SYNCHRONIZATION ALGORITHMS FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

THESIS

Elwyn J. McCargar, 2d Lt, USAF

AFIT-ENG-MS-17-M-050

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-050

SYNCHRONIZATION ALGORITHMS FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

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 Computer Science

Elwyn J. McCargar, B.S.C.S.

2d Lt, USAF

March 2017

AFIT-ENG-MS-17-M-050

SYNCHRONIZATION ALGORITHMS FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

THESIS

Elwyn J. McCargar, B.S.C.S.
2d Lt, USAF

Committee Membership:

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

Chair

LTC Mason J. Rice, Ph.D.

Member

Juan Lopez Jr., Ph.D.

Member

Abstract

Nations rely heavily on critical infrastructures to be the backbone of both the economy and the nation's general well-being. Many of these critical infrastructures depend on Industrial Control Systems (ICS). ICS is a general term used to describe an interaction where data is received from sensors and then actions are taken based off the data received. Most ICSs were designed and implemented before the Internet became widely used. When some systems were finally connected to the Internet, the emphasis was on the integrity of the data sent and the availability between devices, not the confidentiality. This lack of confidentiality is why ICS security is the focus of this research.

The problem this research investigates is finding an optimal Synchronization Algorithm used to synchronize a back-end Programmable Logic Controller (PLC) with a honeypot in the ScriptGenE Framework, which is a honeypot solution to ICS security. A honeypot is designed to emulate a desired device on the network and can be a superficial emulation or a complete replication of the desired device.

This research uses the ScriptGenE Framework as a Hybrid Honeypot solution. ScriptGenE is a Hybrid Honeypot because each instance of the honeypot is only a superficial emulation with a finite amount of data about the device it is emulating. However, each honeypot is connected to a back-end PLC, identical to the device being emulated for the ability to provide more authentic responses than just a superficial emulation. Synchronization between the back-end PLC and the honeypot is important for the emulation of a device's protocol because some protocols are stateful and provide responses to requests based on previous requests. The honeypot needs to ensure the back-end PLC is in the same state before sending a request to the PLC.

This research hypothesizes: selecting a Synchronization Algorithm for the ScriptGenE Framework based on the shape and size of the protocol tree will yield faster latency response times and a smaller load on the back-end PLC than the existing configuration.

This research answers the hypothesis by creating a simulation of interactions between a honeypot and a client. There are four Synchronization Algorithms under test: Catchup, Lockstep, Latelock, and Triggerlock. These four algorithms are each tested against eleven protocols. Three of these protocols: Hypertext Transfer Protocol (HTTP), EtherNet/IP (ENIP), and STEP7 are used by industrial PLCs. The other eight protocols are created for the purpose of this research to simulate possible protocols other than HTTP, ENIP, and STEP7.

Through a full factorial experiment, it is shown for 91% of the protocols the best algorithm is Triggerlock. The only exception is the stateless protocol HTTP, where the best algorithm is Catchup because of the honeypot's instant disconnection from the back-end PLC after receiving the response to its request.

This research helps further improve and increase the functionality of the ScriptGenE Framework. ScriptGenE now analyzes the protocol tree as part of the initial honeypot setup process. The changes made to ScriptGenE based on this research improve the software by decreasing the latency times and decreasing the load on the back-end PLC.

Acknowledgements

I would like to thank Capt Phillip Warner and Mr. Kyle Girtz for the amount of previous work they put in to their research. Their work helped to make this research possible.

I would also like to thank my advisor Dr. Mullins for giving me the tools I needed to earn my graduate degree.

Finally, I would like to thank 2d Lt Anthony Portante, 2d Lt Jon Goodgion, and 2d Lt Justin Gallenstein for their advice and peer reviews.

Elwyn J. McCargar

Table of Contents

Page

Page

Page

List of Figures

List of Tables

List of Acronyms

CSV Comma-Separated Values 44

CIP Common Industrial Protocol 7

CUT Component Under Test 32

DMZ Demilitarized Zone 12

DoS Denial of Service 11

ENIP EtherNet Industrial Protocol 4

FTP File Transfer Protocol 23

HI High-Interaction 17

HMI Human-Machine Interface 7

HTML HyperText Markup Language 15

HTTP HyperText Transfer Protocol 4

ICS Industrial Control Systems 1

IDS Intrusion Detection System 7

IED Intelligent Electronic Device 9

IP Internet Protocol 1

IT Information Technology 10

LI Low-Interaction 18

MAC Media Access Control 22

MTU Master Terminal Unit 9

NIST National Institute of Standards and Technology 10

OS Operating System 21

pcap Packet Capture 1

PLC Programmable Logic Controller 1

RTU Remote Terminal Unit 9

SCADA Supervisory Control and Data Acquisition 4

SUT System Under Test 32

TCP Transmission Control Protocol 16

VM Virtual Machine 20

WAN Wide Area Network 9

SYNCHRONIZATION ALGORITHMS FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

I. Introduction

1.1 Background

The problem this research investigates is finding an optimal Synchronization Algorithm used to synchronize a back-end Programmable Logic Controller (PLC) with a honeypot in the ScriptGenE Framework. The ScriptGenE Framework is a honeypot solution to Industrial Control Systems (ICS) security. A honeypot is designed to look like a desired device on the network and can be a superficial emulation or a complete replication of the desired device. The superficial emulation is designed to emulate the device in network scans by having the same ports open and having the same protocol running on those ports. In the complete replica emulation, two identical devices are used. The first device is used as the production device on the network. The second device is used as the honeypot.

