3-14-2014

# Firmware Modification Analysis in Programmable Logic Controllers

Arturo M. Garcia


# FIRMWARE MODIFICATION ANALYSIS IN PROGRAMMABLE LOGIC CONTROLLERS

THESIS

Arturo M. Garcia Jr., Captain, USA

AFIT-ENG-14-M-32

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

AFIT-ENG-14-M-32

# FIRMWARE MODIFICATION ANALYSIS IN 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

Arturo M. Garcia Jr., B.S.S.E.C.A.

Captain, USA

March 2014

AFIT-ENG-14-M-32

# FIRMWARE MODIFICATION ANALYSIS IN PROGRAMMABLE LOGIC CONTROLLERS

Arturo M. Garcia Jr., B.S.S.E.C.A.
Captain, USA

Approved:

| | |
|---|---|
| //signed// | 7 MAR 2014 |
| Robert F. Mills, PhD (Chairman) | Date |
| //signed// | 7 MAR 2014 |
| Maj Jonathan Butts, PhD (Member) | Date |
| //signed// | 7 MAR 2014 |
| Juan Lopez Jr. (Member) | Date |

AFIT-ENG-14-M-32


## Abstract

Incorporating security in supervisory control and data acquisition (SCADA) systems and sensor networks has proven to be a pervasive problem due to the constraints and demands placed on these systems. Both attackers and security professionals seek to uncover the inherent roots of trust in a system to achieve opposing goals. With SCADA systems, a battle is being fought at the *cyber–physical* level, specifically the programmable logic controller (PLC). The Stuxnet worm, which became increasingly apparent in the summer of 2010, has shown that modifications to a SCADA system can be discovered on infected engineering workstations on the network, to include the ladder logic found in the PLC. However, certain firmware modifications made to a PLC can go undetected due to the lack of effective techniques available for detecting them.

Current software auditing tools give an analyst a singular view of assembly code, and binary difference programs can only show simple differences between assembly codes. Additionally, there appears to be no comprehensive software tool that aids an analyst with evaluating a PLC firmware file for modifications and displaying the resulting effects. Manual analysis is time consuming and error prone. Furthermore, there are not enough talented individuals available in the industrial control system (ICS) community with an in-depth knowledge of assembly language and the inner workings of PLC firmware.

This research presents a novel analysis technique that compares a suspected-altered firmware to a known good firmware of a specific PLC and performs a static analysis of differences. This technique includes multiple tests to compare both firmware versions, detect differences in size, and code differences such as removing, adding, or modifying existing functions in the original firmware. A proof-of-concept experiment demonstrates the functionality of the analysis tool using different firmware versions from an Allen-Bradley ControlLogix L61 PLC.

*"But let your communication be, Yea, yea; Nay, nay:*
*for whatsoever is more than these cometh of evil." – Matthew 5:37*

**Acknowledgments**

My sincere gratitude to my committee for their guidance and teamwork which made this thesis possible. Dr. Mills – thank you for your patience, expertise, and understanding. Mr. Lopez – for lending your technical expertise and spending time to ensure I succeeded. Maj Butts – for the resources and opportunities to conduct my research. Capt Sonya – for your advice and insight in software engineering. To my family, church, and friends – thank you for your prayers, unwavering support, and motivation.

Arturo M. Garcia Jr.

## Table of Contents

Page

Page

## List of Figures

Figure Page

## List of Tables

## List of Acronyms

| Acronym | Definition |
|---|---|
| ABI | application binary interface |
| ANSI | American National Standards Institute |
| ARM | Advanced RISC Machine |
| ASCII | American Standard Code for Information Interchange |
| B | Branch (ARM instruction) |
| BIC | Bit Clear (Arm instruction) |
| BL | Branch with Link (ARM instruction) |
| BX | Branch and Exchange (ARM instruction) |
| CMP | Compare (ARM instruction) |
| COTS | commercial off-the-shelf |
| CPU | central processing unit |
| CRC | cyclic redundancy check |
| CUT | component under test |
| DOD | Department of Defense |
| DF1 | D1 (data transparency) + F1 (two-way simultaneous transmission with embedded responses) based on ANSI X3.28-1976 specification subcategories |
| FCC | Federal Communications Commission |
| FMA | firmware modification analysis |
| FMAT | firmware modification analysis tool |
| FPGA | field programmable gate array |
| FRN | firmware revision number |
| GUI | graphical user interface |
| HMI | human machine interface |

| Acronym | Definition |
|---|---|
| HxD | Hexadecimal editor and Disk editor |
| IA | information assurance |
| ICE | in-circuit emulator |
| ICS | industrial control system |
| IDA | Interactive Disassembler |
| IDC | Intaractive Disassembler scripting Code/Commands |
| IDS | intrusion detection system |
| IPS | intrusion prevention system |
| IT | information technology |
| JTAG | Joint Test Action Group |
| LAN | local area network |
| LDC | Load Co-processor from Memory (ARM instruction) |
| LDM | Load Multiple (ARM instruction) |
| LDMDB | Load Multiple, Decrement Before (ARM instruction) |
| LDR | Load Register (ARM instruction) |
| MOV | Move (ARM instruction) |
| MTU | master terminal unit |
| MVN | Move Negative (ARM instruction) |
| MB | Mega Byte |
| ODA | Online Disassembler |
| opcode | operation (instruction) code |
| ORR | Logical OR (Arm instruction) |
| OS | operating system |
| PC | personal computer |
| PLC | programmable logic controller |

| Acronym | Definition |
|---|---|
| PRO | professional |
| RAM | random-access memory |
| RISC | reduced instruction set computing |
| ROM | read only memory |
| RTU | remote terminal unit |
| SCADA | supervisory control and data acquisition |
| SHA | Secure Hash Algorithm |
| SoC | system on a chip |
| STM | Store Multiple (ARM instruction) |
| STMFD | Store Multiple, Full Stack Decrement (ARM instruction) |
| STR | Store Register (ARM instruction) |
| SUT | system under test |
| TCP/IP | Transmission Control Protocol/Internet Protocol |
| TDMI | 16 bit *T*humb + JTAG *D*ebug + fast *M*ultiplier + enhanced in-circuit emulator (*I*CE) |
| US | United States |
| VBinDiff | Visual Binary Difference |

FIRMWARE MODIFICATION ANALYSIS
IN PROGRAMMABLE LOGIC CONTROLLERS

# I. Introduction

## 1.1 Background

Ancient and modern militaries have adopted the use of uniforms both to identify themselves on the battle field and bolster pride and *esprit-de-corps* in their units. In modern times they serve to provide concealment using camouflage. Uniforms have an added benefit of aiding military commanders during inspections to quickly identify Soldiers who are out of uniform, due to the standard appearance that is prescribed to uniformity. It is this theme of uniformity that may provide a key to software modification detection.

Security is a cat-and-mouse game that is played asymmetrically. An attacker may be one step ahead of security professionals in one aspect, and the security professionals may be several steps ahead of the attacker in other aspects. In order for information technology (IT) security to converge to a seamless barrier of protection, security professionals cannot wait for attackers to breach a layer of security before addressing a problem. Security professionals must assume that attackers have the knowledge, tools, and skills to conduct viable attacks against an organizations defenses [47, 48].

Industrial control systems (ICS) have a blend of modern and legacy devices that make up our modern manufacturing and energy producing landscape. Our society is economically and physically dependent on this infrastructure for day-to-day operations. Therefore, security techniques must thoroughly cover the limitations and safety concerns inherent with ICS cyber-physical devices.

## 1.2 Motivation

Programmable logic controllers (PLC) are critical components to ICS and supervisory control and data acquisition (SCADA) systems and are crucial to the ICS security aspect. Though no current malware has been published for PLCs, this potential problem can have devastating and long lasting consequences across a community or country. It must be assumed that attackers have the capabilities to alter the firmware of these devices in order to disrupt them and cause damage to SCADA systems found at critical infrastructure facilities such as a water treatment plant or power plant. Efforts must be prioritized to meet these potential threats.

Research in PLC security has grown in the past few years, but is slow due to the complexities and difficulties surrounding it. Many have proposed security solutions aimed at protection, but few have addressed the area of detection. Detecting security issues is the foundation of information assurance (IA) and is key in building the security strategy for protection, detection, response, and recovery capabilities for PLCs.

## 1.3 Research Goals

This research assumes that a facility with ICS has been compromised, and that a vector of attack focused on altering PLC firmware. Since PLCs do not have built in software security, attackers can potentially modify a PLC's firmware and have full control of the device while evading detection [3].

The goal of this research is to develop a technique that detects and analyzes PLC firmware modifications. The main question asked is how can PLC firmware modifications be detected and how does one characterize the nature of the modifications detected? From this main question further questions are asked to answer the main question. Is a modification made? What is added or deleted to the suspect file compare to the baseline file? What opcode characteristics changed? Where is code changed in the firmware? Finally, what functionality is modified, added, or deleted to the suspect file?

The technique that answers these questions is expressed as a software tool that automates the analysis and compares a known-good baseline with a suspect firmware file to determine if the suspect is modified. The tool then characterizes and displays the differences analyzed including additions or deletions made to the suspect file compare to the baseline. This software runs separately from the PLC or engineering workstation, and does not introduce an additional attack vector into the PLC or SCADA system.

## 1.4 Approach

This research develops an automated technique for detecting and analyzing modified PLC firmware. The technique uses several static analysis of differences methods between a baseline firmware file and suspect firmware file of the same version. The major steps in the firmware modification analysis technique are: (1) hash comparison, (2) size comparison, (3) disassembly of both firmware files, (4) opcode histogram comparison, (5) opcode difference comparison, (6) and function difference comparison. The first step detects modifications made to the suspect, while the latter steps characterize the nature of the modification if detected. Once these steps are performed, views are generated to indicate how, and where the modifications were made at the opcode instruction level.

The technique is then tested using five test cases and one control case representing various degrees of modifications. Both the baseline and suspect are obtained from the manufacturer's website. The baseline firmware remains unmodified, while three suspect firmware test cases are modified. The remaining three test cases are unmodified, including the control case. Once the firmware samples are obtained, the test cases are compared to their baseline firmware file versions using the automated technique. The techniques effectiveness is evaluated based on the test results which are validated by the a fellow graduate student who performed the modifications on the suspect firmware samples.

## 1.5 Contributions

This research intends to provide a detection tool that identifies and displays the modifications made to PLC firmware. This tool could be useful in aiding incident response teams in identifying and confirming modifications made to firmware, and enable them to ascertain the extent of the modification quickly and accurately.

## 1.6 Organization

The paper is organized as follows. Chapter II covers background research done with SCADA security, PLC security, and PLC malware detection. It also covers the current tools used in static analysis of PLC firmware and their limitations. Chapter III covers the firmware modification analysis methodology and Chapter IV covers the test cases which will evaluate the methodolgy. Chapter V covers results of the analysis, and Chapter VI presents future work and concludes the paper.

# II. Background

In this chapter, a brief example of SCADA device functionality is covered. Next, general SCADA security issues is discussed, covering different aspects of the security issues to give the reader an appreciation of the difficulty surrounding this topic. Further focus is placed on PLC security and the research that has addressed this topic. Finally, current tools and techniques used for reverse engineering firmware and detecting firmware alterations are discussed.

## 2.1 Cyber-Physical Devices

Remote Terminal Units (RTU) and PLCs are small computerized systems deployed at specific sites operating in a closed loop feedback system [6]. They provide the ability to measure and control physical processes that form the backbone of ICS. These devices collect data and interact with sensors, motors, valves, and other devices throughout an industrial complex and integrate them into the industrial control system for streamlined management and automation control. Industry uses these devices to operate power plants, automobile manufacturing plants, water treatment facilities, and pharmaceutical manufacturers. RTU and PLC devices are purposefully designed with limited on-board computing resources and functionality such as microprocessors, minimal memory, sensor inputs and outputs, and limited network capabilities to balance cost with efficiency. They are required to work in rugged conditions and handle different types of sensory data in real time, and work uninterrupted for an extended period of time. These systems are designed to have an expected life cycle of at least 10 to 12 years [6], and system engineers typically carry identical PLC spares in their inventory for ease of maintenance. Due to modernization, demand for inter-connectivity with business enterprise networks, and Internet accessibility, many closed and standalone SCADA systems are being

interconnected and integrated into these front end networks [6, 34]. The rapid growth on interconnected SCADA systems coupled with improvements in performance requirements of SCADA devices introduced security vulnerabilities that current information technology security models could not adequately protect [8, 34, 71]. Figure 2.1 illustrates a typical water pump station installation commonly deployed in a metropolitan neighborhood. The SCADA system relies on PLCs to perform monitoring, control, and communications tasks.

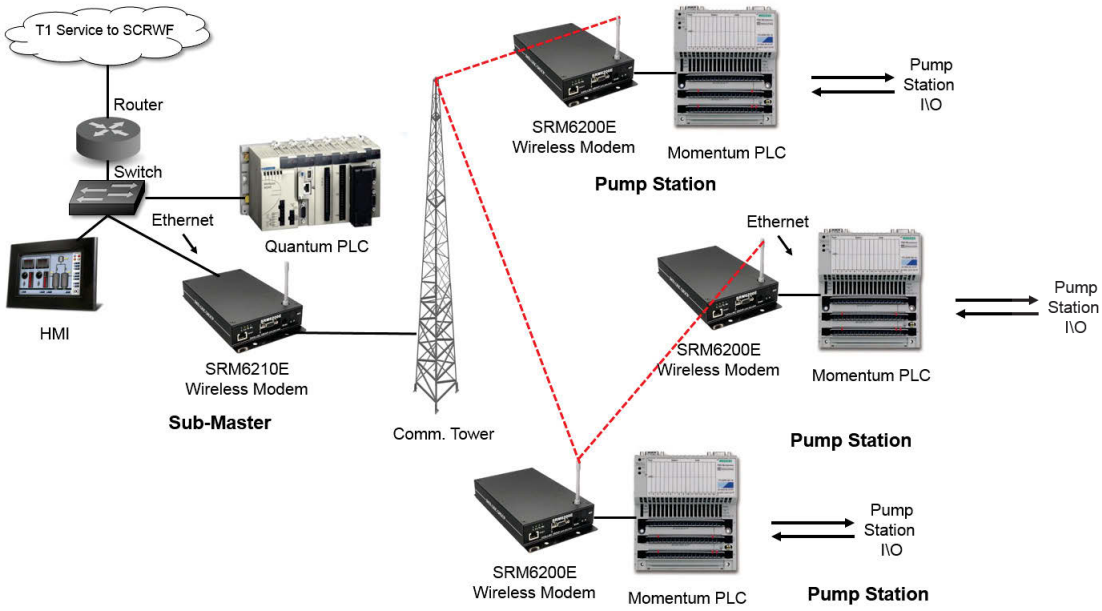



Figure 2.1: Pump Station to Sub-Master PLC Network [58]

In the example shown in Figure 2.1, the PLC controls physical devices such as pumps, flow meters, switches, relays, and solenoids found at a pump station. The PLC operates autonomously and periodically receives updated commands issued through a master terminal unit (MTU) further upstream in the system hierarchy. SCADA wireless data communication links, like the ones shown in the example, are often unsecure or vulnerable to attacks [51, 60]. The unsecure communication links provide an opportunity

for an unauthorized user to intercept and gain access to any of the PLCs through a wireless connection. Anyone with the knowledge, skill, ability, and appropriate equipment can communicate directly with the PLC and affect the physical devices being monitored and controlled by the PLC.

## 2.2 PLC Composition

A PLC is essentially a miniaturized computer which receives inputs from physical devices, performs computations and logic (e.g.,sequencing, timing, counting, and arithmetic) based on the inputs, then produces outputs that further controls physical devices [5, 6]. As shown in Figure 2.2 the PLC itself is composed of three layers as discussed by McMinn *et al*. [49].

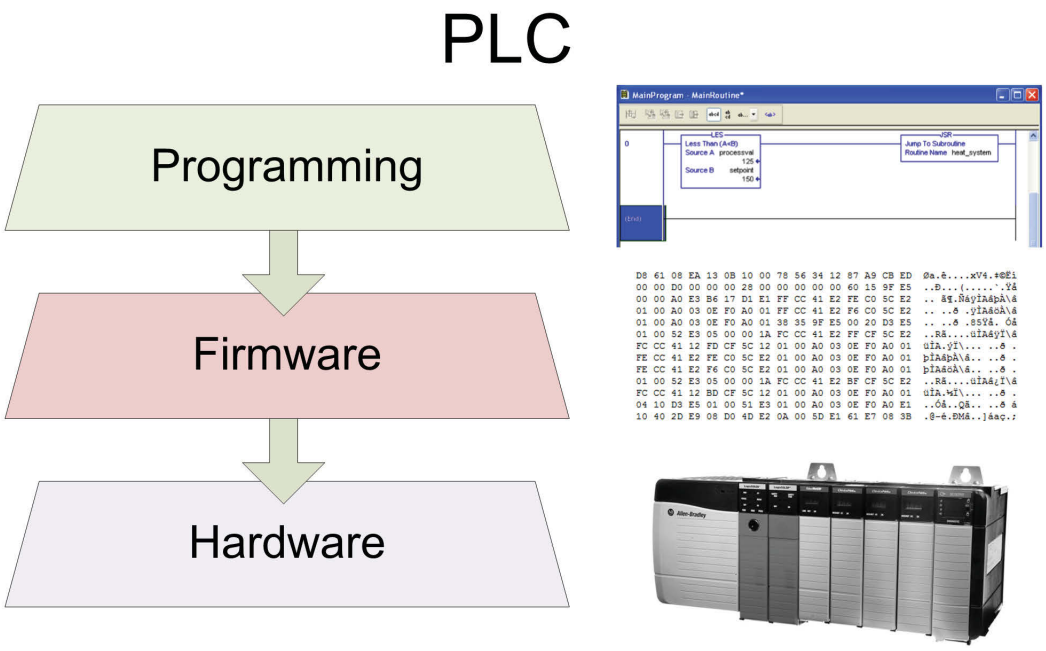



Figure 2.2: PLC Composition Layers [49]

### *2.2.1 Hardware Layer.*

The PLC is a specialized computer systems architecture composed of a microprocessor, volatile and non-volatile memory (RAM and ROM), extended storage for ladder logic (flash memory), power supply, and additional micro-controllers that manage input and output devices connected to the PLC via a back-plane or communications interface [5]. The controller itself may be stand alone or designed to fit on a rack with other devices.

### *2.2.2 Firmware Layer.*

The firmware layer of the PLC represents the software that governs the PLC and the control it has on input and output devices. It contains the operating system and software drivers that control devices such as pumps, solenoids, robotics, and telemetry sensors. PLC manufacturers custom design this software based on the capabilities and application of its hardware. Some PLCs have hardware specifications that allow the operating system to contain a Windows like interface, while other PLC firmwares have monolithic operating systems and do not provide a human interface directly to the PLC environment. PLCs operate under four scan cycles [6].

