tß√  :
0000 00A0: 70 80 BD E8 F4 B2 F5 00  F0 B2 F5 00 F8 B2 F5 00  pÇ╜Φ⌠▓⌡. ≡▓⌡.°▓⌡.
0000 00B0: E4 58 00 00 E0 58 00 00  2E 2E 5C 2E 2E 5C 45 6E  ΣX..αX.. ..\..\En
0000 00C0: 67 69 6E 65 5C 53 6F 75  72 63 65 5C 61 63 6D 61  gine\Sou rce\acma
0000 00D0: 69 6E 2E 63 00 00 00 00  F0 4B 2D E9 20 D0 4D E2  in.c.... ≡K-Θ ╨MΓ
0000 00E0: 0A 00 5D E1 09 0C 05 3B  10 40 8D E2 04 10 A0 E1  ..]ß...; .@ìΓ..áß
0000 00F0: 21 00 A0 E3 8F A0 03 EB  34 26 9F E5 40 1F A0 E3  !.áπÅá.δ 4&ƒσ@.áπ
0000 0100: 16 00 8D E8 01 20 A0 E3  21 10 A0 E3 24 36 9F E5  ..ìΦ. áπ !.áπ$6ƒσ

 Arrow keys move   F find       RET next difference   ESC quit   ALT  freeze top
 C ASCII/EBCDIC    E edit file    G goto position       Q quit   CTRL freeze bottom
```

Figure C.3: VBinDiff of FRN20.013 and FRN13.071 beginning.



```
FRN20.013\PN-181649.bin
002A DFA8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A DFB8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A DFC8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A DFD8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A DFE8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A DFF8: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A E008: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A E018: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A E028: E8 F1 1F E5 E8 F1 1F E5  E8 F1 1F E5 E8 F1 1F E5  Φ±.σΦ±.σ Φ±.σΦ±.σ
002A E038: 03 00 A0 E3 10 0F 07 EE  10 0F 03 EE 50 14 A0 E3  ..áπ...ε ...εP.áπ
002A E048: 00 00 A0 E3 00 00 81 E5  2C 10 9F E5 FF 00 A0 E3  ..áπ..üσ ,.ƒσ .áπ
002A E058: 00 00 81 E5 D1 00 A0 E3  00 00 81 E5 21 00 A0 E3  ..üσ╤.áπ ..üσ!.áπ
002A E068: 00 00 81 E5 14 10 9F E5  00 20 91 E5 40 20 C2 E3  ..üσ..ƒσ . æσ@ ┬π
002A E078: 40 20 82 E3 00 20 81 E5  FE FF FF EA 8C 05 01 08  @ éπ. üσ ■  Ωî...
002A E088: 80 02 01 08 00 00 A0 E1  4C 02 9A DF 02 5B 34 62  Ç.....áß L.Ü▀.[4b
002A E098:
002A E0A8:
Pre-FRN19\FRN13.071\99482964.bin
0015 B280: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B290: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B2A0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B2B0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B2C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B2D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0015 B2E0: 00 00 00 00 00 00 00 00  00 00 E0 00 18 B3 F5 00  ........ ..α..│⌡.
0015 B2F0: 00 00 00 00 00 00 00 00  00 00 00 00 18 B3 F5 00  ........ .....│⌡.
0015 B300: 00 00 00 60 54 00 00 60  40 18 00 00 D8 51 02 00  ...`T..` @...╪Q..
0015 B310: 80 00 00 0C 1C 36 00 0C  03 00 A0 E3 10 0F 07 EE  Ç....6.. ..áπ...ε
0015 B320: 10 0F 03 EE 50 14 A0 E3  00 00 A0 E3 00 00 81 E5  ...εP.áπ ..áπ..üσ
0015 B330: 2C 10 9F E5 FF 00 A0 E3  00 00 81 E5 D1 00 A0 E3  ,.ƒσ .áπ ..üσ╤.áπ
0015 B340: 00 00 81 E5 21 00 A0 E3  00 00 81 E5 14 10 9F E5  ..üσ!.áπ ..üσ..ƒσ
0015 B350: 00 20 91 E5 40 20 C2 E3  40 20 82 E3 00 20 81 E5  . æσ@ ┬π @ éπ. üσ
0015 B360: FE FF FF EA 8C 05 01 08  80 02 01 08 B2 15 9A 3A  ■  Ωî... Ç...▓.Ü:
0015 B370:
0015 B380:

 Arrow keys move   F find       RET next difference   ESC quit   ALT  freeze top
 C ASCII/EBCDIC    E edit file    G goto position       Q quit   CTRL freeze bottom
```

Figure C.4: Length-corrected VBinDiff of FRN20.013 and FRN13.071 end.

Appendix D: Physical Component Analysis

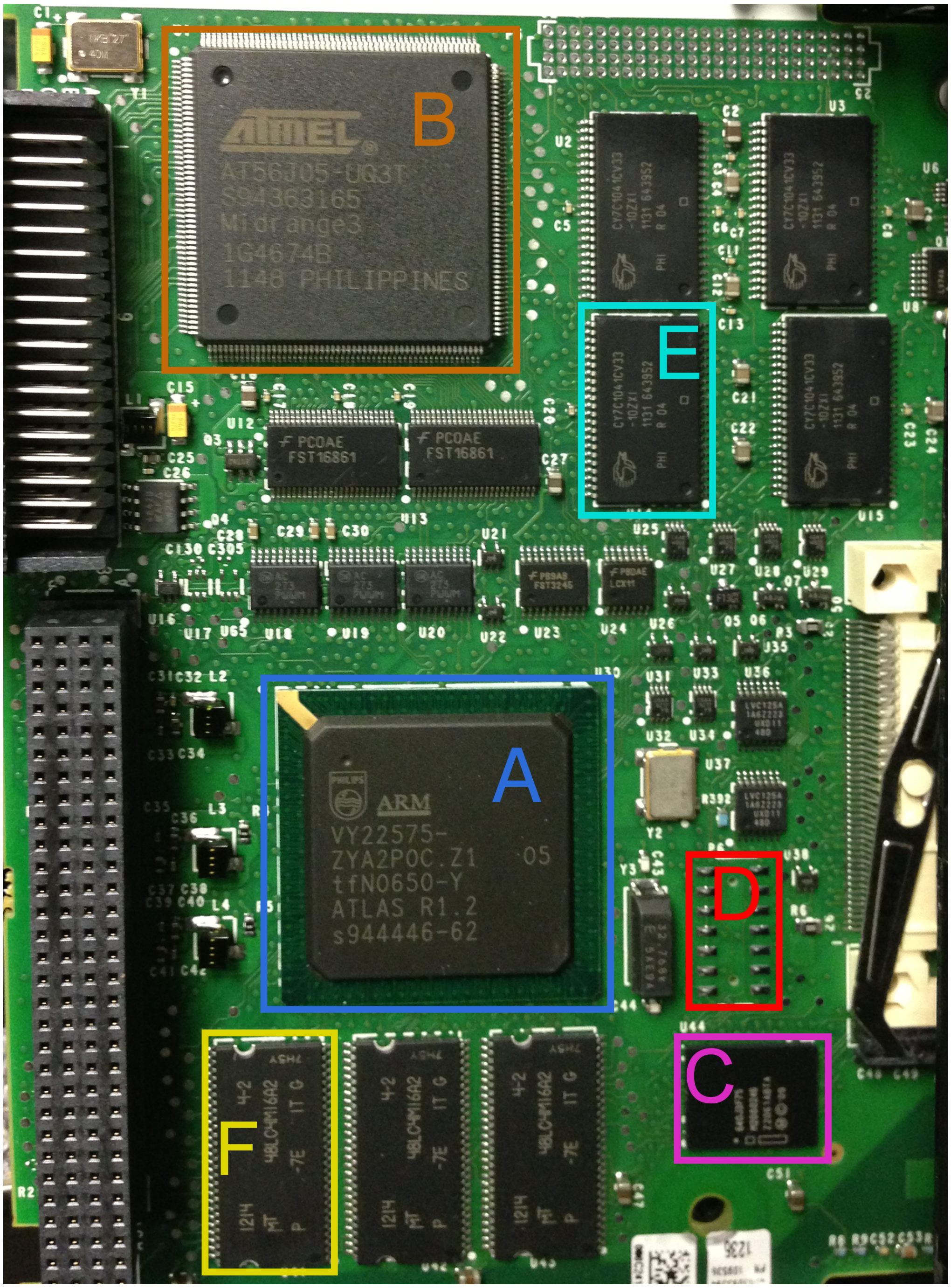


Figure D.1: Circuit board of the 1756-L61/B.

D.1 Components of Interest

A. Central Processing Unit

Manufacturer: Philips

Part Number: VY22575

Description: This is the main processor of the ControlLogix L61. No official documentation is available for this chip. It is presumably a custom design based on the ARM7TDMI, as determined by ARM's RealView Debugger configuration tool. Further research suggests the chip may be similar to the ARM740T, a specific configuration of the ARM7TDMI. This is due to the fact that the ARM740T includes a memory protection unit (MPU) and initialization code for the ControlLogix L61 references MPU initialization functions. A comparison of this MPU initialization code with official documentation for the ARM740T reveals similar, but not identical, operation of the MPU [5]. Little else is known about this chip.

B. Backplane Communications Processor

Manufacturer: Atmel

Part Number: AT56J05-UQ3T

Description: This is the processor used to handle backplane communication on the device. No official documentation is available for this chip. It is presumably another custom chip. The same chip is present on most ControlLogix modules near the backplane connector. Little is known about its design or functionality.

C. Non-volatile Storage (Flash Memory)