The program used in this research, ScriptGenE, is designed to be protocol agnostic. It accepts a network traffic Packet Capture (pcap) file and an Internet Protocol (IP) address target as inputs, then uses the pcap file to mirror the target's responses to client requests. The ScriptGenE Framework can be connected to a replica production device identical to the target it is emulating for a deeper level of response authenticity. In the case of this research, the honeypot's back-end production devices are PLCs. Should the honeypot receive a client request it cannot find in the network capture pcap file, the honeypot then proxies the request to the back-end PLC to receive an

authentic response from the production device. Some protocols are stateless, (e.g., the honeypot returns the same response to the same request regardless of the previous request history) and the honeypot can instantly forward unknown requests to the back-end PLC. However, for stateful protocols the honeypot must first ensure the back-end PLC is in the same state as the honeypot in order to ensure a correct response. In order to put the PLC in the same state, the honeypot must synchronize with the PLC by replaying all the client's requests to the PLC. This is why the Synchronization Algorithm is critical to the honeypot operating. Changing the Synchronization Algorithm alters the latency time between the client request and the honeypot response. It also changes the load placed on the back-end PLC used to correctly answer previously unseen client requests.

1.2 Motivation

The problem of choosing a Synchronization Algorithm for the honeypot is interesting because ScriptGenE creates a protocol tree to help in the device emulation. If a tree is defined as having n nodes and $n\text{-}1$ edges connecting the nodes, then the protocol tree can be thought of as having client requests on the edges of the tree and device/honeypot responses as nodes. Due to this protocol tree design, different protocols produce different tree structures. Because the protocol trees are different, there is a chance that some tree structures work better with one Synchronization Algorithm while other trees work better with another algorithm or a different algorithm configuration. Since the Synchronization Algorithms have the ability to impact the latency time and the back-end PLC load, even incremental improvements to latency and load can have a significant impact in a real world implementation.

1.3 Research Goals

A research hypothesis and three research goals are used to guide this research. This research hypothesizes:

> Selecting a Synchronization Algorithm based on the shape and size of the protocol tree will yield faster latency response times and a smaller load on the back-end PLC than the existing default Synchronization Algorithm setting of Latelock (Latelock is described in Section 3.5.3).

The goal for this research is to answer the following questions:

1. Given a protocol tree, is there a Synchronization Algorithm that yields minimized latency and minimized load on the back-end PLC?
2. Does the shape of the protocol tree impact the best Synchronization Algorithm?
3. Is there an optimal setting for the Triggerlock TriggerDepth based on the tree shape and size?

1.4 Approach

This research conducts a full factorial experiment based on collected network traffic. The experiment uses the network traffic pcaps of the ScriptGenE Framework to get average client request response times from two situations. The first being when the client is receiving a response from the honeypot via the internal protocol tree. The second being when the client is getting a response from the honeypot via a forwarded response from the back-end PLC. These average response times are used to calculate a response time of a randomized scenario using variously shaped protocol trees. The protocol trees used in this experiment are three actual protocols used in industrial

PLC devices: HyperText Transfer Protocol (HTTP), EtherNet Industrial Protocol (ENIP), and STEP7. Eight fabricated protocol trees are designed to reflect various intermediate tree shapes between the three real protocol tree shapes. The experiment tests all eleven trees against the following Synchronization Algorithms: Catchup, Lockstep, Latelock, and Triggerlock. These algorithms are defined in Section 3.5.3. Additionally, the Triggerlock Algorithm is tested against these protocol trees with every possible TriggerDepth setting to examine differences between them.

1.5 Assumptions and Limitations

This research only seeks to compare four previously mentioned Synchronization Algorithms across three protocol trees and eight test trees created specifically for this experiment. Because ScriptGenE is protocol agnostic, it can emulate any network protocol able to be recorded using a network capturing tool like Wireshark or tcpdump. Many other protocols may create protocol trees appearing different than the three used. The three used are all PLC protocols and these protocols seem appropriate as this research project is Supervisory Control and Data Acquisition (SCADA) PLC oriented.

Assumptions are important in framing the experiment and narrowing in on the impact the varying factors have on the performance metrics. Assumptions from this experiment are listed below, along with why they are utilized.

- The communication times between the client, the honeypot, and the PLC are constant. They are calculated into the time it takes for the honeypot to produce a response and send it to the client. This assumption helps focus on the actual differences between the Synchronization Algorithms among the protocol trees and not the differences due to external network traffic.

- The size of the response packets are all the same, and the time it takes the honeypot to send these responses are the same as well. This assumption allows the research to more accurately measure the difference in Synchronization Algorithms by not worrying about a trial using only larger responses or a trial only using smaller responses. This constant size and response time is calculated by taking an average of the responses in the captured network pcaps.

1.6 Research Contributions

This research contributes to the field of ICS honeypots by adding functionality to the ScriptGenE Framework. ScriptGenE now analyzes the protocol tree and selects the best Synchronization Algorithm configuration based on the protocol. As a result of this research, the latency time between client request and honeypot response is decreased. Similarly, the load on the back-end PLC is decreased.

1.7 Thesis Overview

This thesis is divided into five chapters. This first chapter contains the basic introduction to the research problem and the experiment used to solve this problem.

Chapter II contains background information on the research problem and research topic. This chapter gives more insight into ICS and SCADA devices and why this research is important for their security. Background information on the ScriptGenE Framework and how it functions is also in Chapter II.

Chapter III describes the experimental methodology. This chapter reviews the experiment and all the tests used to prove the research hypothesis.

Chapter IV contains the results of the experiment. It contains a discussion of the results and the significance of those results.

Finally, Chapter V is the conclusion. This chapter summarizes the research and provides a broad overview including ideas for future research.