The first cycle involves administrative overhead functions such as I/O integrity and hardware diagnostic validation. This scan also allows the watchdog timer to interrupt the cycles (for safety reasons) in case the processor has locked up, or the device has faulted. The second scan addresses inputs from input devices and writes data to the input memory table. This records data or position feedback provided by sensors. The third scan cycle performs the ladder logic which is discussed in the programmable layer section. The fourth scan writes the results of the logic computations performed in the third scan and writes the data to the output memory table, which is communicated to the output devices. These four cycles repeat and is the essential operations of a PLC.

PLC firmware is upgradable and may be updated directly to the PLC typically using an RS-232 serial cable or Ethernet connection. Other PLCs may require a proprietary cable

connection, such as the Allen Bradley Micrologix 1000. Normally this involves placing the PLC in a programmable state, and using the manufacturer's software interface and firmware update package to perform the update.

#### *2.2.3 Programmable Layer.*

The programmable layer of the PLC is represented by ladder logic, which directs the PLC to perform specific logic tasks. In the third scan cycle, data was written to the input memory table. This data translates to inputs programmed into the ladder logic. The logic will then dictate how that input is interpreted and what the resulting action will be. The ladder logic specifies what type the input will be (e.g., valve or motor), what the value from the input represents (e.g., on, off, true, false, time, count), and what the resulting value will be placed to another specific device (e.g., light switched on). This programmable layer bridges the operating system portion of the firmware to its device-specific driver software [5, 6].

Ladder logic is typically designed using the manufacturers design software, and uploaded to the PLC in a similar manner as the firmware update. This procedure details differs from manufacturer to manufacturer, but essentially involves uploading the ladder logic project from an engineering workstation to a PLC.

### 2.3 SCADA Security Issues

Some examples of cyber-attacks aimed toward ICS and industry include Stuxnet [44], an Australian sewage treatment plant incident [1], and the industrial network breaches of Slammer [16, 46] and Shamoon [7]. In March of 2000, the Maroochy Water Service Plant in Queensland, Australia sustained a cyber-attack from a disgruntled prospect employee over a period of three months [1]. Vitek Boden was not hired by the company most likely caused by his prior strained relationship with a similar water service company named Hunter Watertech. Wanting revenge, he used radio equipment to send commands at least 46 times to the Maroochy sewage equipment. The resulting actions caused approximately

800 thousand liters of raw sewage to spill into local parks, rivers, and on the grounds of the Hyatt Regency Hotel located nearby [1, 65].

The Slammer Worm discovered in January of 2003 used a buffer overflow vulnerability found in Microsoft SQL servers to infect servers and propagate over the Internet [16]. It did this indiscriminately, and consequently caused denial of service and general internet traffic congestion around the world. The worm eventually found its way into the Davis-Besse Nuclear Power Station in Oak Harbor, Ohio. The worm infected one of the plant's contractor computers, which consequently bridged a T1 connection from their computer to the internal SCADA network of the plant. The worm infected an unpatched server in the network and caused enough network congestion that it shutdown the Safety Parameter Display System. The plant continued to operate, though it caused strain on the operators to get the safety system back up while manually monitoring the safety indicators [16, 35].

Stuxnet became increasingly apparent in the summer of 2010 and is considered the epitome of a sophisticated cyber-attack, carried out by a more formally-constituted, multi-disciplinary "tiger team" [23]. It is the first attack of its kind to target ICS implementing four zero-day exploits with two legitimate, stolen digital-certificates. Stuxnet propagated as a worm through the internet and USB flash memory devices to targeted Iranian nuclear uranium enrichment facilities, such as the Natanz facility. Besides a sophisticated infection and information gathering strategy, the ultimate goal of Stuxnet was to infect the Seimens Simatic S7 PLC, injecting altered ladder logic to ruin 1,000 uranium enrichment centrifuges, all while evading detection [44]. Recent alleged siblings of Stuxnet include Duqu (2011) [4] and Flame (2012) [53].

Shamoon targeted the state owned Saudi Aramco Energy Company, the world's largest oil company, in August of 2012 [7, 17]. This cyber-attack effectively wiped 30,000 computers from Aramco and disabled some of its internal networks for weeks. A hacker group called the Cutting Sword of Justice and Arab Youth Group claimed responsibility for

the attack. It appears the only motive was to delete company data by erasing hard drives and corrupting the master boot record of the infected computers, causing maximum disruption along with replacing some files with pictures of a burning US flag [66]. Business processes were affected, possible production was lost, and the virus spread to other oil and gas firms such as RasGas [7]. Although this attack did not specifically focus on SCADA systems, the target focus is on critical infrastructure.

All of these incidents highlight the necessity to drastically increase critical infrastructure protection. The United States Congress drafted the Cybersecurity Act of 2012 [38], with the intent of fostering collaborative efforts between private industry and government to solve an increasing vulnerability to the nation's critical infrastructure. The act allows for private sector industrial companies to voluntarily share information with the U.S. government in exchange for improved situational awareness of cyber-security threats across critical infrastructure sectors. The legislation is intended to reduce the cyber threat knowledge gap for critical infrastructure sectors and provide momentum towards solving these issues collectively. Since many private sector companies historically do not report cyber-attacks, the legislation is a mechanism that will incentivize reporting [67].

Later in the same year, former Secretary of Defense Leon Panetta addressed the United States, particularly speaking to private businesses and industry about the growing cyber-threat [67]. He highlighted that the United States cannot wait for an *Electronic Pearl Harbor* to occur and do nothing to prevent it. Secretary Panetta spoke out to gain private industry support for the cyber-security act, and to place pressure on law makers to pass the legislation. Mcgraw and others [47, 52] emphasized that although government and cyber-security policies around the world are making forward progress, network complexity, extensibility, and connectivity will continue to abound with technology as a whole. Additionally, as these factors increase, flaws in software will cause vulnerabilities to increase as a result.

Industry has developed a variety of commercial off-the-shelf (COTS) products, diverse protocols, and ubiquitous information technology (IT) security solutions to address growing security threats; however security issues continue to persist. SCADA systems remain difficult to secure because of a growing diversity of SCADA hardware platforms and software protocols, the persistence of legacy systems which are vulnerable to newer threats, and the limitations of security practices on the SCADA network due to resource and demand constraints. Furthermore, it is estimated that there are currently 150 to 200 different SCADA protocols [29].

The large diversity of proprietary SCADA equipment and protocols appear to offer security because an attacker would have to learn a variety of different protocols to be able to affect an entire SCADA system. Notwithstanding, security through obscurity is generally thought of as a weak security practice [47]. Industry is increasingly adopting international open standard network protocols that negate the diversity paradigm [29]. This consolidation of communication protocols results in a streamlined effort of protecting a networked system, but also reduces the amount of familiarity with SCADA protocols required by an attacker to penetrate a SCADA system[14, 56]. Considering the security gaps created by diverse platforms and protocols, standardized and non-standardized systems, and modern devices interconnected with legacy devices, it is clear that achieving a comprehensive security program for critical infrastructure will prove difficult [11, 29].

Nevertheless, current security models do incorporate defense-in-depth strategies to SCADA networks [10, 14, 20, 34]. Most SCADA security models include perimeter defense (e.g., firewalls and proxies), network intrusion detection systems (IDS) and intrusion prevention systems (IPS), workstation anti-virus products, computer use policies, and physical security policies. The concept of defense-in-depth is based on the idea that successive layers of protection will provide adequate protection for the system even if

the outside layer is breached [34, 41]. In defending computer networks, however, the compromise of any element may result in the compromise of the entire system.

To compound matters, not all components of a SCADA network are capable of being adequately secured with this current approach [8, 51, 71]. PLC and RTU devices are vulnerable to security threats because of their inability to detect malicious activity or prevent malicious actions from being passed to the cyber-physical components they are connected to. A skilled attacker with minimal resources may be able to alter PLC firmware, maintain a persistent presence on the network, and is capable of causing longer lasting damage without their participation [3]. This also makes it difficult for security professionals to ascertain the origin of the problem. Simply querying the PLC device is not sufficient since it may contain modified code with the purpose of evading detection [49]. This problem would require an analysis of the firmware itself to determine if it has been altered, to include investigating the specific effects of the modification.

Each organization has freedom to implement their IT strategy as they see fit. For example, the Department of Defense (DOD) and private industry have similar but different motivations that guide their decisions for implementing security. The DOD is mission oriented and bases its security policies on national security and operational requirements [18, 19]. Money is a factor for the DOD but not a driving one. With private industry, money is a primary influential driving factor. Other factors that are catalysts for change include safety, regulations, and legal liability [6, 31]. SCADA infrastructure is designed to last for many years, possibly even decades before being upgraded or replaced, and is expensive to maintain [6]. Another concern to industry is the long-term support to their systems. As stated earlier, it is a general practice to keep a large stock of components such as sensors and PLCs that will last the company many years. This unique situation introduces the mixture of legacy systems with modern systems. Having a security solution

that spans modern and legacy systems is a pervasive security problem, since one solution may not apply to all SCADA devices.

Another security issue that stems from legacy systems is long-term system support. SCADA systems are long-term systems with embedded devices that use a particular software application, operating system, or firmware for possibly a decade or longer. A company who originally provided software support for upgrading and patch management may no longer exist due to closing or absorption by another company who discontinues support [6]. Another concern is that some software may continue to be supported, but a new update may break support for legacy software, which in turn may cause the legacy software to stop working. Due to the real-time demands of SCADA systems, system administrators may choose to avoid patching their systems altogether to skirt this issue [34].

SCADA systems have seen an emergence of high bandwidth connectivity and network throughput in recent years [9, 36]. Along with the demand of continuous operations, system administrators have integrated their corporate local area networks (LAN) to their SCADA networks. This is an effort to increase efficiency and productivity, but also introduced another attack vector for attackers to gain a foothold into the corporate LAN and eventually to the SCADA network [29]. In fact, between 2001 and 2006, 70% of security incidents involving SCADA networks originated outside of the network [9, 29]. This is a significant increase from previous years where that same percentage is due to inside accidents and operator error. The Department of Homeland Security (DHS) Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) recently published in their quarterly newsletter (October-December 2013) [27] that they had responded to over 200 cyber-incidents across all critical infrastructure sectors; with 53% of the incidents focusing on the energy sector in just the first half of the fiscal year. This emergence of focused targeted incidents, coupled with the shortage of qualified cyber-security professionals in

the public sector [62] and increased networking of cyber-physical devices, is a cause for concern.

Another security concern is the unrestricted access to emerging wireless RTUs and wireless field devices. Physical security is a concern particularly for unattended remote sites, where an attacker can access the site and modify, add, or destroy equipment. With wireless devices, an attacker can insert themselves in the traffic, or monitor the network traffic to gain information. Although water towers operate under a licensed FCC frequency, this does not inhibit someone from listening to those frequencies and learning information. The protocols involved were designed with efficiency and backwards compatibility in mind, not security and confidentiality. Data flows between wireless modems unencrypted. An attacker can use a program such as Airsnort or Wireshark to sniff protocol packets and decipher protocols and useful information such as user-names or passwords to computer workstations or PLC devices.

The security issues discussed above is categorized and listed below to summarize security weaknesses and their associated threats [11, 34]:

- **External Network Defense:**

  **Control** – SCADA systems have entry attack vectors that include modems, wireless devices, and the internal corporate LAN. A gateway translates modern TCP/IP connections to SCADA protocols to allow access to the SCADA network remotely. Many gateways do not have security features . Also, a lack of multi-factor authentication allows attackers to crack the only defense of access control and that is usually weak passwords. Attack vectors include: war dialing, wireless jamming, man-in-the-middle attacks, brute-force password guessing or credential-replay attacks [11].

  **Perimeter Defense** – SCADA systems lack public-facing firewalls that are capable of understanding SCADA protocols and filtering traffic accordingly. IDS/ IPS

devices currently cannot monitor suspicious SCADA protocols entering the network. Attack vectors include: PLC/RTU spoofing, wireless device spoofing, wormhole attacks which can lead to key-compromise and theft [56, 59].

- **Internal Network Defense:**– IDS/IPS devices placed in SCADA networks currently cannot monitor suspicious SCADA protocols inside the network or below the RTU/PLC layer. Attack vectors include: PLC/RTU spoofing, wireless device spoofing, wormhole attacks, sybil attacks, replication, routing loops, denial of service and information stealing [56].

- **Cyber-Physical Devices, Servers and Workstations:**

  **Protocol Security** – SCADA protocols do not have built in security and can therefore be spoofed and easily sniffed in plain-text (no encryption or authentication). Attack vectors include: sybil attacks (forged identities), replication, routing loops, denial of service, time-synchronization attacks, slander-attacks for role based authentication schemes [12, 29].

  **Device and OS Security** – Resource constraints placed on PLCs and RTUs makes it difficult to apply security techniques that would defend against or recognize attacks [11, 15]. Out-dated operating systems are vulnerable to known exploits and may be made a pivot for a future attack [34, 54].

Though these threats are formidable, many of these attacks have been addressed successfully in modern IT networks. The key to applying security solutions is to analyze the threats based on their impact and appropriately implementing controls that give the maximum amount of risk control for the organizations confidentiality, integrity, and availability requirements [19, 34]. Another consideration is the placement of security solutions based on their hardware/software trust relationship assumptions [14, 59]. New hardware and software solutions will need realization to meet these unique problems.

PLCs, like any other computer device, assume hardware/software trust relationships that must be understood and protected [55].

## 2.4 Previous Work on PLC Firmware Analysis

With respect to information assurance (IA) there have been few discussions and security implementations regarding detection of malware or attacks on PLC firmware. One researcher addresses external PLC firmware verification on an external device connected to the PLC [30]. A field programmable gate array (FPGA) security appliance is loaded with software that attests the validity of PLC firmware using a SHA1 hashing algorithm to compare firmware hash signatures. Figure 2.3 illustrates this functionality below.

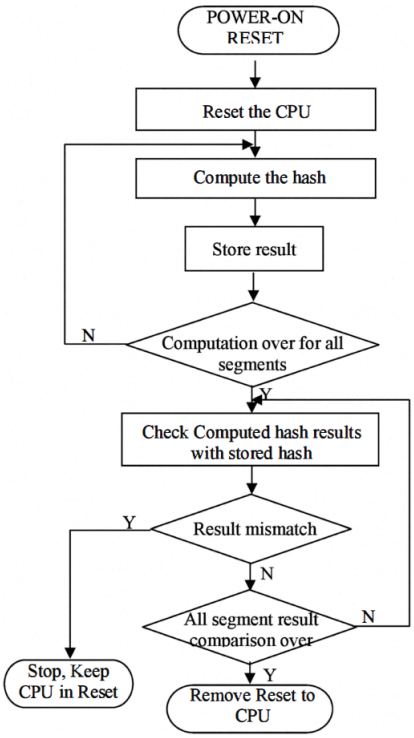



Figure 2.3: Khan's FPGA/PLC Control Flow [30]

A secondary benefit to computing blocked hash checks is the ability to locate mismatched-signature hash blocks, which narrows down the search for detecting modified firmware. Though the appliance is not able to restore corrupted firmware, the FPGA is able to prevent the PLC from executing modified firmware. Once the PLC is restored to its original software, using outside methods from the security appliance, the restored PLC firmware is re-validated and allowed to resume. The security device is attached directly to the PLC and not in-line between the SCADA network and PLC. This research highlights detection of modified code using an external device, and addresses the PLC firmware as a possible attack vector. The author does concede that the security device's inability to restore modified firmware may be considered a form of denial-of-service and undermines the *availability* of the PLC itself.

Another research effort developed software based firmware validation to verify firmware updates being uploaded to a PLC from an engineer workstation [49]. Figure 2.4 illustrates the experiment set up. The hardware/software trust relationship above the PLC is seen as an attack vector leading to the PLC and assumed to be compromised. The security objective of the research is to identify modified firmware before it is downloaded to a PLC during a firmware update. The proof of concept experiment utilized the Allen-Bradley DF1 protocol. DF1 is a byte-oriented Allen-Bradley specific data-link layer protocol that combines features of data transparency and two-way simultaneous transmissions with embedded responses of the ANSI x3.28 specification [2]. The validation software incorporates a firmware hash validation mechanism to verify the firmwares hash signature while in-transit. The software is trained with an original firmware file and then determines if a suspect firmware file is modified while the suspect firmware is uploaded. This approach contributes PLC firmware modification detection using a binary detection classifier: (Yes/No) classification prior to a PLC firmware update.

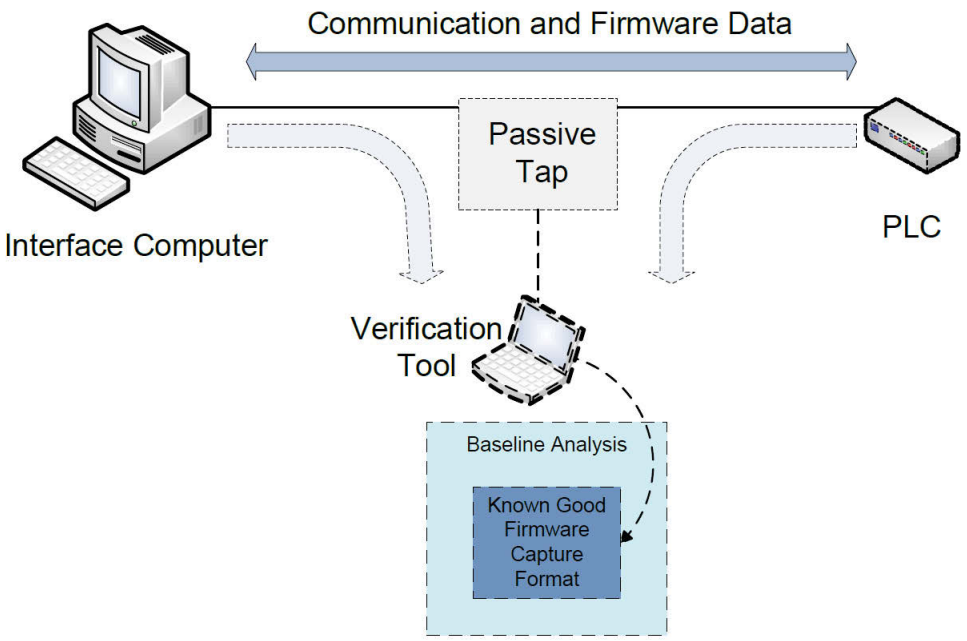



Figure 2.4: McMinn's Passive Capture and Baseline Analysis Experiment [50]

Basnight *et al.* [3] took a deeper look at PLC firmware modification by using reverse engineering techniques to determine the firmware's security protection schema for detecting firmware modifications. They argue that security in the PLC is non-existent, and that a check-sum algorithm in the firmware only serves to ensure that the firmware is not corrupted, and is incapable of preventing or detecting an intentional modification to the firmware. Basnight's reverse engineering technique to identify vulnerabilities associated with the firmware update process resulted in a successful exploit. The reverse engineering process for firmware is illustrated in Figure 2.5 below. The firmware is wrapped in an executable file provided by the manufacturer's website is then extracted and analyzed. The technique used to analyze the ControlLogix L61 PLC provided useful details concerning the L61 firmware structure, functionality, and the language of the processor. Once the

firmware is analyzed and an attack vector is selected based on the upload process, they successfully altered Allen-Bradley PLC firmware version number, recalculated the checksum and CRC codes, and then successfully installed the modified firmware onto the PLC using only commercially available tools from the manufacturer. This proof of concept strengthens the argument for the need to improve critical infrastructure protection. At this juncture, two research techniques demonstrate that modification to PLC firmware is detectable and that firmware modification is feasible. However, these approaches do not provide insight as to the nature of the firmware modification.