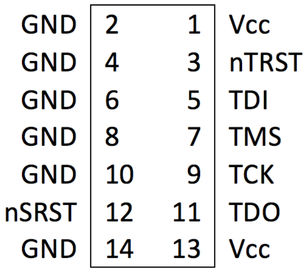
Manufacturer: Intel

Part Number: 640J3F75

Description: This is the flash memory chip used for non-volatile storage on the device. No official documentation is available for this specific chip. However, a search of the part number results in several references to flash chips manufactured by Micron

Technology with similar part numbers. Intel rolled their flash memory operations into a joint venture company called Numonyx in 2008. Two years later, Micron Technology acquired Numonyx [56]. The data sheet found for a Numonyx flash chip by Micron seems to reference a very similar, if not identical chip [36]. A comparison of flash memory operations performed by the ControlLogix L61 firmware correctly correspond to operations found in this datasheet.

D. Standard 14-pin ARM JTAG Connector Pads



GND	2	1	Vcc
GND	4	3	nTRST
GND	6	5	TDI
GND	8	7	TMS
GND	10	9	TCK
nSRST	12	11	TDO
GND	14	13	Vcc

Figure D.2: 14-pin ARM JTAG pin configuration as viewed in Figure D.1

E. User Memory

Manufacturer: Cypress

Part Number: CY7C1041CV33

Description: Static RAM (SRAM), 4Mb; This is the user memory available to the operator for storing projects. Capacity: 4Mb per module, 2MB total. Official documentation is found in Reference [16].

F. Volatile RAM

Manufacturer: Micron Technologies

Part Number: 48LC4M16A2

Description: Synchronous Dynamic RAM (SDRAM), 64Mb; This is the volatile RAM used by the device during runtime. Capacity: 64Mb per module, 24MB total. Official documentation is found in Reference [35].

Appendix E: Source Code

E.1 zlib_analysis.sh