II. Background and Related Research

2.1 Overview

Chapter II provides background information relevant to the research problem and the ScriptGenE Framework in Section 2.2. The importance of Industrial Control Systems (ICS) and the vulnerabilities inherent with these systems are both discussed in Sections 2.2.1 and 2.2.2. This chapter also provides background knowledge on network operations and the protocols used to communicate between devices such as HTTP, Common Industrial Protocol (CIP), and ENIP in Section 2.2.3. The chapter continues with a discussion on how honeypots are a viable solution to the shortcomings of a traditional Intrusion Detection System (IDS) in Section 2.2.4. Finally, Chapter II concludes with a discussion of previous research related to manually-configured honeypots, protocol emulation, and hybrid honeypot systems in Section 2.3.

2.2 Background

2.2.1 Industrial Control Systems.

Nations rely heavily on critical infrastructures to be the backbone of both the economy and the nation's general well-being. In the United States, Presidential Policy Directive 21 defines sixteen sectors of critical infrastructure, a subset of sectors includes: chemical, commercial facilities, communications, dams, energy, information technology, nuclear, and water/wastewater systems [2]. ICSs are often used in these sectors. ICS is a general term used to describe an interaction where the data is received from sensors and then an action is taken based off the data received.

Figure 1 shows the flow of an ICS. There are three critical parts of the diagram: the Human-Machine Interface (HMI), the Remote Diagnostics and Maintenance, and the Control Loop. It is easier to understand this diagram by focusing on interactions

between the Control Loop and the HMI, especially when the diagram is applied to a water treatment facility. The sensors in the Control Loop send data (e.g., pH of the water) to the Controller, which then sends it to the HMI. An operator sees the data presented at the HMI. The operator can then take action (e.g., if the pH is too high or too low) and send a command back to the Controller to lower or raise the pH. If everything is normal, the operator will continue to monitor the HMI. If the Controller receives a command, it can then send the command to the actuators to execute. The Remote Diagnostics and Maintenance taps into the Control Loop to ensure the sensors, Controller, and actuators are all working correctly. Even though Figure 1 is of a general ICS, it also applies to the SCADA systems, which is the focus of this research.

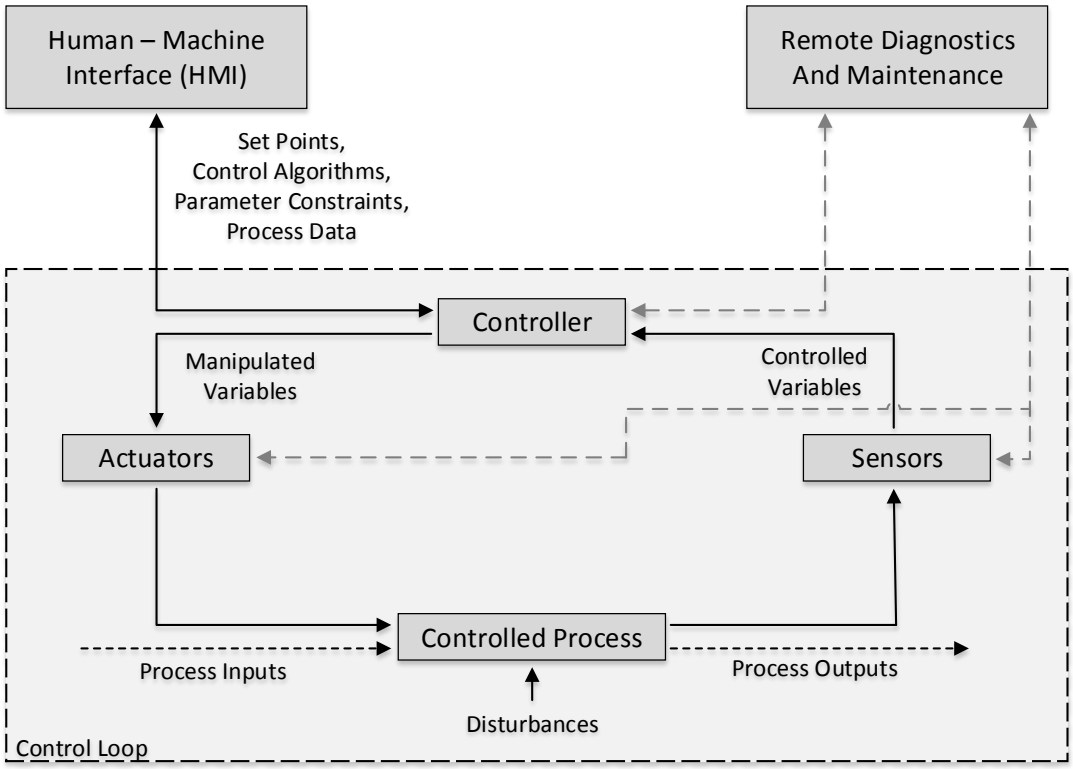

Figure 1. ICS block diagram [3]

Figure 2 shows a general layout of a SCADA system. It is broken down into three parts: the Control Center, the Wide Area Network (WAN), and the Field Sites. The Control Center contains the SCADA and Master Terminal Unit (MTU) receiving data from the Field Sites over a WAN. This system is typically used to control a Remote Terminal Unit (RTU), capable of being miles apart from other RTUs also controlled by the same system [4]. This system has an HMI presenting information to the operator monitoring the system. The MTU is a unit which can both receive data from RTUs and control RTUs in the field. The other two Field Site units in Figure 2 are the PLC and the Intelligent Electronic Device (IED). An IED allows for basic handling of the data (e.g., adjusting the pH to maintain an acceptable range) before sending it to the MTU. The PLC on the other hand allows for more data processing and control than the IED or the RTU because it allows for execution of entire programs.