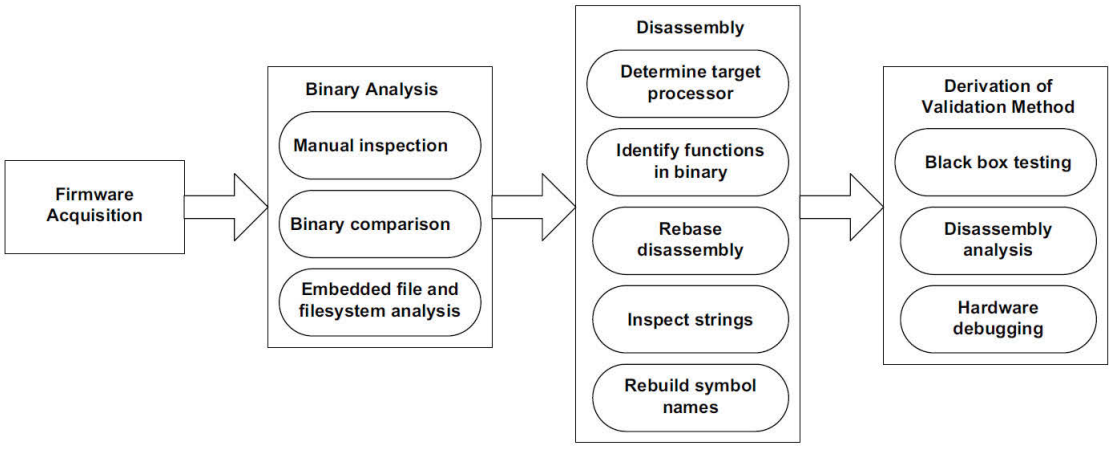



Figure 2.5: Basnight's Firmware Reverse Engineering Process [3]

Sickendick [64] researched file carving and statistical analysis techniques which detect and classify file types and firmware architectures. Using a firmware disassembly algorithm, Sickendick determined that the L61 PLC firmware is uncompressed and that the majority of the firmware is instruction code, with little data. Several classifier algorithms and file segmenter algorithms are tested in a system to determine the most efficient and accurate algorithms that will be used in an automated firmware disassembly and file identification method. The firmware disassembly system is then evaluated as a whole system via

simulation in order to validate the system and characterize real world PLC firmware. The firmware disassembly system and parameters are shown in Figure 2.6 below.

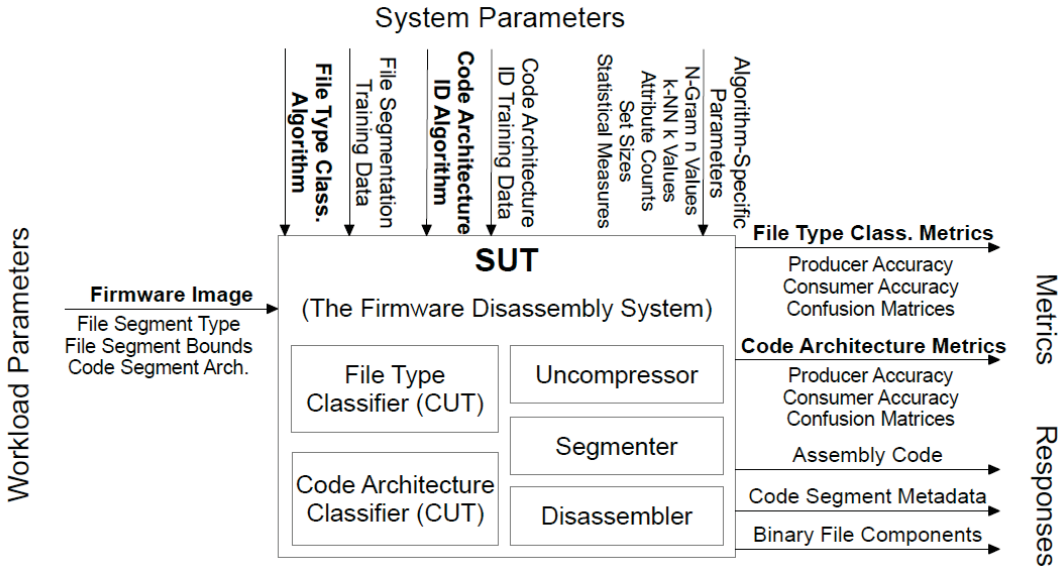



Figure 2.6: Sickendick's Firmware Disassembly System [64]

A notable metric of the classifiers tested is their accuracy, which is measured using true positives, true negatives, false positives, and false negative rates. Furthermore, the time required to train each classifier is considered as well. For example, the normalized compression distance (NCD) algorithm file type classifier, only achieved a true positive rate of 11.1% overall. In a case where all 12 file types needed to be classified, the algorithm required approximately 76 days of training on data prior to classification [64]. Single file type classification training ranged from three to seven days. Time and accuracy were considered for selecting the firmware dissassembly system algorithms. This research steps closer toward firmware analysis with respect to opcode and file type classification. It is important to PLC firmware reverse engineering by providing a technique for classification

and its capabilities are useful for firmware modification analysis when dealing with unknown firmware file types.

Lim & Lee [39] created a methodology for forensic analysis of embedded systems. They proposed a two prong approach that involves hardware component and trace analysis along with software comparison analysis based on manufacturer specifications. The software analysis focuses on comparisons of firmware revision numbers (FRN), computer system configurations, directory configurations, file/firmware formats, log files, time-lines, meta data, and others with the manufacturers specifications. Once changes have been identified, the analyst can investigate the changes made in a specific category. Although this research provides a framework for understanding forensic analysis of embedded systems, there is a critical assumptions made on the reliance of input from industry. They admit that their software analysis approach assumes industry cooperation to provide proprietary data sheets and functionality inventories of the code, which focuses the analysis scope at structured code layers above the operating system or firmware level to the application and file system level. Without this assumption, the software analysis process is constrained to operate a reverse engineering perspective which must be accomplished at the assembly instruction level first, then abstracting upwards towards understanding the syntax differences of the application and operating system equivalents. Although Lim and Lee's research provides an overarching starting point of forensic analysis for embedded systems, however a technique for firmware modification analysis is not presented.

## 2.5 Current Firmware Analysis Tools and Techniques

Software is normally written in high level languages such as Python, C++, and Java, and then converted into low level machine code for the computer processor to execute. Other programs that require speed and efficiency (e.g., firmware) are written in lower level assembly languages for microprocessor architectures such as ARM, PowerPC, and Intel and subsequently converted into machine code. These lower level languages require more

in-depth knowledge of the computer system architecture compared to high level languages, which abstracts these aspects away from the programmer, and require specialized tools and techniques [21, 22, 26]. If the source code and binary mapping of the firmware is provided to the analyst, the analysis of the software is straight forward, and the analyst may view the source code using an appropriate program language editor/viewer. In the event that the architecture or binary mapping of the firmware is unknown, reverse engineering tools and techniques are utilized first to ascertain the architecture and map the functionality of the firmware and its salient features [68]. Once this step is complete, the analyst can then proceed to determine the firmware's software behavior and structure. Reverse engineering tools specifically focused on assembly code and binary files are discussed along with the techniques that implement them.

### *2.5.1 Techniques.*

The goals of reverse engineering techniques are to derive approximations of software behavior and source code syntax [68]. Not having the original source code implies the need to reverse the process to derive original source code from machine code; from a low level language to high level language again. Software contains both *syntax*, which dictates instructions to be carried out by the microprocessor; and *semantics*, which governs the software's intended states of behavior [22, 68]. These two aspects are discovered using static and dynamic analysis techniques.

#### *2.5.1.1 Static Analysis.*

Static analysis of a binary file seeks to uncover the syntax structure of the file without executing the program code [68]. Many aspects of the file are determined such as the language of the architecture (e.g., ARM instructions), determining encoded files and data such as pictures or video that are embedded in the file, and mapping functionality and program flow. Control flow graphs and call graphs may be created, in order to build upon for dynamic analysis. Static analysis is generally performed first, especially if the file

details are not known prior to analysis. The analyst must ascertain all of the possible states of the source code before the behavior can be understood [22, 68]. The end state of static analysis is to derive a facsimile of the original source code.

#### *2.5.1.2 Dynamic Analysis.*

Dynamic analysis of a binary file seeks to uncover its behavior by executing the program code. This technique seeks to determine how the states of the program changes from one execution point to the next [68]. It takes into consideration the change in data values and how that alters the flow of the program. With the static analysis all flow paths were given as possible paths, however in dynamic analysis only one flow path will be taken based on the execution of register values and data conditions. Dependencies can be discovered, which may lead to uncovering vulnerabilities as a result. This technique captures the conditional relationship between states [68].

Both techniques allow the analyst a method for reverse engineering software to extract its original intended source code and understand its behavior. This task is considered non-trivial [21]. This research, limits the scope of firmware comparison to static analysis because it is non trivial, and because of the additional task of performing these techniques simultaneously while comparing the files side-by-side for differences.

### *2.5.2 Tools.*

The tools presented below, perform aspects of static analysis on binary firmware files. They are not all inclusive in terms of representation, but are common examples. Some are more versatile than others and each has unique functions and capabilities. A single tool is adequate for the task when performing static analysis on a single file; however, comparing two firmware files may require multiple tools to aid the analyst in the comparison.

#### *2.5.2.1 HxD.*

HxD [28] is a free versatile hexadecimal code viewer and editor. Its main purpose is to display hexadecimal code and data from a file, and displays its ASCII equivalent side-by-

side with the hex code on a single screen. Providing the ASCII equivalent characters near the hexadecimal values provides an indication of human-readable string texts in the file. HxD is also an editor, which means the data file is writable, regardless of file type, since it is written back to the file in binary format. This tool is capable of comparing binary files side-by-side (Figure 2.7), and also performs statistical frequency analysis for individual hexadecimal bytes (Figure 2.8). If an analyst were to use this tool for file comparison, they must scroll through the code and determine how the programs differs manually; the tool does not indicate differences. The frequency histogram shown in Figure 2.8 provides a hexadecimal fingerprint for a file at the byte level. Each byte is comprised of two letters like **0xA0** and is counted for each instance in the file, accounting for bytes ranging from **00** to **FF**. When a specific histogram bar is hovered over, the histogram bar turns red and displays the count for that particular byte.

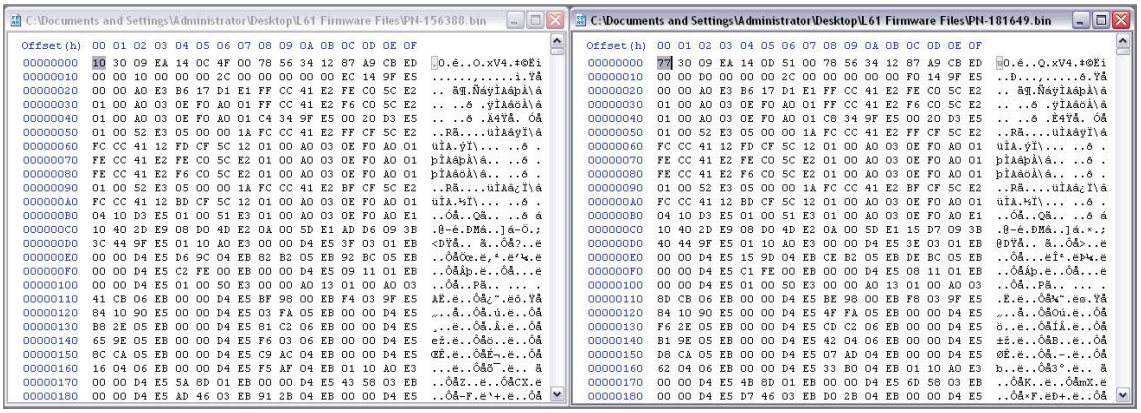



Figure 2.7: HxDs Side-by-Side Code Comparison View [28]

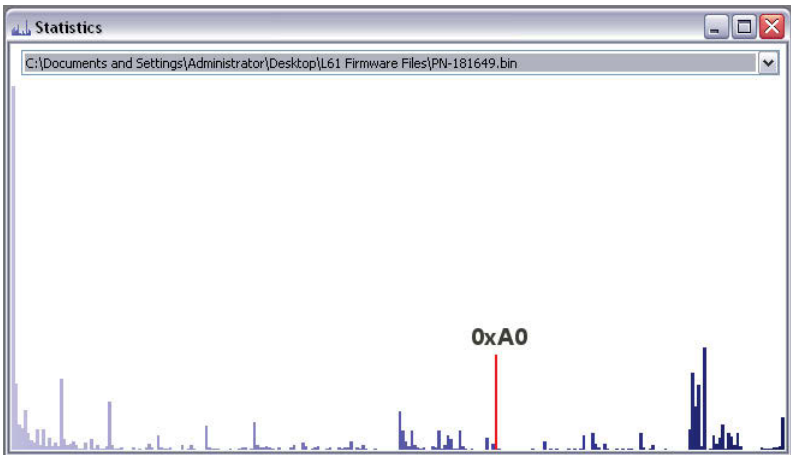



Figure 2.8: HxD's Hexadecimal Byte Histogram [28]

#### *2.5.2.2 VBinDiff.*

Short for Visual Binary Difference [43], this is a free command-line program that is supported by Linux, Mac, and Windows. It has two features: (1) display a single file in its hexadecimal form, and (2) compare two binary files. The byte differences are highlighted in red as shown in Figure 2.9. This tool provides a visual difference view similar to HxD. One effect worth noting is the comparison of two files who have a displacement difference (e.g., additions or subtractions). If code were inserted into one file contrary to the other, all of the data to the right of the insertion would show red inclusively. This is due to byte alignment comparison.

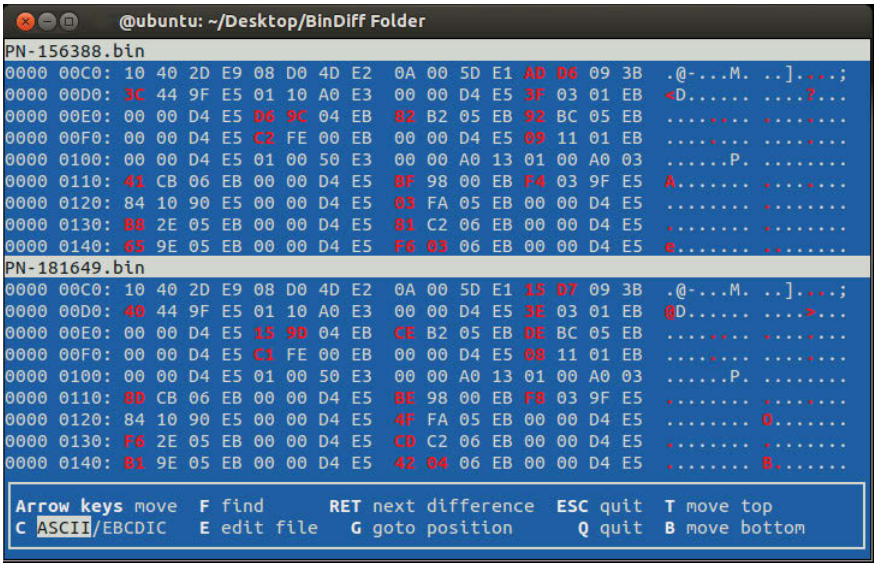



Figure 2.9: VBinDiff Program Comparing Two L61 PLC Firmware Files [43]

#### *2.5.2.3 IDA Pro.*

IDA Pro [25] is a versatile and robust disassembler viewer and debugger, which displays hexadecimal code and the actual instructions based on architecture specifications. IDA Pro is commercially licensed software. It contains scripting command language, IDC, which allows the user to expand the functionality of the program with custom scripts to aide in file processing. One example of the usage of scripts is to identify functions in the PLC firmware. The program effectively disassembles the firmware (Figure 2.10), but it cannot anticipate every binary file format layout. When the L61 firmware is opened, IDA Pro is not able to resolve the function names of the program and created automatically numbered names. An analyst may not recognize the intent of a function if the function name is replaced by a number. With the script command-line, a script can be created to identify each function and re-display the firmware file with the newly found function names. Firmware analysis can begin after the code is prepared to gain as much information as possible. VBinDiff and HxD can be used in conjunction with IDA Pro to investigate

changes in a suspected firmware file from the original. IDA Pro does not allow side-by-side file comparison, and only allows a single program to be analyzed.