```
#!/bin/bash
#zlib script

if [ ! -n ”$1” ]
then
  echo ”usage: ‘basename $0‘ <filename>”
  exit
fi

mkdir tmp
cd tmp
binwalk -y zlib ../$1 --dd=zlib:zlib >/dev/null

for file in *.zlib
do
  if ../zpipe -d < $file > $file.bin 2>/dev/null
  then
    echo ”$file seems legit”
    cp $file.bin ../
  #else
    #rm $file.bin
    #echo ”$file was deleted”
  fi
done

cd ..
#rm -rf tmp
```

Appendix F: IDA Scripts

F.1 FindARMFunctions.idc

```
// FindARMFunctions.idc
/* Based on MakeFuncs.idc by Ruben Santamarta, www.reversemode.com */
/* Edited by Zachry Basnight - AFIT */
/* Find and make functions in ControlLogix 1756-L61 firmware */

#include <idc.idc>
static main() {
  auto ea;
  auto eaFunc;
  auto minea;
  auto prolog;
  auto i;
  auto gProArray;

  minea = MinEA();
  SetStatus(IDA_STATUS_WORK);
  Message(”Fixing firmware...\n”);

  // create array of op code signatures for functions
  gProArray = CreateArray(”ProGos”);
  if (gProArray == -1)
    gProArray = GetArrayId(”ProGos”);
  SetArrayString(gProArray,0,”2D E9”); // ”2D E9” is the only one used here

  for( i = 0; i<1; i++ ) { // for each op code in the array
    ea = minea;
    prolog= GetArrayElement(AR_STR,gProArray,i);
    Message(”Opcodes: [ %s ]...\n”,prolog);

    while(1) { // search binary for op code
      eaFunc = FindBinary( ea, SEARCH_DOWN, prolog ); // find next instance
      if( eaFunc == BADADDR )
        break;                         // break if there are no more
      eaFunc = eaFunc - 2;             // back up to the start of the little-endian word
      MakeCode( eaFunc );              // disassemble the code here
      MakeFunction( eaFunc, BADADDR ); // make the code into a function
      ea = eaFunc + 4;                 // move to the next word
    }
    Message(”OK\n”);
  }
  Message(”Done\n”); SetStatus(IDA_STATUS_READY);
}
```

F.2 NameL61Functions.idc

```
// NameL61Functions.idc
// Written by Zachry Basnight - AFIT
// Loosely based on code by Ruben Santamarta, www.reversemode.com
// Name functions in ControlLogix 1756-L61 firmware based on source file strings

#include <idc.idc>
static main() {
  auto ea;
  auto eaFunc;
  auto str;
  auto i;
  auto name;
  auto len;
  auto end;
  auto start;
  auto ch;
  auto refAddr;
  auto funcName;
  auto suffix;
  auto NamesArray;
  auto arrayLen;
  auto idx;

  ea = MinEA();
  SetStatus(IDA_STATUS_WORK);

  str = ”2E 2E 5C”; // ”..\” magic string identifying the start of source filenames
  NamesArray = CreateArray(”Names”);
  arrayLen = 0;

  while (1) { // search entire binary
    ea = FindBinary(ea, SEARCH_DOWN, str); // search for magic string
    if( ea == BADADDR )
      break; // break if error or past the end
    name = Name(ea); // get the IDA database name of the current location
    len = 1;

    if (name != ””) { // if the location is not already named...
      Message(”%s\n”, name);

      // get binary as a string and determine its length
      name = GetString(ea, -1, ASCSTR_C);
      len = strlen(name);
      end = len;

      do { // working backwards, remove the file extension from ”name”
        end = end-1;
        ch = substr(name, end, end+1);
      } while (ch != ”.”);

      start = end;
      do { // still working backwards, find the start of the filename (not the path)
        start = start-1;
        ch = substr(name, start, start+1);
      } while (ch != ”\\”);
      start = start+1;
      name = substr(name, start, end); // ”name” is now just the filename sans extension
      Message(”%s\n”, name);