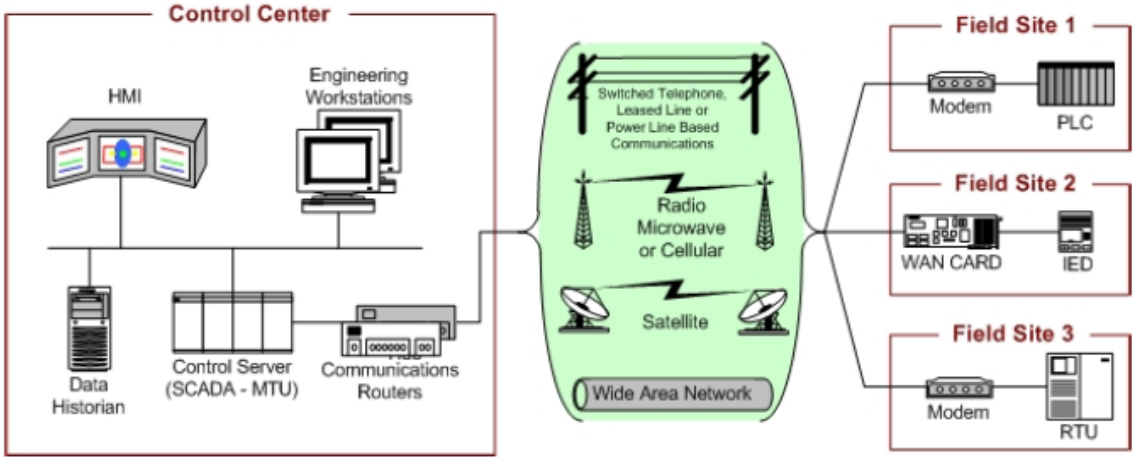

Figure 2. General SCADA layout [3]

PLCs are designed to be easily programmed, maintained, and able to communicate with a central control system [4]. They are often programmed to complete a specific task. There may be multiple PLCs at a location, each taking various measurements

and controlling different tasks. Because of this modularity, PLCs can be combined into many different configurations to allow for flexibility and functionality.

2.2.2 ICS Security.

Most ICSs were designed and implemented before the Internet became widely used. When some systems were finally connected to the Internet, the emphasis was on the integrity of the data sent and the availability between devices. This emphasis is where ICSs differ from traditional Information Technology (IT) systems. IT systems focus on ensuring the confidentiality of the data because information is typically most important to system operators. Whereas in an ICS, the availability of the system is most important. If the SCADA device is unavailable or only receiving partial data from the PLCs, then the system is not working properly which could lead to loss of profitability, loss of life, or system damage.

Since the emphasis was placed on data integrity and availability, confidentiality was often ignored. This oversight has placed ICSs in the spotlight as especially vulnerable to attack. Some noteworthy adversarial threats to ICS highlighted by the National Institute of Standards and Technology (NIST) in their guide to ICS security are: foreign intelligence services, industrial spies, criminal groups, and terrorists [3]. All of these threats have something to gain if they are successful in attacking an ICS controlling critical infrastructure.

Some examples of attacks on ICSs include the Stuxnet [5] virus and the Maroochy Shire Sewage Spill [6]. In both of these incidents an ICS was targeted and attacked. The Stuxnet virus was crafted to target only a very specific PLC, and while it infected many computers, it only did damage to its specific target [5]. The Maroochy Shire Sewage Spill incident happened in 2000 when a former employee of an Australian company that installed SCADA equipment for the Maroochy Shire Council applied

for a job and was rejected. The ex-employee then decided to hack into the SCADA equipment using radio equipment and release sewage into the river over the course of two months. He managed to attack the system in a way which pumps would run when they were not supposed to. The alarms on the HMI did not report anything to the central computer. Additionally, there were times when the MTU could not talk to the various pumping stations [6].

However, not all attacks on an ICS need to affect system safety. Some attacks aim to increase the cost of running the system. Such attacks focus on Denial of Service (DoS) or rendering a PLC unusable to the point requiring replacement. A study on the physical and economic consequences of attacks on control systems focuses on two types of attacks on a chemical reactor: integrity attacks and DoS attacks [7]. The integrity attack focuses on changing the sensor output in an attempt to get the MTU to send a command to bring the value back to its normal level, when in reality the reactor is already at an acceptable level. The DoS attack on the other hand ensures the sensor cannot communicate with the MTU. When these two attacks are used in combination, they can cause the equipment to catastrophically fail. Researchers highlight systems capable of being commanded to run inefficiently for long durations of time, causing the company to spend more money on resources than the cost of the system if it were destroyed [7]. One of the things making ICS attacks different than traditional cyber attacks is the estimated cost of damage on an ICS attack in an industrial plant. The cost can often be calculated by using the rate of production and the consumption of materials, versus a tradition cyber attack which remains difficult to predict damage cost.

In some systems, even conducting routine network ping scans can cause some unintended consequences for a SCADA network. In one instance, a ping scan was conducted on a network to ensure connectivity to all proper devices and one device

on the network controlled a 9-foot robotic arm. This device was pinged and the arm swung 180 degrees [8]. In another case, when a ping sweep was being conducted in a circuit fabrication plant, the sweep acted as a DoS attack on the device and cost the plant $50,000 worth of processing chip wafers [8]. A third case of unintentional consequences happened when a gas utility company hired a penetration test company to test their system. The penetration test company attacked the part of the network that had SCADA devices and caused the system to become unresponsive for 4 hours, resulting in a 4 hour gas outage [8].

Many companies owning ICSs are reluctant to add security to their systems because of increases in cost and they are concerned the additional complexity may break their systems [3]. These companies need to recognize there are financial reasons to add security to their systems. If a system is attacked and the damage to the SCADA network is significant, parts may need to be replaced, which can be very expensive. Decisions of whether or not a private company decides to secure their ICSs will ultimately be financially based.

The current guidance for secure ICS network configuration is to set up the SCADA system so it is completely separate from the corporate network [3]. This separation is to ensure if the corporate network is compromised, the SCADA network will not also be compromised along with it. The guidance suggests the network is setup in a similar fashion to network in Figure 3. In this figure, there is a Demilitarized Zone (DMZ) between the corporate network and the control network. This DMZ is in place so the corporate network can view information about the control network without directly being part of it. The control network can still update the Data Historian and the Data Server without ever being directly part of the corporate network. As orderly as this configuration sounds, it is still not an end all for ICS security.