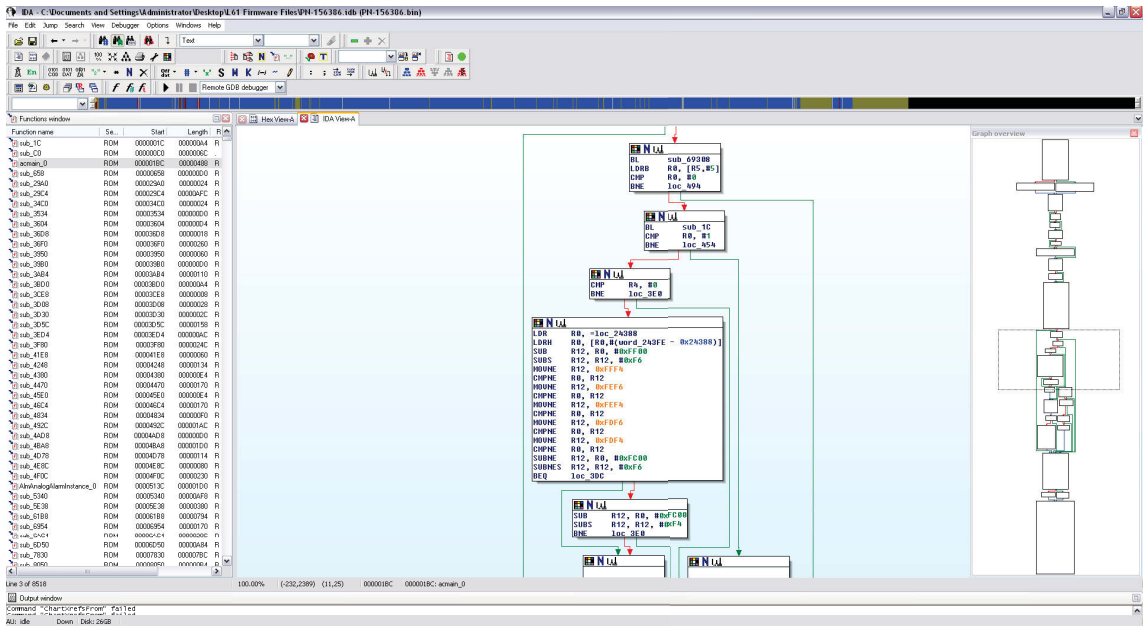


Figure 2.10: IDA Pro Program Displaying a Disassembled L61 Firmware Function [25]

## 2.6 Summary

This chapter covered an introduction to PLC functionality along with general SCADA security issues and PLC security issues. Industry is reluctant to adopt some IT security standard practices because they disrupt safety and availability of data and services in these systems. This is caused by the misalignment of IA objectives between Industry and the generic IT solutions. In order to achieve a comprehensive security solution for PLCs, a strategy must be developed for detecting security threats, reacting to threats, preventing threats, and restoring services from disruptions in a way that addresses PLC availability and integrity of data. The related works covered PLC firmware classification, firmware modification detection, and PLC security vulnerabilities. Assuming a PLC is

compromised, firmware reverse engineering tools and techniques are required to perform modification analysis to uncovering intentional/unintentional modifications. This type of analysis is currently a manual process with specific tools and techniques which aide and shape the process. Tools that automatically compare files highlight changes made and characterize the differences at the hexadecimal byte level. Chapter 3 describes how this research implements a technique that detects and characterizes firmware modifications, while Chapter 4 describes how this research validates its results.

# III. Methodology

In this chapter, the methodology of firmware modification analysis is covered. The research goals, experimental setup, and firmware selection criteria are explained. The scope, assumptions, and limitations are covered under research goals, and validation and approach are covered under the experimental setup section. Chapter four will cover the specific test cases that will evaluate the technique.

## 3.1 Research Goals

Much of the research and issues discussed in chapter two applies to defense-in-depth strategies that prevents malware or disruptions from eventually effecting the PLC's operations. Assuming that security has failed to protect the PLC, it is conceivable that an attacker may install modified firmware onto a PLC once inside the SCADA network. Acting on this assumption, the main goal of this research is to develop a technique that detects and characterizes PLC firmware modifications. Therefore, the main research question asked is how can PLC firmware modifications be detected and how does one characterize the nature of the modifications?

From this main question, additional questions are asked to further illicit details regarding detection and characterization. With respect to detection two questions are asked: (1) Was a modification made?, and (2) does the size comparison of the two files differ? These two questions relate to the first part of the main research question.

Concerning characterization, several questions are asked to understand the nature of the modification, *if* it has occured: (1) What is added or deleted to the suspect firmware file compare to the baseline?, (2) What opcode characteristics changed?, (3) Where is code changed in the firmware?, and finally (4) what functionality is modified, added, or deleted

to the suspect file? These two questions relate to the second part of the main research question.

The overall approach to detect and characterize modified firmware is to compare a suspect firmware file to a baseline firmware file. This baseline is the known-good standard, and the suspected firmware represents firmware that may have been modified on a PLC. Both files undergo several tests that compares the tests' results. Next, strategies that answer the specific questions are discussed.

### *3.1.1 Detecting Modifications.*

To detect modifications made to the suspect file two tests are performed. The first test hashes the base and suspect files separately, then compares the hashes to determine if they differ. This test is performed twice using two hashing methods. The second test is the file size comparison and displays the file sizes of both files, and indicates if the files are the same size or different sizes, and which file is larger or smaller. Each test is explained in further detail.

#### *3.1.1.1 Hashing the Firmware Files.*

The process of hashing involves a bit-wise operation on fixed blocks of an entire file, producing a smaller fixed-length digest represented as hexadecimal characters [70]. In effect, hashing produces a unique signature for a given file. Different hashing algorithms have different advantages and disadvantages to their use. The hash test performed uses cryptographic hashing, since the algorithm consistently produces unique hash signatures for different inputs [70]. It is possible, though rare, that identical digests (collision) can result from hashing two different files, but with most cryptographic hashes this is not a concern since the possibility is so remote. This is not to confuse the fact that an attacker would try to create a modified file, such that hashing it would produce an identical digest to the hashed original file; this possibility is even more remote than a random collision [70,

p.219]. Using two separate hash techniques on each file ensures that this issue would be reduced to a remote possibility, and that modification detection is almost certain.

It seems that having the same SHA algorithm with different hash lengths could potentially be circumvented if a collision is found for one method, say SHA256, and produces the same resulting hash for the SHA512 method. This is not the case, since SHA256 and SHA512 have different key space sizes. A collision found for SHA256 does not produce the same collision for SHA512, though they use the same algorithm to compute the hash. An alternative method of using two different cryptographic hashing algorithms would have the same effect.

Hashing is *permutation*-sensitive, meaning that rearranging the same *combination* of opcodes in a different order produces different hashes. This property is tested in Figure 3.1 below. Four instructions are hashed, and then rearranged in a different order and hashed again. Both hashes turn out to be different. This is critical to the initial step of detecting differences in the base and suspect firmware, and later in the difference analysis process, where functions are hashed to retain their identity even if it is shifted in the suspect firmware. Matching the order of the data in both binary files is important in determining modification detection, and is proven effective using cryptographic hash functions.

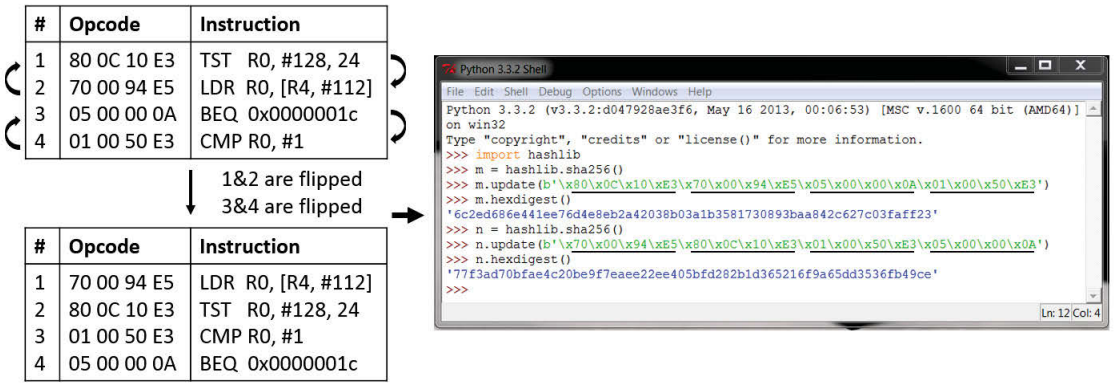



Figure 3.1: Hash Test Done With Two Permutations of Opcodes

The first comparison done with the two firmware files is a hash comparison. Two binary files are hashed separately using SHA256, their digests compared, then the files are hashed a second time using SHA512 and their digests compared again. If digests match both times, the files are identical; if they do not match even once, the suspect firmware file has been altered. This method is a quick and effective way of determining if the two files are identical without going through the entire analysis process. This test saves precious time if the files prove to be identical since no further testing would be required.

#### *3.1.1.2 File Size Comparison.*

Both baseline and suspect files are sized using bytes as the standard measurement. It seems that kilobytes is a sufficient standard measurement to compare the files, however it is foreseeable that a simple four byte addition to a larger file may be missed due to rounding. Once each file is sized separately, their sizes are compared and determined to be equal or different, where the suspect file is larger or smaller than the baseline. Knowing this information may yield clues later in determining the nature of the modification with respect to their size difference. The sizes of the firmware files is used as inputs to the opcode comparisons and functionality comparison steps later on. Assuming the hash test concludes that the suspect differs from the baseline, file size indicates only four possible characterizations: (1) modifications to the suspect only, (2) additions to the suspect, (3) deletions to the suspect, and (4) a combination of all or some additions, deletions, and modifications.

### *3.1.2 Characterizing the Nature of Modifications.*

Once modification is detected between the baseline and suspect firmware, other tests will characterize the nature of the modification in terms of opcode characteristics, opcode differences, and functionality differences. At this point, only details about the differences are ascertained. No effort is made to classify the differences as malicious or intentional. Prior to characteristic comparison, disassembling the baseline and suspect

firmware files is required. The disassembly will produce instructions or opcodes in the launguage of the microprocessor architecture, then the characteristics can be compared at the instruction level. It is believed that comparing the instructions from the baseline and suspect enables the characterization of modification. After all, executable firmware files contain a combination of instructions and data in binary format that are interpreted and executed by the microprocessor of the PLC. The characterization methods are described below.

#### *3.1.2.1 Opcode Histogram Comparison.*

Like a file's unique signature given by a digest, each firmware version in a specific architecture contains specific amounts of instructions. Not all files contain every instruction found in the instruction set; rather a file is uniquely identified by the varying amounts of instructions and amounts of each instruction type. Note that the entire binary file image is not solely assembly instructions, and contains a mixture of assembly instructions and data, such as strings. The ARM7TDMI instruction set for example contains 48 different instructions with numerous combinations of 32 bit opcodes based on the placement of registers, addresses, and conditions after the instruction [26, 40]. The opcode counts indicate the extent to which the two firmware differ. Figure 3.2 illustrates this concept.

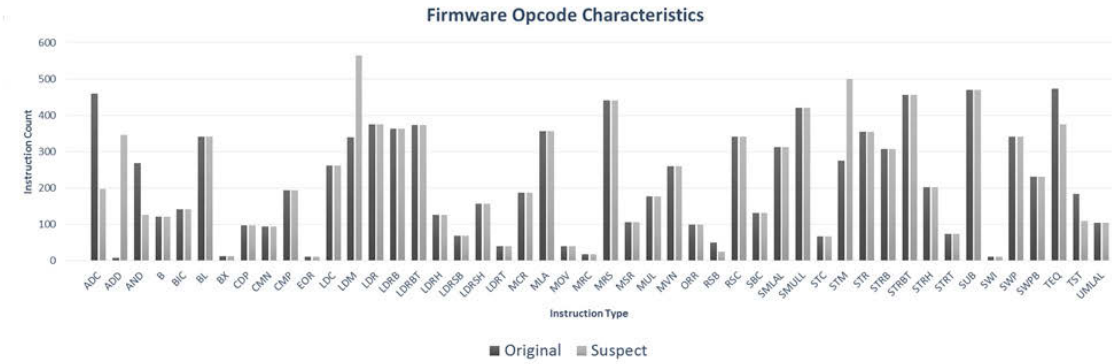



Figure 3.2: Histogram Displaying Contrasting Opcode Counts for Two Firmware Files

As each file is disassembled, a running count of each instruction decoded is tabulated. A single array holds the tabulated counts for each instruction and is representative of every instruction found in a particular instruction set. This proof of concept model only contains one instruction set, namely ARM. That means that the histogram chart represents all 48 instructions and contains a count of each instruction as the file is disassembled. Once an entire file is disassembled, the final count creates the histogram chart in Figure 3.2.

The opcode histogram comparison represents the opcode characteristics of the baseline and suspect firmware in terms of opcode combinations. The destination registers, source registers and other conditions found in the parsed instruction are not accounted for in this method but are accounted for in the opcode comparison portion. This comparison visualizes the opcode combination differences; whether it is a single instruction change, or many instruction changes. This also gives an indication of the extent of the modification made to the firmware.

The histogram bars become difficult to differentiate visually as the number of opcodes increases. As opcode counts reach into the thousands, scaling becomes a problem, and minute differences may be missed. A second histogram solves this issue by subtracting the base histogram counts from the suspect histogram counts. This histogram delta, or difference, allows a visualization of opcode differences that is scaled for improved difference-recognition. For this histogram chart, equal baseline and suspect histogram counts results in zero counts for each opcode instruction in the array. Any differences between the base and suspect will result in a positive or negative histogram bar. A positive histogram bar count indicates that the suspect has more instructions, and a negative histogram bar count indicates that the base has a higher instruction count for that particular opcode. Figure 3.3 illustrates the delta histogram.

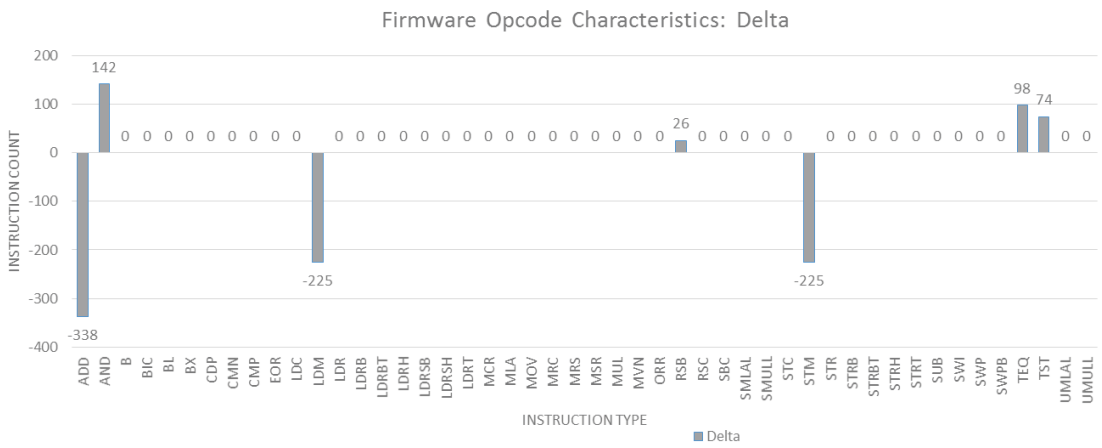



Figure 3.3: Histogram of the Difference Between Base and Suspect Histogram Counts

There are two instructions that distinguish subroutines and functions [26], and their histogram differences may indicate additions or deletions between the suspect and baseline. The Store Multiple (STM) and Load Multiple (LDM) opcodes are the prologues and epilogues to individual functions found specifically in the L61 firmware. These opcodes are used to save and restore registers on and off the stack. If there is a difference in these two specific instruction counts, it may indicate functionality differences, though their counts being identical does not counter the indication. Some functions do not contain these instructions at all.

In conjunction with this type of alteration, Branch instructions (B) are modified in order to reference added functions, or to prevent functions from being executed. If branching instructions are modified, it may indicate that an attacker used hooks to reroute functionality to their custom functions and then return the flow of control to the original function. The histogram count cannot solely determine subroutine modifications because of non-standard code structure. However, it does offer opcode characteristics comparisons for

opcode combinations, and may be used in conjunction with other comparisons to ascertain the nature of the modification.

#### *3.1.2.2 Opcode Difference Comparison.*

Comparing two files side-by-side to determine instruction differences is a straight forward comparison. If the baseline and suspect are identical, there should be no difference between each instruction at each specific address. To achieve this comparison, each file is disassembled into opcode instructions that contain specific source and destination registers, or immediate values, depending on the type of instruction. A list stores the entire disassembled instructions for each file and includes the address (e.g., 0x000008), raw four-byte hexadecimal data, opcode type (e.g., 'B') , and the entire disassembled instruction (e.g., BEQ &24C9F8). The opcode type field in the list is used in the histogram comparison done earlier. All of the list items are collected for display purposes, and the actual comparison is done between the raw bytes at each address. A separate list is kept to annotate the differences between lists, appends both lists together, and adds a True/False indication at the end of the list after the instructions are compared. This way, the display is able to highlight the differences in red based on the T/F indicator and display the information from the baseline and suspect files, side-by-side. This technique is similar to the technique used by VBinDiff, with the difference being that this technique disassembles hexadecimal bytes into opcodes and compares at the instruction-level vice byte-level.

#### *3.1.2.3 Function Difference Comparison.*

Though comparing instruction differences is straight forward, comparing functionality differences is more complex. Utilizing the same baseline list and suspect list from the disassembly, this comparison technique builds another data structure similar to a call-graph for the base and the suspect, then compares the data in a way that determines how the program flow differs. The selection of the call-graph structure along with the process for determining modifications is described below.

#### *3.1.2.4 Call Graph Structure Selection.*

Call-graphs represent the flow of a program from the main program to its functional components, between functions, and functions returning to the main program again. The code must be traversed in order to identify functions and the control flow. Once these features have been identified they must be cataloged and then compared. Branch instructions and their address references are cataloged and cross-checked between suspect and base files to determine which functions may have been modified, removed, or added. It may also be the case that subroutines, not functions were modified. Subroutines behave differently because they have simple branch instructions leading in and out of the subroutine rather than containing a prologue and epilogue. The main program also behaves like a subroutine in that it has unconditional branches to other sections of code and does not have indicators of start and stop code blocks. This challenge weighed heavily on the selection criteria of the data structure.

There were several data structures to choose from: the lattice, the graph, and the tree. The data structure selected to perform the function difference analysis for this comparison is a call-tree. This structure is selected based on the flexibility and speed it offers for firmware traversal. All three structures have a similar approach in traversing software code. Call-graphs are quite common in analyzing software. Knupfer and Nagel have discussed using complete call graphs for post-mortem analysis of software programs [32]. Their data structure utilizes a graph in order to annotate differences made to a single program over time for change analysis. A graph is predominately cyclical, meaning that a node in the graph may have children who can potentially point back to the parent node using another vertex or have children who point back to the child's parent, creating a cycle or loop. Consequently, Knupfer and Nagel only focused their research on higher level programming languages such as Java, and did not extend their work to cover low level programming languages, like the ones used to write firmware.

Li and Sun discussed combining lattices and graphs in order to conduct impact analysis on software programs [37]. Their technique uncovers unexpected and potential side effects caused by software changes. Consequently, [32, 37] both focused on high level programming languages to conduct their research. These data structures and techniques are capable of traversing high level program code because of their structured nature. Each function and subroutine have set epilogues and prologues after they are compiled. This level of detail is abstracted away since the assembly code does not need to be considered, only the syntax of the higher level language; which makes the start and stop of functions very clear and structured. This is not the case with PLC firmware or *modified* firmware. An attacker is not bound by compiler rules, and can modify opcodes to suit their purpose and evade or stifle analyzers.