      //search for name in array of names already found/used
      idx = arrayLen;
      suffix = -1;
      for (i=0; i<arrayLen; i++) {
        if (name == GetArrayElement(AR_STR, NamesArray, i*2)) {
          suffix = GetArrayElement(AR_LONG, NamesArray, (i*2)+1);
          idx = i*2;
          break;
        }
      }

      //if not found add it
      if (idx == arrayLen) {
        SetArrayString(NamesArray, idx, name);
        SetArrayLong(NamesArray, idx+1, 0);
        arrayLen = arrayLen+2;
      }

      refAddr = DfirstB(ea); // get the first address that references the filename string
      //Message(”%x\n”, refAddr);

      // for every cross reference address, name that address as the filename
      while (refAddr != BADADDR) {
        funcName = GetFunctionName(refAddr);
        if (funcName != ””) { // if address is a named function...
          if (strstr(funcName, ”sub_”) == 0) { // that starts with ”sub_” (autonamed)...
            suffix = suffix + 1;
            eaFunc = LocByName(funcName); //rename func and add the appropriate suffix
            MakeNameEx(eaFunc, sprintf(”%s_%d”, name, suffix), SN_NOCHECK & SN_NON_AUTO & SN_NOWARN);
          }
        }
        refAddr = DnextB(ea, refAddr);
        //Message(”%x\n”, refAddr);
      }
      SetArrayLong(NamesArray, idx+1, suffix);
    }
    ea = ea+len; // increment past the string and keep going
  }
  Message(”Done\n”); SetStatus(IDA_STATUS_READY);
}
```

Appendix G: ARM DS-5 Debugger Scripts

G.1 EasyReset.ds

```
# EasyReset.ds
# Written by Zachry Basnight -- AFIT
# This script resets the L61 when halted in the DS-5 debugger.

# The L61 starts execution at 0x08000000 on powerup, but the
# DS-5 debugger resets PC to 0, so we need to load the boot code
restore 0x08000000.bin binary 0x0
    # Where ”0x08000000.bin” is a binary file containing the
    # L61 boot code located at 0x08000000.
wait
hbreak -p *0x00D00000     # set hardware breakpoint at the start of the OS firmware
set var $pc = 0x00000000  # reset PC to 0
wait
continue

# execution will break at the start of the firmware, use ”continue” to run OS
```

G.2 FullReset.ds

```
# FullReset.ds
# Written by Zachry Basnight -- AFIT
# This script loads a desired OS firmware into memory, resets the L61 allowing ExecLoader
# to initialize, then jumps directly to the loaded firmware. Useful for debricking.

# load 0x08000000 boot code to 0, see EasyReset.ds
restore 0x08000000.bin binary 0x0
wait

# load desired firmware to runtime memory location
restore ”..\mem_dumps\PN-66834_with_padding.bin” binary 0xD00000
wait
hbreak -d -p *0x0A000098  # set hardware breakpoint after ExecLoader initializes
set var $pc = 0x00000000  # reset PC to 0
wait                      # wait for ”set var” to finish
continue                  # continue execution at 0
wait                      # wait until execution breaks at 0x0A000098
set var $pc = 0x00D00000  # set PC to start of loaded firmware
wait