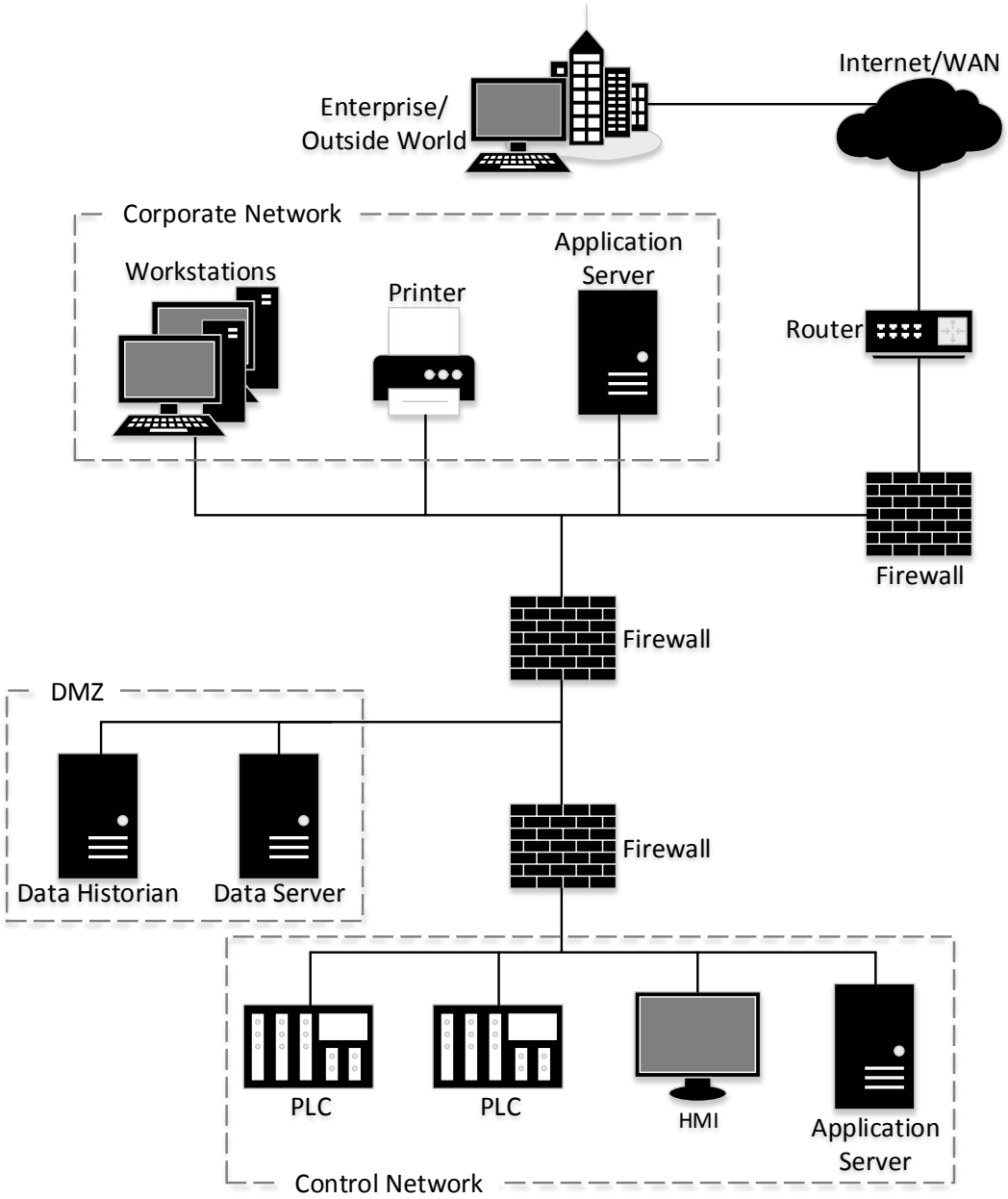

Figure 3. ICS network configuration recommended by NIST [3]

The main problem with SCADA security is the devices are inherently trusting. Many security techniques for ICSs are unconventional in an attempt to confuse the attacker as opposed to focusing on protecting the network through traditional means. One such unconventional technique is repackaging, where the user causes the system or security process to look like another system or process which is uninteresting to the attacker [9].

2.2.3 Networking and Application Layer Protocols.

To begin understanding honeypots and the necessary interactions between honeypots and client users, an understanding of the Networking and Application Layer protocols is first needed. Computer networks are commonly explained using the five layer model as seen in Figure 4. The Transport, Network, Link, and Physical layers are fairly rigid in what they are used for and what their content consists of. The most versatile layer is the Application Layer. While other layers focus on getting the packet to the proper destination, the Application Layer focuses on process content of the packet. This research focuses on the emulation of the Application Layer. The emulation of this layer is more difficult than focusing on other layers because of the variability and what can be sent in this layer. The remainder of this section is an overview of the application layer protocols used in this research.

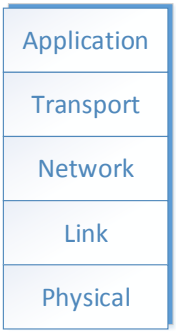

Figure 4. Internet Protocol stack [10]

2.2.3.1 HTTP.

The first protocol, HTTP is the most well known protocol of the three in this research. It is designed to be a stateless, generic protocol that is used to send HyperText Markup Language (HTML) over a network commonly received on port 80 [11]. This research uses a common HTTP message, the GET request. The GET request is the basic means to retrieve information from a server [11]. A normal response to this request is the HTTP message OK. The OK response is an acknowledgment the request has succeeded and the information to the request is enclosed [11]. Figure 5 shows an example of a GET message from the client (IP address 172.16.0.103) to the honeypot (IP address 172.16.0.104).