Maurer eludes to this concept and offers a solution for rearranging low level code programs to become generalized structured programs [45]. He argues that high level languages abstract away the details that low level programs implement. Low level programs are difficult to represent using graphs and lattices because of their high cyclical structure, and lack of defining code structure like high level programs. He proposes representing the low level programs using tree structures that take into consideration the *loops* and replacing them with artificial nodes (dead branches) that do not cycle, and then reorganize the program using the tree in order to eliminate the cyclical flow, and continues to maintain the intended program flow. He states that:

> ...every rooted graph may be reordered in such a way that, with respect to the new order, the only reverse edges are loopbacks. This may be done by removing all loopbacks, at all levels, from a graph *G*; it is then not difficult to show that the resulting graph must be acyclic...which will be useful in developing a compact representation for loop trees. (Maurer, 2006:231)

This indicates that deleting loopbacks from a graph essentially creates a tree. Walkinshaw confirms the use of trees to represent assembly program code flow, and discusses the limits of static and dynamic analysis techniques [68]. Although it appears that trees have not been used before to conduct firmware modification analysis, the data structure is suited for assembly language level programs.

Starting with the base firmware, a tree node is created as the root and the first instruction of the firmware is inspected. Consequently, the first instruction for the L61 is a branch instruction, which immediately causes the program to populate the tree node fields for the root and initiates the call tree process. Not all firmware have this format, and may have the initial entry point further into the file. If firmware from another PLC type does not have this same format, the technique can easily be modified to traverse the file until the initial entry point is located. The branch instruction address is calculated in order to jump to the specific instruction. Figure 3.4 illustrates the creation and structure of the tree nodes and the information that each tree node contains.

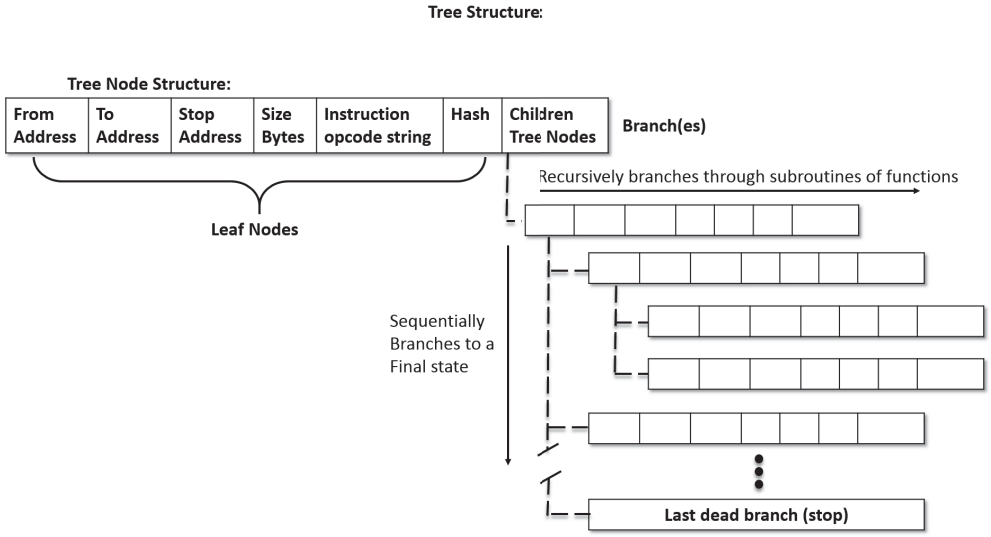



Figure 3.4: FMAT Call Tree Structure

A tree node is created when the program encounters a control flow instruction such as a Branch (B), Branch with Link (BL), or Branch and Exchange (BX) instruction. The tree node address fields TO and FROM are updated to reflect the address where the control flow instruction is calling from, and the address the instruction points to. Other fields in the tree node such as the HASH, are calculated later once the tree is completely built. Once this creation occurs another instance of the tree node function is called, becoming a recursive event. The instruction that the previous control flow instruction pointed to is read and determined if it is also a control flow instruction. If it is, another tree node is created and linked to its parent, and fields updated as before, then recursively calls another instance of the tree node function.

If the next instruction is not a control flow instruction, the program counter is incremented, and the next instruction from that function is read. Once a function is complete, it returns control back to its parent with a return instruction. This returns control to the previous tree node function and continues to step through its function until it requests a return and so on. This is equivalent to a *push* and *pop* instruction for manipulating the stack. Subroutines, however, are not bound by this structure and do not necessarily have prologues or epilogues. Subroutines may terminate with return instructions or unconditional control flow instructions, so the technique accounts for these differences by recognizing the different program flow instructions and.

In order to keep the tree acyclic, any branches that point to a previously executed part of code is given a dead branch. This dead branch is a tree node, but the program does not jump to the instruction, instead the program completes the STOP field with the same address as the FROM and START field, and updates the SIZE field with an instruction count of 1 (just the instruction itself). If cycles were allowed, the program would become trapped in an infinite loop and would crash the program. Not allowing the program to jump through previously executed code is essential for this method to work. Pilot studies indicated that

the entire program terminates when the last dead node (branch to self) is encountered. Once the last dead node is encountered, return of control is returned to the last function which terminates the tree node traversal and completes the tree.

At this point, the tree data structure is completed, with the exception of the tree nodes' HASH fields. This field is filled on a second traversal of the tree. The hash is calculated for each node by using its START and STOP fields to iterate over an existing list of instructions and updating the hash for each instruction until it reaches the STOP field. Each tree node is visited and its hash field updated.

After the tree structure and its nodes are completed, the tree data structure is *flattened* into a dictionary for faster comparison. Tree node comparisons done with a tree data structures would be time consuming due to tree traversal timing. Comparing tree structures would also be time consuming due to the complexity of comparing different tree paths. This comparison is akin to comparing apples to oranges. Instead, the tree is flattened into a dictionary, with each tree node being an item entry in the dictionary. A separate dictionary is created with the entries of the tree nodes added to it. Each node is visited starting with the root. A parent node is visited and checked if it has children. If the node has children, the node is added to the dictionary first, then the child are visited. This is done until all nodes are visited on the tree and the separate dictionary contains all of the tree nodes.

The dictionary data structure is composed of keys and values, and each tree node represents a key/value set. The key is the unique FROM fields, which is the unique address each branch instruction originated from. The values associated with the key are the values of the tree node contained in a tuple (START,HASH). The rest of the tree node fields STOP and SIZE are omitted from the value set due to constraints.

Once the baseline and suspect files are converted to trees, then flattened to dictionaries, the functionality of the base and suspect is compared. The Keys and values are compared to determine *how* the firmware has been modified at the function-level. Table 3.1 represents

the decision matrix for determining modification type based on comparing function similarities and differences (intersecting keys and difference keys).

Table 3.1: Modification Decision Matrix

| From Address | Start Address | Hash | Comparison Implication |
|---|---|---|---|
| 1 | 1 | 1 | Identical |
| 1 | 1 | 0 | Function Modified |
| 1 | 0 | 1 | Shifted (same function) |
| 1 | 0 | 0 | Control Flow Modified |
| 0 | 1 | 1 | Control Flow Modified |
| 0 | 1 | 0 | Control & Function Modified |
| 0 | 0 | 1 | Control Flow Modified & Shifted |
| 0 | 0 | 0 | Added,Deleted, or Modified * |
| 1 = Same, 0 = Different, *depending on dictionary | | | |

Rows one through four of Table 3.1 represent the intersection of keys from the base and suspect firmware. This means that control flow is initiated at the same FROM address locations. Since the FROM fields are the same, the START field is analyzed. If the START fields are the same, it indicates that they point to the same address location where the function started. The Hash field is then checked to determine if the functions are identical, with respect to the base. If they are identical, then the suspect function is identical to the base in three aspects: the same FROM-call address, the same function-START address, and the same function-HASH. Therefore, the control flow for this portion is identical, with identical functionality. If the hashes are different, it indicates that the suspect firmware has a modified function (callee) at the same start location compared to the base. Hash differences have implications that affects the decision of the analyzer later in the program. The next two comparisons from intersecting keys, reveals that the start addresses are different and are either the same function or are different functions. Both present unique cases since a shift indicates added code or added functionality somewhere else in the program. Having

different locations and different functionality indicates at a minimum that the flow has been altered for the caller function. The callee function is modified if the hash is different.

#### *3.1.2.5 Displaying Results.*

Once all the differences are categorized, they are displayed in separate side-by-side opcode comparison windows. There are a total of four displays for difference analysis. The *Difference Analysis HEX view* visually depicts the comparison of the firmware files from start to finish. The *Added functions to SUSPECT text view* displays additions to the suspect only, *Deleted functions to BASE text view* displays deletions to the base only, and the *Modified functions SUSPECT—BASE text view* displays the modifications made, comparing suspect and base side-by-side.

### *3.1.3 Scope, Assumptions, and Limitations.*

The scope of this research is limited to static firmware analysis and not dynamic analysis. Additionally, time constraints allowed for only one architecture to be explored for analysis, which is ARM7TDMI (also called ARMv4T). The physical components of the PLC such as controllers, Ethernet modules, input or output modules are not investigated. Additionally, it is outside the scope of this research to compare firmware from different manufacturers and PLC models.

Concerning assumptions, it is assumed that a PLC firmware has been compromised and extracted from the PLC for comparison to a known good baseline firmware. The baseline firmware is taken from the original manufacturer website and it is assumed that it has not been altered from the manufacturer while downloading to the test computer environments. Downloaded firmware from the manufacturer's website on different dates and then comparing them to ensure they are identical at least proves consistency of working with the same firmware from the website.

As for limitations, the software tool developed for this research is a proof-of-concept of the technique. The functionality difference comparison may not account for all possible

combinations of function or subroutine instruction epilogues or prologues. This technique along with the histogram technique provide a guide to the analyst for further investigation. Also, the disassembler makes no attempt to distinguish code from data and disassembles each four-bytes read as instructions.

## 3.2 Experiment Setup

In order to test the effectiveness of the technique outlined in the research goals a software program is written with the methods described to automate the detection and characterization process, and allows for the technique to be evaluated. The process of the experiment is reviewed, followed by a description of the environment that the experiment runs on. Technical aspects of the experiment are discussed such as the programming language used for the software program, the reason for using a custom disassembler, and how the software is tested and validated. Finally a discussion on how the firmware file is obtained is detailed. The next section covers the selection criteria for the PLC and its firmware versions.

The baseline and suspect firmware are compared using the following steps: (1) a hash comparison, (2) file size comparison, (3) fingerprint histogram comparison, (4) opcode difference comparison, and (5) function difference comparison. Each step is covered in greater detail, and the flow diagram outlining these steps are in Figure 3.5 below.

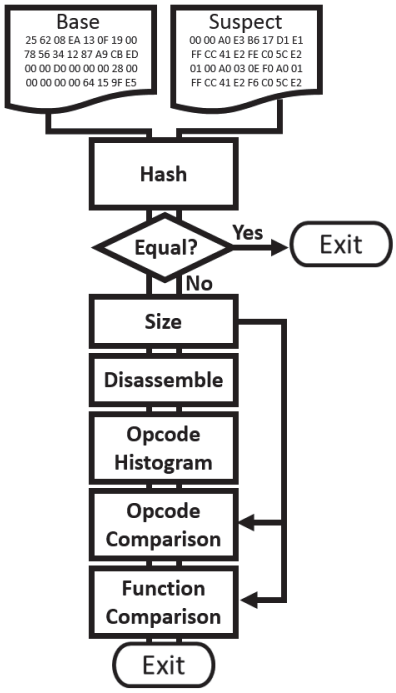



Figure 3.5: FMAT Program Flow Chart

The firmware files are saved to a computer running the Firmware Modification Analysis Tool (FMAT) software. The tool is given the target files to compare and is initiated. At that point, the tool executes the steps outlined above and proceeds to analyze and compare the firmware files. As each step is completed, information is displayed either in the standard output screen or a separate window which the program created. The program completes and analysis from the analyst may evaluate the results.

### *3.2.1 Environment.*

The testing environment for this research is done using two computers. One computer is a desktop and the other is a laptop. The desktop computer is configured with the Windows 7 Enterprise (x64) OS, Intel i7-3770 CPU processor, and 32 GB of DDR3 1333 MHz RAM. To validate the test results the same experiment is performed on a Dell Precision M4500

laptop computer configured with the Windows 7 Enterprise (x64) OS, Intel i7-M640 CPU processor, and 8 GB of DDR3 800 MHz RAM.

Python 3.3 x64 is is needed to run the FMAT software and is installed on both computers. Additional packages are installed in Python in order to display graphs in the tool:

- *matplotlib-1.3.1.win-amd64-py3.3.exe* with five dependencies:
  - *numpy-MKL-1.7.1.win-amd64-py3.3.exe*
  - *pyparsing-2.0.1.win-amd64-py3.3.exe*
  - *python-dateutil-2.1.win-amd64-py3.3.exe*
  - *pytz-2013.7.win-amd64-py3.3.exe*
  - *six-1.4.1.win-amd64-py3.3.exe*

### *3.2.2 Programming Language.*

Python 3.3 (x64) is selected as the programming language to code the FMAT software. It is anticipated that the techniques written would need validation and testing at each step, which Python is suited to handle. Python can be described as a general-purpose programming language that blends procedural, functional, and object oriented paradigms [42]. Its capabilities for graphical user interfaces (GUI), scripting, system programming, rapid prototyping, and numeric programming are a few examples of why it is selected. The code produced in Python does not need to be compiled or made into an executable to run, nor does the entire program need to be syntactically correct to run. Each function in the program can be tested separately and controlled from a separate script running in the execution environment. These features accelerated the software development process for this tool.

### *3.2.3 Using a Custom Disassembler.*

A custom disassembler is written to provide the disassembly capabilities to the analysis tool. Interfacing with IDA Pro or another software program is considered as an alternative option. However, IDA Pro is not selected. It is believed that the overhead of

the requests for each instruction would make the program inefficient. Another reason is the complexity of IDA Pro. There are many options for disassembly in IDA, and the process of a disassembly requesting is non-trivial when interfacing python with the IDC scripting language in IDA. This reinforces the belief that using IDA in conjunction with Python would increase time to the disassembly process. Interfacing Python to disassemblers written in other programming languages raises the same issues discussed above.

The disassembler is written in Python with the histogram and opcode difference comparisons in mind. Although the custom disassembler produces disassembly instructions like any other disassembler, it produces the instruction in one pass and two parts. The first part takes in four hexadecimal bytes and provides the opcode instruction type, such as 'B', while the next part produces the entire instruction, like 'BEQ &27BC9F'. This prevents additional overhead by parsing the instruction later to ascertain if it is truly a branch type instruction. One instruction that is confused with a branch using the parsing technique is a Bit Clear instruction (BIC) which performs a different operation. Parsing this incorrectly introduces false positives. At a minimum, parsings instruction strings still incurs excess overhead.

#### *3.2.4 Validation and Pilot Testing.*

Validating the software tool relies on existing tools such as IDA Pro, Radare, the Online Disassembler (ODA), VBinDiff, and HxD. Pilot testing is needed to test the feasibility of implementing the techniques and ascertain the likelihood of successful validation. Validation is performed for the disassembler, histogram comparisons, and opcode and function comparisons. For the specific attack firmware samples, coordination and validation is done with the originating author. No prior knowledge is given about the details of the firmware attacks. Once the analysis is run on the attack firmware, the details are communicated back to the author and verification of the results is compared.

#### *3.2.4.1 Validating the Disassembler.*

The custom disassembler is validated against IDA Pro, Radare, and ODA. Each opcode type for ARM7TDMI is tested, along with all conditions for each opcode type are tested. Conditions include the condition code field, destination register, source register, operand, immediate operand, shift registers, rotate registers, condition flags, and specific bit fields. Disassembly instructions from the custom disassembler are then cross checked with ODA and Radare for consistency of displaying the address operands, since IDA Pro uses additional custom address methods.

#### *3.2.4.2 Validating the Histogram Comparison.*

Histogram counts are validated with string searches done in IDA Pro, using a sample of the opcode types. For instance, three opcodes are chosen randomly for a cross comparison. The histogram count for those specific opcodes are compared with counts found in IDA Pro. Note that IDA Pro and other disassembly programs cannot distinguish code from data 100% definitively. This is a known difficult problem [69]. This affects the accuracy of IDA Pro's opcode count, and the custom disassembler's as well. Since the custom disassembler is four-byte aligned as it decodes, this provides a degree of consistency for the opcode histogram and its comparisons. Therefore, this histogram comparison method serves only as a guide.

Pilot testing for the histogram comparison revealed a scaling issue between the opcode types. It is difficult to understand opcode differences between both histogram sets with large opcode counts. The delta histogram solves this need to differentiate by displaying the histogram differences, which are significantly less that the overall opcode counts.

#### *3.2.4.3 Validating the Opcode and Function Comparisons.*

Opcode differences are validated with VBinDiff on all test cases, at the byte-level. Like the disassembler validation this is a straight forward comparison that matched results. Like the histogram comparison the function comparison is not 100% definitive. IDA Pro is

used to analyze the static program-flow for call graphs, however comparing each call graph with IDA Pro is a manual comparison and can be error prone. Additionally, the call-tree function analysis can only be as accurate as the disassembly and differentiation of code and data. Knowing that every function is correctly outlined is not exact, even for IDA Pro. Therefore, this function difference method only serves as a guide for further investigation.

## 3.3 PLC and Firmware Selection

A PLC and its firmware must be selected, in order to evaluate the effectiveness of the technique. Below the criteria is discussed for selection. A PLC is selected first, then the firmware selection follows.

### *3.3.1 PLC Selection Criteria.*

PLCs have a variable costs, but range in the thousands. The school provided several PLC options at no cost to the student. The PLCs considered are Allen Bradley, Seimens, and Omron. The Allen Bradley ControlLogix 1756-L61, B Series, Controller Module is selected based on prior research conducted by Basnight [3].

### *3.3.2 Firmware Selection Criteria.*

The L61 PLC has over a dozen firmware files to choose from the Allen Bradley website [61]. The firmware update software is available with a simple user registration. Each firmware update has a corresponding release notes which indicates when the firmware update is released. Reviewing the update release notes revealed a disparity between revisions and the dates released. Some firmware revision numbers (FRN), both major and minor, are periodically updated and their original dates are then superseded with a current release. For example, FRN 20.13 has a release date of October 2013 with a supersede date of May 2012, while FRN 16.23 has a release date of October 2013 with a supersede date of August 2013. By this comparison it seems as though FRN 16.23 is newer than FRN 20.13. Comparing the two firmware in terms of features revealed that FRN 20.13 is the

latest firmware release. Therefore major and minor revision numbers are selection criteria, and date release is not.