# execution will break at the start of the firmware, use ”continue” to run OS
```

Bibliography

[1] "Teenage hacker faces federal charges". *CNN*, March 18, 1998. URL http://www.cnn.com/TECH/computing/9803/18/juvenile.hacker/index.html.

[2] Adler, Mark. "zpipe.c". *zlib.net*, December 11, 2005. URL http://www.zlib.net/zpipe.c.

[3] Agrawal, Dakshi, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. "The EM SideChannel(s)". *Cryptographic Hardware and Embedded Systems-CHES 2002*, 29–45, 2003.

[4] ARM. *ARM Software Development Toolkit Reference Guide*, 1998.

[5] ARM. *ARM740T Datasheet*, February 1998. ARM DDI 0008E.

[6] ARM. *ARM Architecture Reference Manual*, 2005.

[7] Baldwin, Kristen, John F. Miller, Paul R. Popick, and Jonathan Goodnight. "The United States Department of Defense revitalization of system security engineering through program protection". *Systems Conference (SysCon), 2012 IEEE International*, 1–7. IEEE, 2012.

[8] Barker, Elaine, William Barker, William Burr, William Polk, and Miles Smid. *Recommendation for Key Management - Part 1: General (Revision 3)*. National Institute of Standards and Technology, July 2012.

[9] Beresford, Dillon. "Exploiting Siemens Simatic S7 PLCs". *Black Hat USA*, 2011.

[10] Breeuwsma, Ing. M. F. "Forensic imaging of embedded systems using JTAG (boundary-scan)". *Digital Investigation*, 3(4):32–42, March 2006.

[11] Breeuwsma, Marcel, Martien de Jongh, Coert Klaver, Ronald van der Knijff, and Mark Roeloffs. "Forensic Data Recovery from Flash Memory". *Small Scale Digital Device Forensics Journal*, 1(1):1–17, June 2007.

[12] Brunner, Martin, Hans Hofinger, Christoph Krauss, Christopher Roblee, Peter Schoo, and Sascha Todt. *Infiltrating Critical Infrastructures with Next-Generation Attacks: W32.Stuxnet as a Showcase Threat*. Fraunhofer-Institute for Secure Information Technology, Germany, December 2010.

[13] Cai, Ning, Jidong Wang, and Xinghou Yu. "SCADA System Security: Complexity, History and New Developments". *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, 569–574. IEEE, 2008.

[14] Chen, Thomas M. and Saeed Abu-Nimeh. “Lessons from Stuxnet”. 44(4):91–93, April 2011.

[15] Cook, Gregory. “README for CRC RevEng 1.03”. July 12 2012. URL http://reveng.sourceforge.net/readme.htm.

[16] Cypress Semiconductor Corporation. *CY7C1041CV33*, May 28, 2011. 38-05134 Rev. *N.

[17] Deutsch, P. and Jean-Loup Gailly. *ZLIB Compressed Data Format Specification version 3.3*. Network Working Group, Internet Engineering Task Force, 1996. RFC 1950.

[18] DI Management. *BigDigits multiple-precision arithmetic source code*, January 24, 2012. URL http://www.di-mgt.com.au/bigdigits.html.

[19] Falliere, Nicolas, Liam O Murchu, and Eric Chien. *W32.Stuxnet Dossier*. Symantec Corporation, Cupertino, CA, February 2011. Rep. Ver. 1.4.

[20] Fletcher, John G. “An Arithmetic Checksum for Serial Transmissions”. *Communications, IEEE Transactions on*, 30(1):247–252, 1982.

[21] Goldberg, Carey. “Federal Charges for Juvenile In a Case of Computer Crime”. *The New York Times*, March 19, 1998. URL http://www.nytimes.com/1998/03/19/us/federal-charges-for-juvenile-in-a-case-of-computer-crime.html.

[22] Halbronn, Cedric and Jean Sigwald. “iPhone Security Model & Vulnerabilities”. *Proceedings of Hack in the Box Sec-Conference. Kuala Lumpur, Malaysia*. 2010.

[23] Heffner, Craig. “Reverse Engineering Firmware: Linksys WAG120N”. *DEV/TTYS0*, May 29, 2011. URL http://www.devttys0.com/2011/05/reverse-engineering-firmware-linksys-wag120n/.

[24] Industrial Control Systems Cyber Emergency Response Team. *ICS-CERT Monitor*, December 2012. URL http://ics-cert.us-cert.gov/pdf/ICS-CERT_Monthly_Monitor_Oct-Dec2012.pdf.

[25] Institute of Electrical and Electronics Engineers. *IEEE 1149.1-2001 Standard Test Access Port and Boundary-Scan Architecture*, 2001.