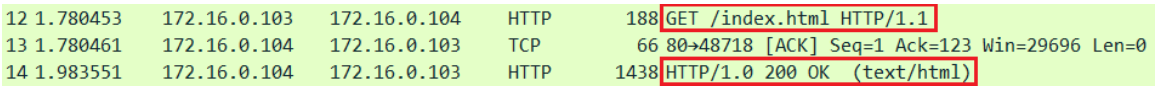

Figure 5. Packet Capture of HTTP GET and OK Messages

The first response from the honeypot in Figure 5 is an Acknowledgment (ACK) message indicating to the client the GET request has been received. The honeypot has not processed the client request yet. Once the honeypot processes the request, it replies to the client with the OK message. In this research, the time between the GET and OK messages is measured to calculate the response time between the honeypot and the back-end PLC and between the honeypot and the client. In the example from Figure 5, the response time is 0.203098 seconds.

2.2.3.2 STEP7.

The second protocol used is STEP7, which is a protocol designed and used by Siemens to encapsulate the proprietary Siemens control protocol called PROFINET [12]. This encapsulation allows the control information to be sent over the Internet in a recognizable fashion as opposed to having to patch the system so it can transfer the

proprietary protocol. STEP7 uses Transmission Control Protocol (TCP) port 102 to communicate [12].

2.2.3.3 ENIP.

The last protocol used in this research is ENIP. This protocol is used because most PLCs are stateful but use the stateless HTTP to send packets over the Internet. ENIP allows for stateful transactions using HTTP as a wrapper. ENIP has become the standard for ICSs. It allows for interoperability between devices from different manufacturers without custom software [13]. ENIP uses TCP port 44818, and in this research ENIP is used by the Allen-Bradley PLC.

2.2.4 Honeypots.

In the field of computer networks, a honeypot is a device set up and connected to an active network for the sole purpose of being attacked [14]. The other devices on the network are setup so they do not interact with the honeypot. This way, any interactions with the honeypot are viewed as anomalies and can be studied to discover the vulnerabilities used to compromise it [14]. A honeypot can be a physical device on the network or a virtual device setup to mimic a physical device. Multiple virtual devices can be setup to run on a single physical device. This virtualization makes them more reasonable for deployment since there is less cost for physical systems and maintenance [14].

Honeypots are often compared to traditional IDSs because of their ability to detect an intrusion. IDSs are systems designed to alert network administrators of unusual activity occurring in the network. However, these systems only detect abnormal traffic based on how to network administrators define normal network traffic. Because of this dictation of normal network activity, increasingly complex intrusions are camouflaging

themselves to look like the normal activity [15]. Another way for malicious actors to circumnavigate these IDSs is to encrypt the traffic so that the packets cannot be read by the IDS. IDSs are also known to have a high rate of false-positives of abnormal activity. This high rate lowers the effectiveness of the network administrator because they may become callus to alerts.

Honeypots can monitor all connections to the device and log keystrokes to allow network administrators to see the actions compromising the device. Because the monitoring occurs on the end device, the encrypted traffic previously unreadable by the IDS is now readable and understandable by the honeypot. Most IDSs use detection based on signatures and because of this style of detection, they can only detect previously known about vulnerabilities. A honeypot, on the other hand, can detect all activity on the device even if this is the first time the exploit is being used (e.g., Zero-Day attacks). This possible discovery of Zero-Days makes honeypots extremely valuable as a network security tool.

2.2.4.1 High-Interaction Honeypots.

A High-Interaction (HI) honeypot is almost a complete replica of the system it is emulating. In some cases it may actually be a second instance of the functional device. It can be costly to use a honeypot which is a physical replica of the original device; this is why virtual honeypots are preferred. In a HI honeypot, an attacker can often fully exploit the device. Because of this depth of interaction, additional defensive knowledge can be gathered from the exploit. However, sometimes with this greater ability to interact, the attacker can use the honeypot as an attack vector against the rest of the network [14]. In order to prevent this from happening, software like Honeywall has been implemented so the honeypots are isolated from the production network [16].

The allure of a HI honeypot is its ability to accurately replicate network traffic and network administrators know actual production devices are safe from these vectoring exploits because the honeypot is setup in a sandbox environment. Additionally, because of their level of authenticity, the HI honeypots can often attract complex attacks and sometimes even Zero-Day attacks. An example of a HI honeypot designed to catch Zero-Day attacks is Argos. The Argos honeypot tracks network data throughout execution and will check to see if the traffic causes any negative reactions. If something is detected, Argos then documents the memory footprint caused by the traffic and reports it to the network administrators [17].

2.2.4.2 Low-Interaction Honeypots.

Low-Interaction (LI) honeypots are only rough emulations of an actual device. In some cases these honeypots only replicate the open and closed ports of a device. LI honeypots are good for detecting autonomous attacks or activity like network probes or worms [14]. LI honeypots also provide additional benefits by requiring very little maintenance and being deployed in bulk at a fraction of the cost of a HI honeypot. These honeypots are designed to convince an attacker, usually automated botnets or worms, the device is real and should be attacked. However, because the device is so simplistic, the physical device running the honeypot itself is not usually attacked.

An example of a LI honeypot is the Dionaea framework. This honeypot framework is designed to attract malware tailored for certain services on a client's network and ultimately create a copy of the malware for further examination by the network operators [18]. Another example of a LI honeypot is Honeyd. Honeyd is a network of honeypots, called a honeynet. Honeyd is discussed further in Section 2.2.4.5. An extension of Honeyd called Honeyd+ is used to deploy the ScriptGenE Framework.