Three firmware versions were selected. FRN 20.13 is selected because it is the latest firmware version for the L61. FRN 19.15 is selected due to a fellow graduate student working on malware samples for the L61 firmware [63]. FRN 20.12 is selected as a minor comparison to 20.13. The major firmware comparison is performed between 20.12 and 19.15, which completes the selection of firmware samples. To review, FRN 19.15 and 20.12 are a major revision apart, while 20.12 and 20.13 are a minor revision apart. Test cases are discussed in the next chapter which will utilize these three firmware versions.

### *3.3.3 Firmware Extraction.*

There are two methods for obtaining the L61 PLC firmware file. One method involves downloading the firmware file from the manufacturer's website, and the other method involves extracting the file from the PLC using a hardware debugger. Both methods are covered by Basnight [3] in greater detail. The method chosen for this experiment is the website method since it is quicker. The process of extracting the firmware binary file is discussed, then the process of verifying the file. The next chapter covers the test cases.

The firmware file for the L61 PLC is contained in an executable installer that is downloaded from the manufacturer's website. After the firmware package is downloaded the executable must be installed to the local computer. Running ControlFlash.msi starts the installation. Another software program called RSLogix is needed to upload the file to the actual PLC. This process is skipped since uploading the firmware to the PLC is not necessary for this experiment. Besides this, the installer still installs the firmware directly to the local hard drive of the computer, regardless if RSLogix is installed or not. Once the installer finishes it reports an error and closes. Searching through the file path after the installer runs: C:\Program Files (x86)\ControlFLASH\0001\000E\0036\, yields the .bin file which is the firmware file for the L61 PLC.

To ensure that the baseline firmware file samples are unaltered from the website, each firmware version is downloaded twice on separate dates. A week was chosen to be sufficient time between downloads. Each firmware version was downloaded on different dates, and their second download was spaced one week apart for each version. Each version is extracted to a separate folder and labeled. To test that each firmware version is unaltered a hash test is performed on the same versions from different download dates. Table 3.2 displays the verification of the files with their corresponding download dates.

Table 3.2: Firmware Download Verification and Dates

| Firmware Version | Date 1 | Date 2 | Hash Difference |
|---|---|---|---|
| 19.15 | 2013-10-05 16:23:03 | 2013-10-12 09:14:52 | No |
| 20.12 | 2013-10-06 10:15:47 | 2013-10-13 11:07:01 | No |
| 20.13 | 2013-10-07 21:26:05 | 2013-10-14 14:50:16 | No |

## 3.4 Summary

This chapter provides a methodology for firmware modification analysis in PLCs. There are five techniques used to determine modifications to include hashing, file size comparison, opcode finger-printing (histogram) comparison, opcode difference comparison, and function difference comparison. These five techniques address the goals of determining if firmware has been modified, and characterizes the nature of the modification. Details regarding the experiment set up, programming language used, disassembler selection, code verification, and scope are detailed. The next chapter covers the five test cases and one control that will measure the effectiveness of the tool. These tests cases offer a range of modification conditions.

# IV. Evaluation Test Cases

This chapter describes the test cases chosen to evaluate the software tool. There are five test cases and one control case. Of the five cases one contains a single bit modification, two others are firmware attacks worked on by a fellow graduate student, and the other two cases are unmodified legitimate firmware versions (major and minor apart). The details of the test cases and the control case are discussed below.

## 4.1 Control Case

Similar to the firmware version validation test performed after the firmware package is downloaded from the manufacturer's website, the control test case baselines the FMAT detection capabilities. Two copies of the FRN 20.013 file are tested. The FMAT should detect that the firmware files are identical. The tool is designed to exit the tests if the firmware hashes match during the first test, however this functionality is suppressed to allow the tool to further analyze the other comparison functions in the tool. The histograms comparison should yield no difference in the characteristics chart and zeros across the delta chart. The opcode difference comparison should yield identical disassembled code side-by-side, and the function difference comparison should indicate that there are no modifications to the functionality of the suspect.

## 4.2 Single Bit Change Case

There is one simple modification test case to evaluate the FMAT. A single bit is changed in firmware 20.013 to test the FMATs capabilities of detecting the smallest change possible. Using HxD to make the modification, an instruction is selected at address 0x24C244. This instruction is located in the function immediately called from the initial entry point when the PLC is turned on. The opcode at the address, in little-endian format, is **50 1E 81 E3**, which is disassembled as ORR R1, R1, 0x500. One bit is changed to

the opcode such that the ORR instruction become a BIC instruction. ORR as its name suggests, performs a logical OR operation between register 1 (R1) and the value 500 (in hexadecimal), and then stores the result back to R1. Changing a single bit in the tenth binary position (Big-endian format) transforms this instruction into the BIC instruction. BIC is also a logical instruction, but different in purpose. BIC stands for Bit Clear, and performs a logical AND between R1 and the *inverted* value of 500 which is 0xFFFFAFF (32 bit fixed-width value), and the result is then stored back to R1. If this modification were successfully uploaded to the PLC, introducing this single bit change will cause latent instability in the entire program execution. Table 4.1 shows the changes made at the binary level and the change represented in hexadecimal and at the instruction level. The change made is emboldened in each view.

Table 4.1: Single Bit Change Views

| **Binary View*** | **Hex View*** | **Instruction View** |
|---|---|---|
| 11100011**0**0000010001111001010000 | E3 **8**1 1E 50 | **ORR** R1,R1, 0x500 |
| 11100011**1**0000010001111001010000 | E3 **C**1 1E 50 | **BIC** R1,R1, 0x500 |
| * Views shown in Big-endian format for easy left-to-right viewing | | |

Note that this simple change results in the firmware being rejected by the PLC if it were uploaded, since it calculates a cyclic redundancy check (CRC) and check-sum of the firmware prior to accepting it. This test ignores this fact in order to test the FMAT's sensitivity. The tool should detect that the firmware files are not identical. The hash test should indicate that a modification has occured and two sets of different hashes should be displayed. The histograms comparison should yield a difference in the characteristics chart based on the opcode being changed, and a one-for-one swap of opcodes for the delta chart. The opcode difference comparison should yield a single difference for the disassembled code side-by-side, and the function difference comparison should indicate that there is

modification made to the functionality of the suspect in terms of additions, subtractions, or modifications.

### 4.3 Firmware Attack Cases

There are two cases that represent legitimate firmware attacks. Both were written by a fellow graduate student. These modifications take the CRC and check-sum alteration into account. Both malware sample are for FRN 19.15. The first firmware attack sample bypasses the PLC's hardware diagnostic routine. No additions or deletions of code were made to this firmware, only opcode alterations were made to the original firmware. These alteration change the program flow of the program (overstepping the diagnostics) and zeroes the diagnostic routine.

A second malware sample represents an attack on the PLC's logging and reporting function. This change essentially reroutes the program to jump to another function, which has been altered. The hardware diagnostic routine is also zeroed-out. The malware payload attacks the mode switching caused by attempts to update the PLC. In order for this to happen, a physical key at the front of the PLC must be switched from *run* to *program mode* and vice-versa. The program mode routine checks for either a serial cable update, or remote update and logs the event. Originally, a hardware interrupt would have moved to the routine, however the suspect firmware modified the flow of control to jump to the altered hardware diagnostic section, which causes the program to go into an infinite loop after it recognizes that a switch has occurred multiple times. Once this threshold is reached, the malware forces an infinite loop which faults the PLC and shuts down. This attack essentially causes a denial of service.

These test cases represent possible malware created by a single attacker with limited resources, or who is part of an organization with limited resources. This attacker lacks access to the manufacturer's proprietary specifications to the firmware binary file organization, such as the ARM Binary Interface (ABI) for the L61. The tool should

determine that both cases are not identical to their baseline firmware file. The hash test should indicate that a modification has occured and two sets of different hashes should be displayed. The histograms comparison should yield a difference in the characteristics chart based on the opcode being changed, and the delta chart should display the differences in opcode changes. The opcode difference comparison should yield differences for the disassembled code side-by-side, and the function difference comparison should indicate that there is modification made to the functionality of the suspect in terms of additions, subtractions, or modifications.

### 4.4 Function Difference Cases

These two test cases evaluate the specific function-difference-comparison aspect of the FMAT. These two test cases take into consideration the fact that there is currently no malware samples available for PLCs, and that the firmware attacks may not contain enough modifications to test the function difference aspect of the tool. These test cases involve legitimate FRN comparison between minor revisions (20.013 compared to 20.012), and major revisions (19.015 compared to 20.012).

These test cases represent a comprehensive test of differences between firmware at all the test levels. It is representative of a possible PLC firmware alteration made by a Nation-State with large resources and man-power. It is conceivable that this type of threat actor would have access to knowledge of the firmware executable layouts, and tools available to compile it like the manufacturer.

The hash test should indicate that a modification has occurred and two sets of different hashes should be displayed. The size comparison may yield different file sizes and will be displayed. The histograms comparison should yield a difference in the characteristics chart based on the opcode being changed, and the delta chart should display the differences in opcode changes. The opcode difference comparison should yield differences for the disassembled code side-by-side, and the function difference comparison should indicate

that there is modification made to the functionality of the suspect in terms of additions, subtractions, or modifications.

## 4.5 Summary

This chapter covered test cases that evaluate the techniques effectiveness. Five test cases and one control case are discussed. Each test case is detailed, and focus on different aspects of the tool comparison functions. Next the results of the tests are discussed.

# V. Firmware Modification Analysis Results

In this chapter the results of the evaluation test cases are discussed. Each section covers each test performed and the results for each test case are summarized. For example, the hash comparison function is evaluated and the results for the five test cases and control case are summarized under that section.

### *5.0.1 Hash Comparison.*

Below is a summary of results for the hash comparison technique. Figure 5.1 displays the hash test performed on the single bit change to illustrate what the tool displays. Table 5.1 summarizes the results of all five tests and the control test.

```
1. HASH Suspect and Base Firmware:

        suspect SHA256 hash is: 1c813da83d7c15ee8dcd7421ecffe25e051897c4ed160c07ab724aa8d486de48
           base SHA256 hash is: a6565a58beb5a8d1c2410781c1c4bc2cff5e0287ecec3e4f068fc3258539bcd9
        suspect SHA512 hash is: 0a1e83cb361879a2fe7a3760ed109b2d8a1fd652e3ccbb427f655d128fb…146d
           base SHA512 hash is: 2051ce8cf4a50a4d419f2841ae9eef14fbfcd72831386371539ed823151…13f6

Is the Suspect firmware different than the Base (known-good) firmware? : YES
```

Figure 5.1: Hashes of Single Bit Modification

Table 5.1: Summary of Single Bit Test Results

| Base Version | Suspect | Hash Difference | Success |
|---|---|---|---|
| 19.15 | Hardware Diagnostic Attack | Yes | Yes |
| 19.15 | Logging Reporting Attack | Yes | Yes |
| 19.15 | 20.12 (Major Version) | Yes | Yes |
| 20.12 | 20.13 (Minor Version) | Yes | Yes |
| 20.13 | 20.13 Control Case | No | Yes |
| 20.13 | Single Bit Change | Yes | Yes |

The hash comparison technique of the FMAT is successful in identifying modifications made to all five test cases and the control case.

### *5.0.2 File Size Comparison.*

Below is a summary of results for the file size comparison technique. Figure 5.2 displays the file size test performed on the control case to illustrate what the tool displays. Table 5.2 summarizes the results of all five tests and the control test.

```
2. Size Comparison of Suspect and Base Firmware:

     Size of Suspect is: 2810008 bytes.
       Size of Base is: 2810008 bytes.
Suspect is same size as Base firmware.
```

Figure 5.2: File Size Comparison of Control Case

Table 5.2: Summary of File Size Test Results

| Base Version | Suspect | Base Size | Suspect Size | Detect Size Diff. |
|---|---|---|---|---|
| 19.15 | Diagnostic Attack | 2,546,464 bytes | 2,546,464 bytes | Equal |
| 19.15 | Logging Attack | 2,546,464 bytes | 2,546,464 bytes | Equal |
| 19.15 | 20.12 (Major) | 2,546,464 bytes | 2,809,528 bytes | Suspect Larger |
| 20.12 | 20.13 (Minor) | 2,809,528 bytes | 2,810,008 bytes | Suspect Larger |
| 20.13 | 20.13 Control Case | 2,810,008 bytes | 2,810,008 bytes | Equal |
| 20.13 | Single Bit Change | 2,810,008 bytes | 2,810,008 bytes | Equal |

The file size comparison technique of the FMAT is successful in comparing file sizes for all five test cases and the control case.

### *5.0.3 Opcode Histogram Comparison.*

Below is a summary of results for the opcode histogram comparison technique. Figure 5.3 displays the histogram test performed on the major revision case, and histogram delta

test performed on the minor revision case to illustrate what the tool displays. Table 5.2 summarizes the results of all five tests and the control test.

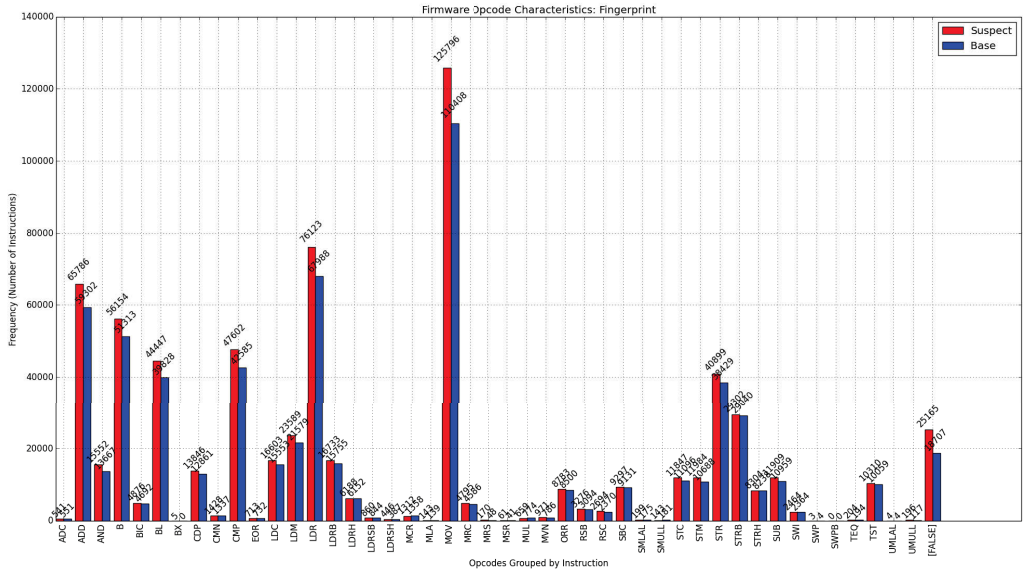



Figure 5.3: Histogram Comparisons of Firmware Major Revision

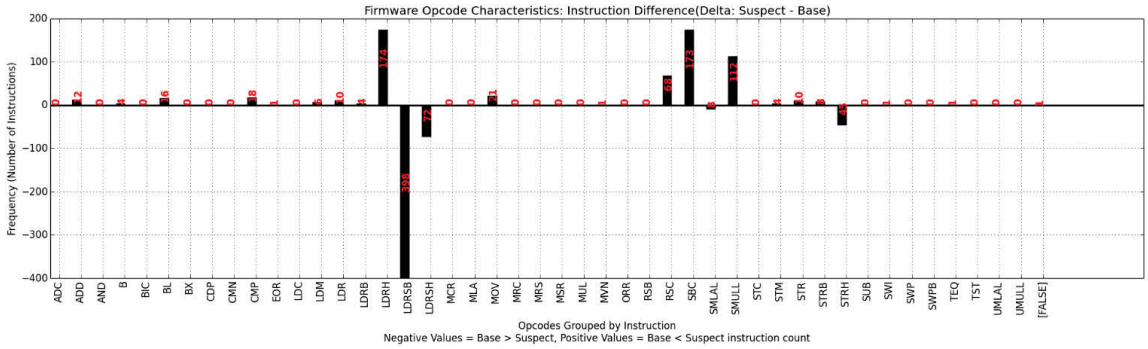



Figure 5.4: Histogram Comparisons of Firmware Minor Revision

Table 5.3: Summary of Opcode Histogram Test Results

| Base Firmware Version | Suspect | Detect Opcode Char. Shift |
|---|---|---|
| 19.15 | Diagnostic Attack | Yes (Modifications only) |
| 19.15 | Logging Attack | Yes (Modifications only) |
| 19.15 | 20.12 (Major Version) | Yes (Additions, Deletions) |
| 20.12 | 20.13 (Minor Version) | Yes (Additions, Deletions) |
| 20.13 | 20.13 Control Case | No (Equal) |
| 20.13 | Single Bit Change | Yes (Modifications only) |

The opcode histogram comparison technique of the FMAT is successful in identifying opcode characteristic changes for all five test cases and the control case. The two firmware attacks and the single bit change case had the same file size, so the modifications represented one-for-one opcode swaps. The control case did not detect a shift in opcodes which is correct, and the major and minor cases had different file size comparisons so they represented a mixed combination of additions, deletions, and one-for-one code swaps.

### *5.0.4 Opcode Difference Comparison.*

Below is a summary of results for the opcode difference comparison technique. Figure 5.4 displays the opcode difference test performed on the hardware diagnostic firmware attack, including the difference bar chart to illustrate what the tool displays. Table 5.4 summarizes the results of all five tests and the control test.