[26] Institute of Electrical and Electronics Engineers. *IEEE 802.3-2008 IEEE Standard for Information technology-Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, 2008.

[27] International Electrotechnical Commission. *IEC 61131-3*, 2003.

[28] International Organization for Standardization. *ISO 1155:1978 Information processing – Use of longitudinal parity to detect errors in information messages*, 1978.

[29] Kocher, Paul. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". *Advances in CryptologyCRYPTO96*, 104–113. Springer, 1996.

[30] Kocher, Paul, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". *Advances in CryptologyCRYPTO99*, 789–789. Springer, 1999.

[31] Koopman, Philip. "Embedded Software Testing". *Carnegie Mellon University*, 2011. URL http://www.ece.cmu.edu/~ece649/lectures/08_testing.pdf.

[32] Maxino, Theresa C. and Philip J. Koopman. "The Effectiveness of Checksums for Embedded Control Networks". *IEEE Transactions on Dependable and Secure Computing*, 6(1), March 2009.

[33] McMinn, Lucille and Jonathan Butts. "A Firmware Verification Tool for Programmable Logic Controllers". *Critical Infrastructure Protection VI*. Springer, 2012.

[34] Menezes, Alfred J., Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC, 1996.

[35] Micron Technology, Inc. *64Mb: x4, x8, x16 SDRAM*, 2003. Rev. F.

[36] Micron Technology, Inc. *Numonyx Embedded Flash Memory (J3 65 nm) Single Bit per Cell (SBC)*, January 2011. 208032-03.

[37] Microsoft. *Microsoft Portable Executable and Common Object File Format Specification*, September 21, 2010.

[38] Monti, Eric. "Retsaot is Toaster, Reversed: Quick 'n Dirty Firmware Reversing". *Chargen*, April 29, 2008. URL http://chargen.matasano.com/chargen/2008/4/29/retsaot-is-toaster-reversed-quick-n-dirty-firmware-reversing.html.

[39] National Communications System Technology & Standards Division, General Services Administration Information Technology Service. *Telecommunications: Glossary of Telecommunication Terms*, April 7, 1996. Federal Standard 1037C.

[40] National Institute of Standards and Technology. *Vulnerability Summary for CVE-2005-1983*, March 8, 2011. URL http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2005-1983.

[41] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*, March 2012. FIPS PUB 180-4.

[42] Niland, Marty. "Virus Disrupts Train Signals". *CBS News*, August 21, 2003. URL http://www.cbsnews.com/stories/2003/08/21/tech/main569418.shtml.

[43] Oshana, Rob. "Introduction to JTAG". *Embedded*, October 29, 2002. URL http://www.embedded.com/electronics-blogs/beginner-s-corner/4024466/Introduction-to-JTAG.

[44] Peck, Daniel and Dale Peterson. "Leveraging Ethernet Card Vulnerabilities in Field Devices". *SCADA Security Scientific Symposium (S4)*, 2009.

[45] Poulsen, Kevin. "Slammer worm crashed Ohio nuke plant network". *Security Focus*, August 19, 2003. URL http://www.securityfocus.com/news/6767.

[46] Ramabadran, Tenkasi V. and Sunil S. Gaitonde. "A Tutorial on CRC Computations". *Micro, IEEE*, 8(4):62–75, 1988.

[47] Roberts, Paul F. "Zotob, PnP Worm Slam 13 DaimlerChrysler Plants". *eWeek*, August 18, 2005. URL http://www.eweek.com/c/a/Security/Zotob-PnP-Worms-Slam-13-DaimlerChrysler-Plants/.

[48] Roberts, Paul F. "Zotob Worm Targets Windows 2000 Hole". *eWeek*, August 15, 2005. URL http://www.eweek.com/c/a/Security/Zotob-Worms-Target-Windows-2000-Hole/.

[49] Rockwell Automation. *ControlLogix System User Manual*, February 2012. 1756-UM001M-EN-P.

[50] Rockwell Automation. *Logix5000 Controllers Nonvolatile Memory Card*, November 2012. URL http://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm017_-en-p.pdf. 1756-PM017E-EN-P.

[51] Santamarta, Ruben. "Reversing Industrial Firmware for Fun and Backdoors I". *Reversemode*, December 12, 2011. URL http://reversemode.com/index.php?option=com_content&task=view&id=80&Itemid=1.