2.2.4.3 Active Honeypots.

The honeypots described in Sections 2.2.4.1 and 2.2.4.2 are passive: they log interactions, but they do not create any interactions. Active honeypot devices attempt to create interactions to appear as attractive to the attacker as possible [19]. In traditional honeypot systems the honeypot is static and the attacker may not find it an attractive target, they then move on to another system in the network, where once exploited may do actual damage to the network. Active honeypots can simulate multiple different devices and while one of the simulated devices may not attract the attacker, another might. An active honeypot is able to keep the attacker engaged with the honeypot, allowing additional insight into their attack techniques.

An example of an active honeypot is Beeswarm. Beeswarm deploys in a configuration of clients and honeypots. Beeswarm's clients interact with the honeypots and send information to the honeypot attractive to attackers like plaintext credentials and account activity [20]. These interactions between the fictious clients and the honeypots are shown in Figure 6. When the attacker intercepts these interactions, they appear as an actual set of devices interacting causing the attacker to attack the honeypot. The entire time, the Beeswarm clients and honeypots are reporting the actions of the attacker to the Beeswarm Server [20].

2.2.4.4 Hybrid Honeypots.

As previously discussed in Sections 2.2.4.1 and 2.2.4.2, a HI honeypot offers a high amount of authenticity while a LI honeypot offers scalability and flexibility. A hybrid honeypot tries to combine the best of both worlds by deploying a network of LI honeypots dealing with simple interactions. Then proxy interactions to a HI honeypot when the LI honeypots receive an interaction more complicated than they

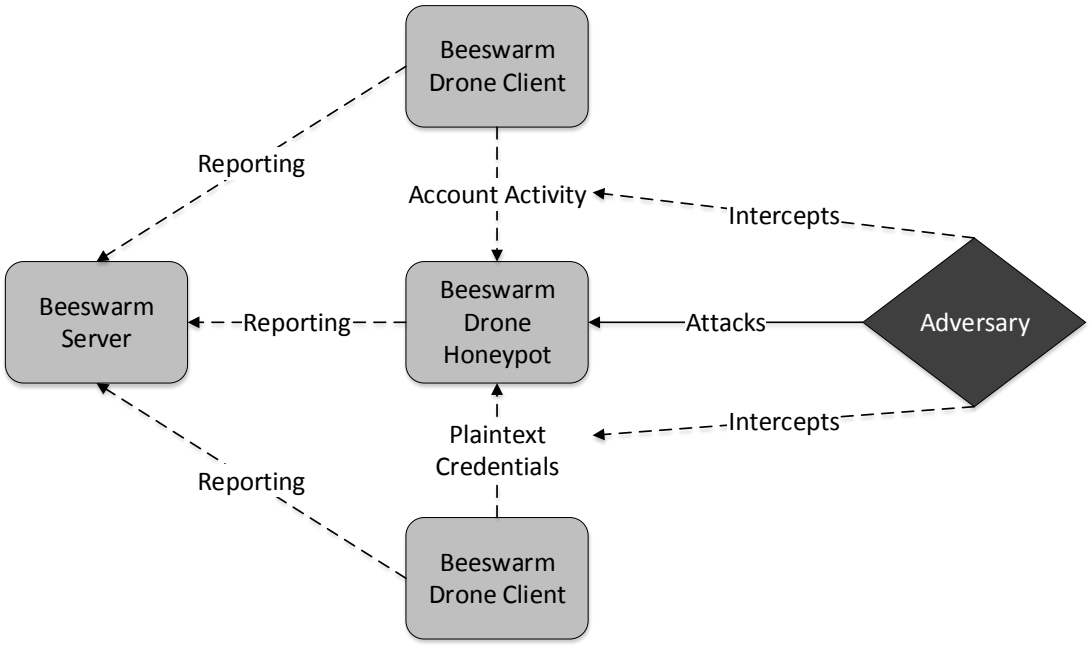

Figure 6. Beeswarm Overview

know how to respond. The rest of this section provides some examples of hybrid systems.

The first example is Potemkin. Potemkin is a system of Virtual Machines (VMs) activating as needed by external requests. Potemkin can be assigned a few hundred thousand virtual honeypots using only ten physical servers [21]. When the attacker tries to contact a virtual honeypot, Potemkin passes the interaction to one of its HI honeypots. This HI honeypot interacts with the attacker and monitors the connection. Through this process, Potemkin can look like many devices while only having a few HI honeypots running on the servers.

The second example is Collapsar, which is a centralized honeynet existing on multiple networks. It does so by having traffic redirected from the networks to the central honeynet handling the interactions and then returns the traffic to the original network [22]. Collapsar provides HI honeypots for all the connected networks while only having to maintain one centralized set of honeypots. One of the downsides

of Collapsar is the traffic forwarding may cause a large delay between the attacker interacting with a network and receiving a response, alerting the attacker they are interacting with a honeypot [22].

The third example of a hybrid system is Honeybrid. Honeybrid is a system that accepts network traffic and then based on the traffic, it decides to send it to a HI or LI honeypot [23]. The entire Honeybrid system is dependent on the decision engine that determines what type of traffic it is seeing. This configuration allows for the system to publically present many LI honeypots. Then the configuration uses a small number of HI honeypots when the traffic is deemed specialized and in need of more authenticity.

2.2.4.5 Honeyd.

Honeyd is an open-source framework capable of deploying many LI honeypots. Since they are LI honeypots, the devices are an empty shell designed to emulate an actual device in network scans. Honeyd is able to be configured to run arbitrary services and change the TCP fingerprint so the honeypots appear to be running certain applications on a predetermined Operating System (OS) [24]. The attacker can conduct network pings and traceroutes against Honeyd devices [14].

An additional useful function of Honeyd used previously building up to this research is the ability to proxy information from the Honeyd device to another machine. This is useful for giving each honeypot more depth in the types of responses it returns to the attacker. However, as with the Collapsar honeynet, too much proxying may alert the attacker to the inauthenticity of the device.