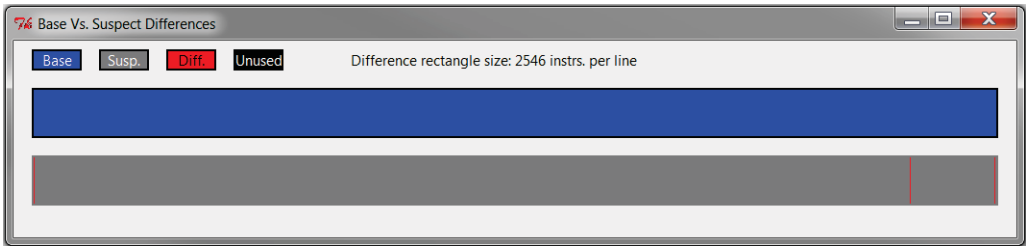
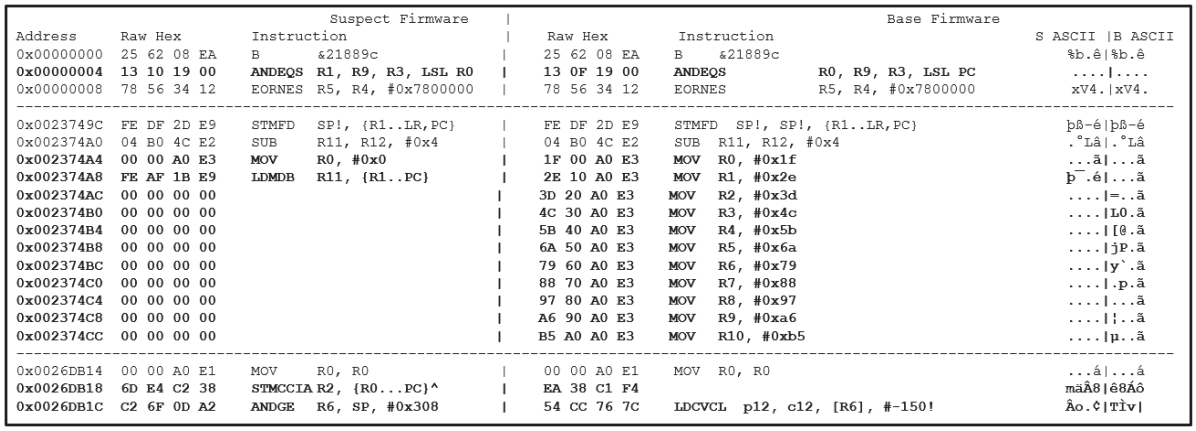



Figure 5.5: Difference Bar Comparison of Hardware Diagnostic Modification



| Suspect Firmware | | | | Base Firmware | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | Raw Hex | Instruction | | Raw Hex | Instruction | | S ASCII | B ASCII |
| 0x00000000 | 25 62 08 EA | B &21889c | | 25 62 08 EA | B &21889c | | %b.ê | %b.ê |
| **0x00000004** | **13 10 19 00** | **ANDEQS R1, R9, R3, LSL R0** | **\|** | **13 0F 19 00** | **ANDEQS R0, R9, R3, LSL PC** | | ....| .... |
| 0x00000008 | 78 56 34 12 | EORNES R5, R4, #0x7800000 | \| | 78 56 34 12 | EORNES R5, R4, #0x7800000 | | xV4.| xV4. |
| 0x0023749C | FE DF 2D E9 | STMFD SP!, {R1..LR,PC} | \| | FE DF 2D E9 | STMFD SP!, SP!, {R1..LR,PC} | | þß-é | þß-é |
| 0x002374A0 | 04 B0 4C E2 | SUB R11, R12, #0x4 | \| | 04 B0 4C E2 | SUB R11, R12, #0x4 | | .°Lâ | .°Lâ |
| **0x002374A4** | **00 00 A0 E3** | **MOV R0, #0x0** | **\|** | **1F 00 A0 E3** | **MOV R0, #0x1f** | | ...ã | ...ã |
| **0x002374A8** | **FE AF 1B E9** | **LDMDB R11, {R1..PC}** | **\|** | **2E 10 A0 E3** | **MOV R1, #0x2e** | | þ¯.é | ...ã |
| **0x002374AC** | **00 00 00 00** | | **\|** | **3D 20 A0 E3** | **MOV R2, #0x3d** | | ....| | =..ã |
| **0x002374B0** | **00 00 00 00** | | **\|** | **4C 30 A0 E3** | **MOV R3, #0x4c** | | ....| | L0.ã |
| **0x002374B4** | **00 00 00 00** | | **\|** | **5B 40 A0 E3** | **MOV R4, #0x5b** | | ....| | [@.ã |
| **0x002374B8** | **00 00 00 00** | | **\|** | **6A 50 A0 E3** | **MOV R5, #0x6a** | | ....| | jP.ã |
| **0x002374BC** | **00 00 00 00** | | **\|** | **79 60 A0 E3** | **MOV R6, #0x79** | | ....| | y`.ã |
| **0x002374C0** | **00 00 00 00** | | **\|** | **88 70 A0 E3** | **MOV R7, #0x88** | | ....| | .p.ã |
| **0x002374C4** | **00 00 00 00** | | **\|** | **97 80 A0 E3** | **MOV R8, #0x97** | | ....| | ...ã |
| **0x002374C8** | **00 00 00 00** | | **\|** | **A6 90 A0 E3** | **MOV R9, #0xa6** | | ....| | ¦..ã |
| **0x002374CC** | **00 00 00 00** | | **\|** | **B5 A0 A0 E3** | **MOV R10, #0xb5** | | ....| | µ..ã |
| 0x0026DB14 | 00 00 A0 E1 | MOV R0, R0 | \| | 00 00 A0 E1 | MOV R0, R0 | | ...á | ...á |
| **0x0026DB18** | **6D E4 C2 38** | **STMCCIA R2, {R0...PC}^** | **\|** | **EA 38 C1 F4** | | | mäÂ8 | êßÁô |
| **0x0026DB1C** | **C2 6F 0D A2** | **ANDGE R6, SP, #0x308** | **\|** | **54 CC 76 7C** | **LDCVCL p12, c12, [R6], #-150!** | | Âo.¢ | Tìv | |

Figure 5.6: Side-by-side Opcode Comparison of Hardware Diagnostic Modification

Table 5.4: Summary of Opcode Difference Test Results

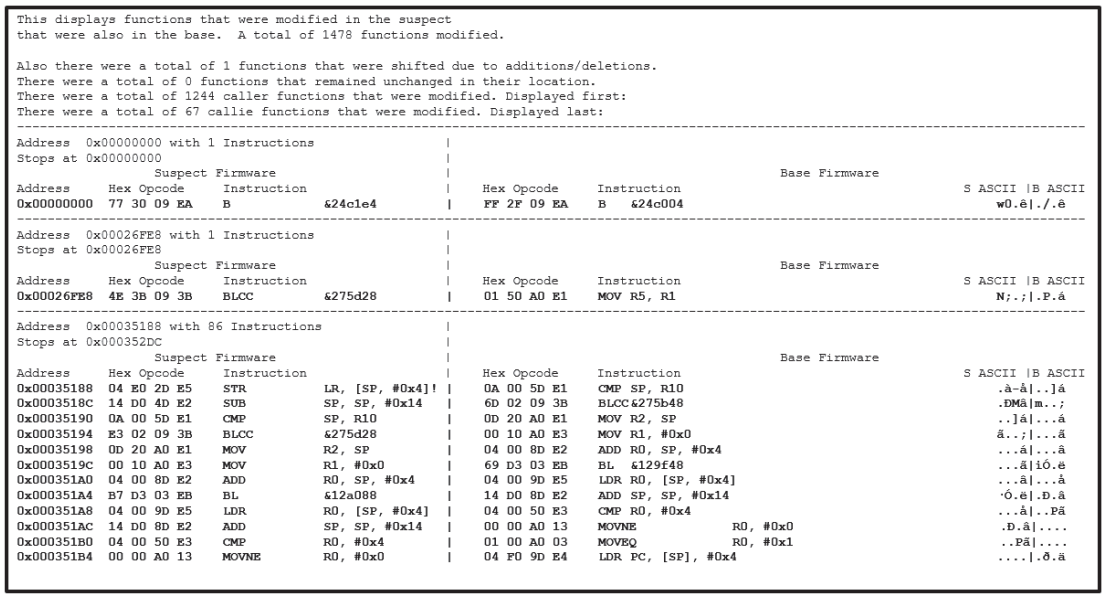
| Base Firmware Version | Suspect | Detect Opcode Differences |
|---|---|---|
| 19.15 | Diagnostic Attack | Yes (3 areas) |
| 19.15 | Logging Attack | Yes (4 areas) |
| 19.15 | 20.12 (Major Version) | Yes (Entire File) |
| 20.12 | 20.13 (Minor Version) | Yes (Entire File) |
| 20.13 | 20.13 Control Case | No |
| 20.13 | Single Bit Change | Yes (1 area) |

The opcode difference comparison technique of the FMAT is successful in identifying opcode differences at each address for all five test cases and the control case. The two firmware attacks and the single bit change case had opcode differences at a few locations, whereas the major and minor cases had differences throughout the entire file. The control case did not have any differences, and the tool was able to display no difference.

### *5.0.5 Opcode Functionality Comparison.*

Below is a summary of results for the function difference technique. Figure 5.5 displays the opcode function test performed on the minor firmware revision, to including

the modifications, additions and subtractions windows to illustrate what the tool displays. Table 5.5 summarizes the results of all five tests and the control test.



```
This displays functions that were modified in the suspect
that were also in the base.  A total of 1478 functions modified.

Also there were a total of 1 functions that were shifted due to additions/deletions.
There were a total of 0 functions that remained unchanged in their location.
There were a total of 1244 caller functions that were modified. Displayed first:
There were a total of 67 callie functions that were modified. Displayed last:
```

Address 0x00000000 with 1 Instructions — Stops at 0x00000000

| Suspect Firmware | | | | Base Firmware | | | |
|---|---|---|---|---|---|---|---|
| Address | Hex Opcode | Instruction | | Hex Opcode | Instruction | S ASCII | B ASCII |
| 0x00000000 | 77 30 09 EA | B | &24c1e4 | FF 2F 09 EA | B &24c004 | w0.ê | ./.ê |

Address 0x00026FE8 with 1 Instructions — Stops at 0x00026FE8

| Suspect Firmware | | | | Base Firmware | | | |
|---|---|---|---|---|---|---|---|
| Address | Hex Opcode | Instruction | | Hex Opcode | Instruction | S ASCII | B ASCII |
| 0x00026FE8 | 4E 3B 09 3B | BLCC | &275d28 | 01 50 A0 E1 | MOV R5, R1 | N;.; | .P.á |

Address 0x00035188 with 86 Instructions — Stops at 0x000352DC

| Suspect Firmware | | | | Base Firmware | | | |
|---|---|---|---|---|---|---|---|
| Address | Hex Opcode | Instruction | | Hex Opcode | Instruction | S ASCII | B ASCII |
| 0x00035188 | 04 E0 2D E5 | STR | LR, [SP, #0x4]! | 0A 00 5D E1 | CMP SP, R10 | .à-å | ..]á |
| 0x0003518C | 14 D0 4D E2 | SUB | SP, SP, #0x14 | 6D 02 09 3B | BLCC &275b48 | .ÐMã | m..; |
| 0x00035190 | 0A 00 5D E1 | CMP | SP, R10 | 0D 20 A0 E1 | MOV R2, SP | ..]á | ...á |
| 0x00035194 | E3 02 09 3B | BLCC | &275d28 | 00 10 A0 E3 | MOV R1, #0x0 | ã..; | ...ã |
| 0x00035198 | 0D 20 A0 E1 | MOV | R2, SP | 04 00 8D E2 | ADD R0, SP, #0x4 | ... | ...ã |
| 0x0003519C | 00 10 A0 E3 | MOV | R1, #0x0 | 69 D3 03 EB | BL &129f48 | ...ã | iÓ.ë |
| 0x000351A0 | 04 00 8D E2 | ADD | R0, SP, #0x4 | 04 00 9D E5 | LDR R0, [SP, #0x4] | ...ã | ...å |
| 0x000351A4 | B7 D3 03 EB | BL | &12a088 | 14 D0 8D E2 | ADD SP, SP, #0x14 | ·Ó.ë | .Ð.â |
| 0x000351A8 | 04 00 9D E5 | LDR | R0, [SP, #0x4] | 04 00 50 E3 | CMP R0, #0x4 | ...å | ..Pã |
| 0x000351AC | 14 D0 8D E2 | ADD | SP, SP, #0x14 | 00 00 A0 13 | MOVNE R0, #0x0 | .Ð.â | .... |
| 0x000351B0 | 04 00 50 E3 | CMP | R0, #0x4 | 01 00 A0 03 | MOVEQ R0, #0x1 | ..Pã | .... |
| 0x000351B4 | 00 00 A0 13 | MOVNE | R0, #0x0 | 04 F0 9D E4 | LDR PC, [SP], #0x4 | ....\| | .ð.ä |



Figure 5.7: Modification Comparison of Firmware Minor Revision

Figure 5.8: Addition and Subtraction Comparison Views

Table 5.5: Summary of Function Difference Test Results

| Base Firmware Version | Suspect | Detect Function Differences |
|---|---|---|
| 19.15 | Diagnostic Attack | No (Fail) |
| 19.15 | Logging Attack | No (Fail) |
| 19.15 | 20.12 (Major Version) | Yes (1478) |
| 20.12 | 20.13 (Minor Version) | Yes (1478) |
| 20.13 | 20.13 Control Case | No (Success) |
| 20.13 | Single Bit Change | Yes (1) |

The function difference comparison technique of the FMAT is partially successful in identifying function differences for four out of the five test cases, including the control case. The two firmware attacks were not successfully detected. Clues were learned from the graduate student about the nature of the failure. The modifications made to the attack firmware are not accounted for in the call tree. This means that there are functions in firmware that are not called by the main program. This highlights a limitation of the function difference process using just the call tree analyzer. The L61 PLC instead utilizes registers to jump from the main program to the hardware diagnostic attack and the logging attack, which is the reason the FMAT did not detect the change.

## 5.1 Analysis

The FMAT performed as expected with the exception of missing function differences in the firmware attack cases. Below is a table which compares metrics which summarizes feature comparisons with the reverse engineering tools mentioned in chapter 2.

Table 5.6: FMAT vs. Standard Tool Comparison

| Metric | IDA Pro | VBinDiff | HxD | FMAT |
|---|---|---|---|---|
| Syntax Comparison | None | Auto, Byte | Manual, Byte | Auto, Instruction |
| Semantic Comparison | None | None | None | None |
| Opcode Histogram | None | None | Byte-level | Instruction-level |

As seen in the results, the FMAT unifies views like the ones found in IDA Pro, VBinDiff and HxD into a single view and tool. Automating the task reduces the time required to annotate all the firmware differences with a suspect binary file, and allows analyst to shift their focus towards researching the semantic meaning of the differences. Each analysis took the computer program under two minutes to complete the analysis, and took the analyst about 20 minutes to run through all the changes and begin to understand what the changes meant. The FMAT is not a stand alone tool, but is meant to be used in conjunction with current tools to provide a complete summary of the differences between two firmware files. As with all tools, the FMAT's limitations stem from the uniqueness of the code structure for PLCs and limits of static analysis.

## 5.2 Summary

In this chapter, the FMAT results are summarized to evaluate the FMAT detection technique for five test cases and one control case. The tool shows that it can successful identifying modifications made to firmware in every case and can characterize the nature of the modifications for four of the six cases given (two failed to detect function differences). Next the paper is concluded, and future work is discussed based on the set and discovered limitations.

# VI. Future Work and Conclusion

## 6.1 Conclusions

This research outlined a technique for automating modification analysis for PLC firmware. It addresses PLC security concerns dealing with firmware modification detection. This technique provided a single tool which unifies capabilities from multiple tools currently provided, introduced new techniques for viewing and analyzing modified code, and automated the syntax analysis process. The tool focused on identifying modifications made to firmware at an architecture level and did not determine if modifications were malicious, instead an objective analysis of all differences made to the assembly code is provided. The tool compares a known good baseline firmware file to a suspected, compromised firmware file and conducts a static analysis of difference test. five tests are conducted to detect and characterize the nature of the modification. These tests are the evaluated using five test cases and one control test. These five test cases and control case validated the FMAT and uncovered unexpected limitations inherent to analyzing PLC firmware. Although the function differences were not detected in the firmware attack cases the other other tests performed before it detected the opcode differences.

## 6.2 Impact

Current security auditing tools focus on protection and recovery capabilities and focus less on detection. This PLC security technique provides thorough detection capabilities for static code analysis in PLC firmware and characterizes the nature of the modification. Being able to detect static code modifications sets the foundation for more precise analysis techniques and thorough protection and recovery capabilities. Automating the analysis process and providing a singular difference view reduces the skill set and tools needed to perform syntax analysis, while reducing the errors made in the analysis process. The

FMAT performs this code difference analysis in just a few minutes with a singular view. Automating the analysis task shifts the focus to semantic code analysis, which may save a forensic security teams time. Adding the unique opcode histogram view at the instruction level-instead of the hex byte level-gives meaning to the opcode differences, and provides another method of fingerprinting malware.

This technique is applicable to industrial security as a whole, and has practical uses in forensic analysis and future industrial security developments. It may also be used to conduct future research in understanding firmware version changes. Private industry and government agencies alike could benefit from using this tool to respond to current threats and is a foundation in developing dynamic analysis for firmware modification. Having this technique provides options for detecting firmware modification, and narrowing the search for other possible security breach vectors.

## 6.3 Future Work

The FMAT proved to conduct firmware modification analysis effectively. However, the scope, limitations, and constraints addressed provides a wealth of future work for improvement and the expanded research area of firmware modification analysis.

### *6.3.1 Complete Firmware Inventory.*

A limitation of the FMAT is the static analysis approach. The limits of static analysis are reached and made apparent when conducting function difference analysis. The two firmware attack cases performed on the L61 firmware revealed limited branch paths in the entire firmware file. Clues in the tests revealed that many of these drivers had functions that were not referenced from the main program and did not have return addresses to the main program either. References to addresses outside the firmware address range were also observed. Although the correct percentage of code and data for this specific PLC firmware is not shared by the manufacturer, or readily known by conducting modification analysis, the low percentage of code reached through the call trees suggests that the ladder

logic bridges the gap between the main program and its functions with the exception of interrupts. Figure 5.1 illustrates this limitation.

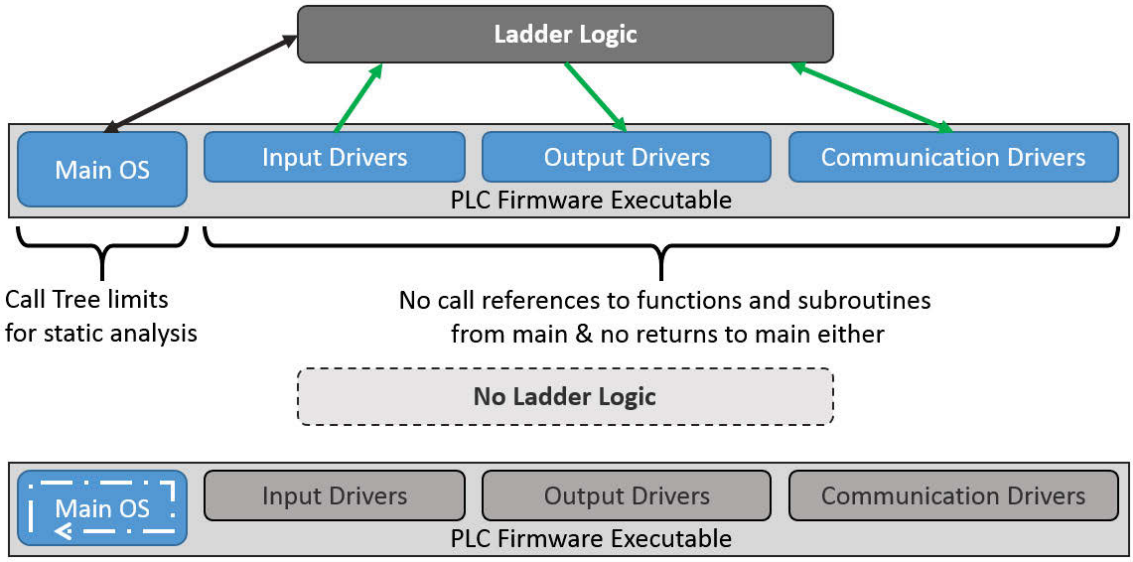



Figure 6.1: Ladder Logic and PLC Firmware Roles

As seen in the attack firmware cases, the malware is able to run without ladder logic because the attacks relied on hardware and software interrupts or register values, which were not detected by the call tree analyzer. This limitation may be overcome with additional methods that account for functions outside the call tree. Since a complete inventory of opcodes is retained in the data structures of the FMAT, it takes less effort to construct other analyzer functions. The function inventory would be complete and the function difference analysis detection would improve. Completing the function inventory improves the accuracy of detecting function differences.