[52] Santamarta, Ruben. "Project Basecamp: Attacking ControlLogix". *SCADA Security Scientific Symposium (S4)*, 2012.

[53] Skochinsky, Igor. "Intro to Embedded Reverse Engineering for PC Reversers". *Recon*, 2010.

[54] Slay, Jill and Michael Miller. "Lessons Learned from the Maroochy Water Breach". *Critical Infrastructure Protection*, volume 253, chapter 6, 73–82. Springer, Boston, 2007.

[55] Stouffer, Keith, Joe Falco, and Karen Scarfone. *Guide to Industrial Control Systems (ICS) Security*. National Institute of Standards and Technology, June 2011.

[56] Swanekamp, Kelsey. "Micron Shares Short Out". *Forbes*, February 10, 2010. URL http://www.forbes.com/2010/02/10/micron-numonyx-intel-markets-equities-acquisition.html.

[57] Tehranipoor, Mohammad, Hassan Salmani, Xuehui Zhang, Xiaoxiao Wang, Ramesh Karri, Jeyavijayan Rajendran, and Kurt Rosenfeld. "Trustworthy Hardware: Trojan Detection and Design-for-Trust Challenges". *Computer*, 44(7):66–74, July 2011.

[58] Viehböck, Stefan. "Reverse engineering an obfuscated firmware image E02 - analysis". *Braindump*, September 9, 2011. URL http://sviehb.wordpress.com/2011/09/09/reverse-engineering-an-obfuscated-firmware-image-e02-analysis/.

[59] Wang, Tielei, Tao Wei, Guofei Gu, and Wei Zou. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection". *Security and Privacy (SP), 2010 IEEE Symposium on*, 497–512. IEEE, 2010.

[60] Zetter, Kim. "SCADA Systems Hard-Coded Password Circulated Online for Years". *Wired*, July 19, 2010.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From — To)*
21–03–2013	Master's Thesis	Oct 2011–Mar 2013

4. TITLE AND SUBTITLE Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Basnight, Zachry H., First Lieutenant, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB, OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-13-M-06
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security, ICS-CERT Neil Hershfield 900 N. Stuart St. Apt. 715 Arlington, VA 22203	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Recent attacks on industrial control systems (ICSs), like the highly publicized Stuxnet malware, have perpetuated a race to the bottom where lower level attacks have a tactical advantage. Programmable logic controller (PLC) firmware, which provides a software-driven interface between system inputs and physically manifested outputs, is readily open to modification at the user level. Current efforts to protect against firmware attacks are hindered by a lack of prerequisite research regarding details of attack development and implementation. In order to obtain a more complete understanding of the threats posed by PLC firmware counterfeiting and the feasibility of such attacks, this research explores the vulnerability of common controllers to intentional firmware modifications. After presenting a general analysis process that takes advantage of various techniques and methodologies applied to similar scenarios, this work derives the firmware update validation method used for the Allen-Bradley ControlLogix PLC. A proof of concept demonstrates how to alter a legitimate firmware update and successfully upload it to a ControlLogix L61. Possible mitigation strategies discussed include digitally signed and encrypted firmware as well as preemptive and post-mortem analysis methods to provide protection. Results of this effort facilitate future research in PLC firmware security through direct example of firmware counterfeiting.

15. SUBJECT TERMS

industrial control system, programmable logic controller, firmware, reverse engineering

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Maj Jonathan Butts (ENG)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(include area code)*
U	U	U	UU	120	(937) 255-3636 x4332 Jonathan.Butts@afit.edu