### *6.3.2 Additional Architecture Disassemblers.*

This research technique utilized a single custom made disassembler (ARM7TDMI) to evaluate firmware. Future research should include multiple ARM architecture versions and

other architectures such as MIPS, PowerPC, Intel, and Motorola. Incorporating additional disassemblers would require an architecture selection option, and a best guess algorithm that selects a architecture type based on opcode heuristics. The second option would prove useful in assisting the analyst if the architecture type is unknown. Sickendick [64] proposed that Kolter and Maloofs boosted decision tree algorithm [33] is most efficient at determining architecture types and should be considered.

### *6.3.3 Classifying Code vs. Data.*

Another area of consideration is the automation of classifying code from data. As mentioned previously, the FMAT tool does not attempt to distinguish code from data, but takes advantage of the fixed four byte opcode/data alignment to decode the entire firmware file as-is. An experienced analyst can decipher portions of code from data and use the ASCII character window to aide in making decisions, but it is not error-proof. Code classification becomes crucial when attempting to correctly disassemble variable length opcodes found in Complex Instruction Set Computing (CISC) architectures such as Intel, since the reader will have to step through variable lengths of opcodes. Wartell *et al.* [69] proposed a method for this type of classifications for Intel, and may prove useful in incorporating this technique along with adding multiple disassemblers. It should be noted that embedded devices use RISC architectures almost exclusively, so adding the CISC architecture would broaden the tools search beyond PLC firmware to general computer architecture platforms, or future PLC platforms that would include CISC architectures.

### *6.3.4 Incorporating Thumb Instruction Analysis.*

The FMAT limited its instruction analysis and disassembly to regular fixed-length (32 bit) ARM instructions and did not analyze or disassemble 16 bit fixed-length Thumb instructions. This process requires a separate pass through the file to analyze potential Thumb instructions. This step is necessary for completing the static analysis capability of

the FMAT and the ARM disassembler, and provides the needed foundation for dynamic analysis.

#### *6.3.5 Dynamic Firmware Modification Analysis.*

The static differences in code were made readily apparent, but analysis of the semantic differences is left to the analyst to evaluate. Dynamic analysis is needed to understand the semantics of firmware modification. A scenario is proposed to build a test case for determining behavioral differences between a suspect and base firmware. First a specific ICS system must be selected and engineered to specification, unless it is able to be simulated through software means. A working PLC base firmware and ladder logic program is chosen based on the scenario and data is then collected which captures the time and states of the PLC and the devices attached to it. The system under test would be the dynamic firmware modification analyzer which records a baseline pattern of static and dynamic data. Later an attack firmware and identical ladder logic program is recorded and then analyzed by the dynamic firmware modification analyzer to detect the dynamic changes made to the firmware. The dynamic firmware modification analyzer coupled with the static firmware analyzer create a picture which is compared for syntax and semantic differences. This would address the shortfalls of the static analysis and provide an automated approach for understanding semantic behavior differences. Additional information regarding semantic and dynamic analysis is discussed in Christodorescu *et al.* papers [13, 24, 57].

### 6.4 Concluding Remarks

Detecting security threats is the cornerstone of information assurance. This research demonstrates a technique that aides analysts in finding differences between altered firmware and a known good firmware baseline quickly and efficiently. Automation of complex tasks, while keeping the analysis simple and straightforward, ensures that a variety of experience-level analysts can use this technique effectively. More critical is the presentation of the

firmware modification analysis technique as a foundation for further detailed and complex analysis techniques that focus on semantics rather than syntax.

## Bibliography


[1] Abrams, Marshall and Joe Weiss. "Malicious Control System Cyber Security Attack Case Study–Maroochy Water Services, Australia". *McLean, VA: The MITRE Corporation*, 2008.

[2] Allen-Bradley. "DF1 Protocol and Command Set Reference Manual". 1996.

[3] Basnight, Zachry, Jonathan Butts, Juan Lopez Jr, and Thomas Dube. "Firmware Modification Attacks on Programmable Logic Controllers". *International Journal of Critical Infrastructure Protection*, 6:76–84, 2013.

[4] Bencsáth, Boldizsár, Gábor Pék, Levente Buttyán, and Márk Félegyházi. "Duqu: Analysis, detection, and lessons learned". *ACM European Workshop on System Security (EuroSec)*, volume 2012. 2012.

[5] Bolton, William. *Programmable logic controllers*. Newnes, 2009.

[6] Boyer, Stuart. *SCADA: Supervisory Control and Data Aquisition 4th Edition*. International Society of Automation, NC, USA, 2010.

[7] Bronk, Christopher and Eneken Tikk-Ringas. "The Cyber Attack on Saudi Aramco". *Survival*, 55(2):81–96, 2013.

[8] Burmester, Mike, Emmanouil Magkos, and Vassilis Chrissikopoulos. "Modeling security in cyber–physical systems". *International Journal of Critical Infrastructure Protection*, 5(3):118–126, 2012.

[9] Byres, Eric, David Leversage, and Nate Kube. "Security incidents and trends in SCADA and process industries". *The Industrial Ethernet Book*, 39(2):12–20, 2007.

[10] Carcano, Andrea, Alessio Coletta, Michele Guglielmi, Marcelo Masera, Igor Nai Fovino, and Alberto Trombetta. "A multidimensional critical state analysis for detecting intrusions in SCADA systems". *Industrial Informatics, IEEE Transactions on*, 7(2):179–186, 2011.

[11] Cardenas, Alvarado A., Tanya Roosta, and Shankar Sastry. "Rethinking security properties, threat models, and the design space in sensor networks: A case study in SCADA systems". *Ad Hoc Networks*, 7(8):1434–1447, 2009.

[12] Chandia, Rodrigo, Jesus Gonzalez, Tim Kilpatrick, Mauricio Papa, and Sujeet Shenoi. "Security strategies for SCADA networks". *Critical Infrastructure Protection*, 117–131. Springer, 2007.

[13] Christodorescu, Mihai, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. "Semantics-aware malware detection". *Security and Privacy, 2005 IEEE Symposium on*, 32–46. IEEE, 2005.

[14] Coates, Gregory M, Kenneth M Hopkinson, Scott R Graham, and Stuart H Kurkowski. "A trust system architecture for SCADA network security". *Power Delivery, IEEE Transactions on*, 25(1):158–169, 2010.

[15] Cui, Ang, Michael Costello, and Salvatore J Stolfo. "When firmware modifications attack: A case study of embedded exploitation".

[16] Dacey, Robert F. "Critical Infrastructure Protection: Challenges and efforts to secure control systems (Testimony Before the Subcommittee on Technology Information Policy, Intergovernmental Relations and the Census, House Committee on Government Reform)". 2004.

[17] Dehlawi, Zakariya and Norah Abokhodair. "Saudi Arabia's response to cyber conflict: A case study of the Shamoon malware incident". *Intelligence and Security Informatics (ISI), 2013 IEEE International Conference on*, 73–75. IEEE, 2013.

[18] Department of Defense Directive 8500.01E. "Information Assurance", April 2007.

[19] Department of Defense Instruction 8500.2. "Information Assurance Implementation", February 2003.

[20] Department of Homeland Security. *National Infrastructure Protection Plan*. Technical report, 2009.

[21] Eilam, Eldad. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[22] Erickson, Jon. *Hacking: The art of exploitation*. No Starch Press, 2008.

[23] Falliere, N., L.O. Murchu, and E. Chien. "W32. stuxnet dossier". *White paper, Symantec Corp., Security Response*, 2011.

[24] Fredrikson, Matt, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. "Synthesizing near-optimal malware specifications from suspicious behaviors". *Security and Privacy (SP), 2010 IEEE Symposium on*, 45–60. IEEE, 2010.

[25] Hex-Rays. "Interactive Disassembler Professional", October 2013.

[26] Hohl, William. *ARM Assembly Language: Fundamentals and Techniques*. CRC Press, Inc., 2009.

[27] of Homeland Security, Department. "ICS-CERT Monitor Newsletter". December 2013.

[28] Horz, Mael. "HxD – Freware hex editor and disk editor", 2013.

[29] Igure, Vinay M., Sean A. Laughter, and Ronald D. Williams. "Security issues in SCADA networks". *Computers and Security*, 25(7):498–506, 2006.

[30] Khan, Arindam, Mukesh Kr. Sharma, G Ganesh, SD Dhodapkar, BB Biswas, and RK Patil. "A cryptographic primitive based authentication scheme for run-time software of embedded systems". *Reliability, Safety and Hazard (ICRESH), 2010 2nd International Conference on*, 500–504. IEEE, 2010.

[31] Kilman, Dominique and Jason Stamp. "Framework for SCADA security policy". *Sandia National Laboratories report SAND2005-1002C*, 2005.

[32] Knupfer, Andreas and Wolfgang E. Nagel. "Construction and compression of complete call graphs for post-mortem program trace analysis". *Parallel Processing, 2005. ICPP 2005. International Conference on*, 165–172. 2005.

[33] Kolter, J Zico and Marcus A Maloof. "Learning to detect and classify malicious executables in the wild". *The Journal of Machine Learning Research*, 7:2721–2744, 2006.

[34] Kuipers, David and Mark Fabro. *Control systems cyber security: Defense in depth strategies*. United States Department of Energy, 2006.

[35] Kuvshinkova, S. "SQL Slammer worm lessons learned for consideration by the electricity sector". *North American Electric Reliability Council*, 2003.

[36] Leverett, Eireann P. "Quantitatively assessing and visualising industrial system attack surfaces". *University of Cambridge, Darwin College*, 2011.

[37] Li, Bixin, Xiaobing Sun, and Hareton Leung. "Combining concept lattice with call graph for impact analysis". *Advances in Engineering Software*, 53:1–13, 2012.

[38] Lieberman, Joseph. "Floor Statement for Sen. Joseph Lieberman, Introduction of Cybersecurity Act of 2012, Washington, DC, February 14, 2012", 2012.

[39] Lim, Kyung-Soo and Sangjin Lee. "A methodology for forensic analysis of embedded systems". *Future Generation Communication and Networking, 2008. FGCN'08. Second International Conference on*, volume 2, 283–286. IEEE, 2008.

[40] Limited, ARM. "ARM7TDMI Instruction Set", 2010.

[41] Lippmann, Richard, Kyle Ingols, Chris Scott, Keith Piwowarski, Kendra Kratkiewicz, Mike Artz, and Robert Cunningham. "Validating and restoring defense in depth using attack graphs". *Military Communications Conference, 2006. MILCOM 2006. IEEE*, 1–10. IEEE, 2006.

[42] Lutz, Mark. *Learning python*. " O'Reilly Media, Inc.", 2013.

[43] Madsen, Christopher J. "Visual Binary Difference", March 2013.

[44] Matrosov, Aleksandr, Eugene Rodionov, David Harley, and Juraj Malcho. "Stuxnet under the microscope". *ESET LLC (September)*, 2010.

[45] Maurer, Ward Douglas. "Generalized structured programs and loop trees". *Science of Computer Programming*, 67(2):223–246, 2007.

[46] Maynor, David and Robert Graham. "SCADA security and terrorism: Were not crying wolf". *Black Hat*, 2006.

[47] McGraw, Gary. "Software security". *Security & Privacy, IEEE*, 2(2):80–83, 2004.

[48] McGraw, Gary. "The 7 Touchpoints of Secure Software". 2005.

[49] McMinn, Lucille, Jonathan Butts, David Robinson, and Billy Rios. "Exploiting the critical infrastructure via nontraditional system inputs". *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*, 57. ACM, 2011.

[50] McMinn, Lucille R. *External Verification of SCADA System Embedded Controller Firmware*. thesis, Air Force Institute of Technology, March 2012.

[51] Morris, Thomas, Anurag Srivastava, Bradley Reaves, Wei Gao, Kalyan Pavurapu, and Ram Reddi. "A control system testbed to validate critical infrastructure protection concepts". *International Journal of Critical Infrastructure Protection*, 4(2):88–103, 2011.

[52] Morrisett, Greg. "Attacking Malicious Code: A Report to the Infosec Research Council". *IEEE Software*, 14(3):342–351, 2000.

[53] Munro, Kate. "Deconstructing Flame: the limitations of traditional defences". *Computer Fraud & Security*, 2012(10):8–11, 2012.

[54] Nai Fovino, Igor, Andrea Carcano, Marcelo Masera, and Alberto Trombetta. "An experimental investigation of malware attacks on SCADA systems". *International Journal of Critical Infrastructure Protection*, 2(4):139–145, 2009.

[55] Papa, Stephen, William Casper, and Suku Nair. "Placement of trust anchors in embedded computer systems". *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, 111–116. IEEE, 2011.

[56] Pollet, Jonathan. "Developing a solid SCADA security strategy". *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, 148–156. IEEE, 2002.

[57] Preda, Mila Dalla, Mihai Christodorescu, Somesh Jha, and Saumya Debray. "A semantics-based approach to malware detection". *ACM SIGPLAN Notices*, volume 42, 377–388. ACM, 2007.

[58] Ralston, Jim. “From VHF to wireless ethernet: modernization without compromise”, February 2005.

[59] Reeves, J., A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith. “Intrusion detection for resource-constrained embedded control systems in the power grid”. *International Journal of Critical Infrastructure Protection*, 5(2):74–83, 2012.

[60] Robles, Rosslin John and Min-Kyu Choi. “Symmetric-Key Encryption for Wireless Internet SCADA”. *Security Technology*, 289–297. Springer, 2009.

[61] Rockwell Automation. “Firmware Updates”, 2013. URL http://www.rockwellautomation.com/support/firmware/overview.page.

[62] Rowe, Dale C, Barry M Lunt, and Joseph J Ekstrom. “The role of cyber-security in information technology education”. *Proceedings of the 2011 conference on Information technology education*, 113–122. ACM, 2011.

[63] Schuett, Carl. *Programmable logic controller modification attacks for use in detection analysis*. thesis, Air Force Institute of Technology, March 2014.

[64] Sickendick, Karl A. *File carving and malware identification algorithms applied to firmware reverse engineering*. thesis, Air Force Institute of Technology, March 2013.

[65] Slay, Jill and Michael Miller. *Lessons learned from the maroochy water breach*. Springer, 2007.

[66] Symantec. “The Shamoon Attacks Continue”, 2012. URL http://www.symantec.com/connect/blogs/shamoon-attacks-continue.

[67] Thompson, Mark. “Panetta Sounds Alarm on Cyber-War Threat”, October 2012. URL http://nation.time.com/2012/10/12/panetta-sounds-alarm-on-cyber-war-threat/.

[68] Walkinshaw, Neil. “Reverse-Engineering Software Behavior”. *Advances in computers*, 91:1–58, 2013.

[69] Wartell, Richard, Yan Zhou, Kevin Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. “Differentiating code from data in x86 binaries”. *Machine Learning and Knowledge Discovery in Databases*, 522–536. Springer, 2011.

[70] Washington, Lawrence C and Wade Trappe. *Introduction to cryptography: with coding theory*. Prentice Hall, 2006.

[71] Xiao, Kun. “Retrofitting Cyber Physical Systems for Survivability through External Coordination”. 465–473. 2008.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704–0188*



| 1. REPORT DATE *(DD–MM–YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From — To)* |
|---|---|---|
| 27–03–2014 | Master's Thesis | Oct 2013–Mar 2014 |

**4. TITLE AND SUBTITLE**

Firmware Modification Analysis
In Programmable Logic Controllers

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

Garcia Jr., Arturo M., Captain, USA

**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-14-M-32

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Department of Homeland Security ICS-CERT
POC: Nick Carr, DHS ICS-CERT Technical Lead
ATTN: NPPDCS&CNCSDUS-CERT
Mailstop: 0635, 245 Murray Lane, SW, Bldg 410, Washington DC 20528
Email: ics-cert@dhs.gov; Phone: 1-877-776-7585

**10. SPONSOR/MONITOR'S ACRONYM(S)**

DHS ICS-CERT

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**13. SUPPLEMENTARY NOTES**



**14. ABSTRACT**

Incorporating security in supervisory control and data acquisition (SCADA) systems and sensor networks has proven to be a pervasive problem due to the constraints and demands placed on these systems. Both attackers and security professionals seek to uncover the inherent roots of trust in a system to achieve opposing goals. With SCADA systems, a battle is being fought at the *cyber–physical* level, specifically the programmable logic controller (PLC). The Stuxnet worm, which became increasingly apparent in the summer of 2010, has shown that modifications to a SCADA system can be discovered on infected engineering workstations on the network, to include the ladder logic found in the PLC. However, certain firmware modifications made to a PLC can go undetected due to the lack of effective techniques available for detecting them.

Current software auditing tools give an analyst a singular view of assembly code, and binary difference programs can only show simple differences between assembly codes. Additionally, there appears to be no comprehensive software tool that aids an analyst with evaluating a PLC firmware file for modifications and displaying the resulting effects. Manual analysis is time consuming and error prone. Furthermore, there are not enough talented individuals available in the industrial control system (ICS) community with an in-depth knowledge of assembly language and the inner workings of PLC firmware.

This research presents a novel analysis technique that compares a suspected-altered firmware to a known good firmware of a specific PLC and performs a static analysis of differences. This technique includes multiple tests to compare both firmware versions, detect differences in size, and code differences such as removing, adding, or modifying existing functions in the original firmware. A proof-of-concept experiment demonstrates the functionality of the analysis tool using different firmware versions from an Allen-Bradley ControlLogix L61 PLC.

**15. SUBJECT TERMS**

Firmware, Modification, Analysis, Detection, PLC Security

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | Dr. Robert F. Mills (ENG) |
| U | U | U | UU | 93 | 19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4527 robert.mills@afit.edu |