2.3 Related Research

2.3.1 Manually Configured ICS Honeypots.

Most of the previously discussed honeypots were built to be used on traditional IT systems, and they come with the ability to load interfaces emulating common applications or protocols found in a commercial network. However, it becomes much more challenging to have a honeypot preconfigured for the network it is deploying in because there are many different ICS protocols and different PLCs. Due to this difficulty, most ICS honeypot systems have to be manually configured to emulate the different devices and protocols. Once configured, there is very little flexibility, since the honeypots can only mimic the specific device they were set up to emulate.

Honeyd is a framework manually configured to simulate an ICS network. In an attempt to improve the authenticity of the Honeyd framework, Michael Winn created an extension called Honeyd+. Honeyd+ is designed to deploy on inexpensive hardware like a Raspberry Pi board and remotely connect to a back-end PLC helping the honeypot provide real responses to client requests [25]. Using this configuration, there can be multiple honeypots using Honeyd+, all using a single PLC being kept at a different location than the honeypots.

Honeyd+ also improves upon the framework of Honeyd by adding the increased functionality of changing the IP address and the Media Access Control (MAC) address of the proxied traffic. In Honeyd, the traffic forwarded to the physical PLC is returned to the client with the PLC IP and MAC address instead of the honeypot's IP and MAC address. The new functionality in Honeyd+ added by Winn, ensures all communications with the client have the same IP and MAC addresses [25].

2.3.2 Automatic Protocol Emulation.

In an attempt to overcome the amount of work needed to manually configure a honeypot or honeynet, some honeypot research sets out to automatically emulate the protocol. These systems are built to observe the network traffic and then use the port numbers and traffic format to determine which protocol should be emulated. The remainder of this section explores some of the previous research attempts at protocol emulation.

The first example is research done by Chowdhary et al. The research aimed to emulate protocols using a process they refer to as "service-mining", where they use large amounts of network traffic to attempt to recognize patterns and then the software emulates that traffic [26]. This research emulated File Transfer Protocol (FTP) and is considered to be a success.

The second example is research called BAIT-TRAP. This research realized many honeypots are static and manually configured for one service. In order to fix this issue, BAIT-TRAP monitors network traffic and identifies the most utilized services on a network. It then adapts itself to emulate those services in order to be more relevant in the network [27]. This system pulls from a database of preset protocols to emulate and is not able to emulate new protocols without a new entry to the database.

Another example of protocol emulation was research done by Rafique et al. This research aimed to examine network traffic and compare it against previous network traffic in an attempt to determine the similarities and differences to create a better emulation of the network traffic [28].

2.3.3 Hybrid Honeypots with Replay.

In order to create a better honeypot, some programs and research attempt to create a protocol-agnostic device. These devices are designed to completely reverse

engineer protocols, then determine which fields need to be change and which must stay constant. Such a device is useful because it is available at the cost of a LI honeypot since it is deployable on a VM. But it can also provide the levels of interaction comparable to a HI honeypot. Sections 2.3.3.1 and 2.3.3.2 explore some of these projects.

2.3.3.1 Protocol-Agnostic Replay.

An example of a program using previously monitored network traffic to replay under the appropriate context is Role Player [29]. It is a system capable of emulating both the client and server side of a session's application protocol without any specific knowledge of the application. It uses various algorithms to determine which parts of the conversation are transport, network, link, and physical layer overhead [29]. RolePlayer is capable of detecting and adjusting network addresses, port, cookies, and length fields in the network traffic [29].

The next example of protocol emulation is ScriptGen. ScriptGen is designed to work with the Honeyd to create a new script using state machines to understand the structure of the network protocol [30]. As with RolePlayer, ScriptGen does not need to know anything specific about the protocol to be able to emulate it. After the new script is created, it can be deployed on a Honeyd framework [30]. ScriptGen also has the ability to proxy traffic to a device familiar with the protocol being used. After ScriptGen proxies this traffic, it can dynamically change itself to reflect this new knowledge of the protocol [31].

ScriptGen is taken a step further by Phillip Warner to better handle unknown transactions and default responses [32]. This new iteration is named ScriptGenE and is utilized in this research. This new design is made specifically to be utilized in ICSs and with many different, and often proprietary protocols. ScriptGenE processes

the observed network traffic to create protocol trees to follow a conversation between a client and device. As with ScriptGen, these trees can be converted into Honeyd scripts to replay conversations. A further discussion of these protocol trees can be found in Section 2.3.4.

2.3.3.2 Hybrid Honeypots.

One of the first hybrid systems to utilize protocol-agnostic replay is GQ [33]. GQ uses RolePlayer as the interaction point for all incoming traffic. Once the traffic is processed by RolePlayer, it is either dealt with by RolePlayer or if RolePlayer does not know how to respond to it, it is sent to a HI honeypot capable of authentically responding to it [34].

Another example of a hybrid honeypot framework is SGNET. SGNET is designed to work similarly to GQ except instead of the honeypots needing to be deployed on a centralized computer, they can be decentralized onto many smaller platforms [35]. This style of deployment allows for the honeypot network to be deployed on consumer computers (e.g., the Dell Latitude E6520 Laptop used in this experiment) and does not require specialized systems. SGNET uses ScriptGen to emulate network traffic without needing to know protocol specifics. SGNET is the first deployment of the ScriptGen project in an Internet environment.

2.3.4 Protocol Trees.

Protocol trees are created by the ScriptGenE Framework to aid the honeypot in emulating a protocol. ScriptGenE divides network traffic pcap files into client requests and corresponding server responses. ScriptGenE then creates a tree similar to the one in Figure 7. ScriptGenE utilizes the generated protocol tree to follow a series of client requests to return the correct server response.

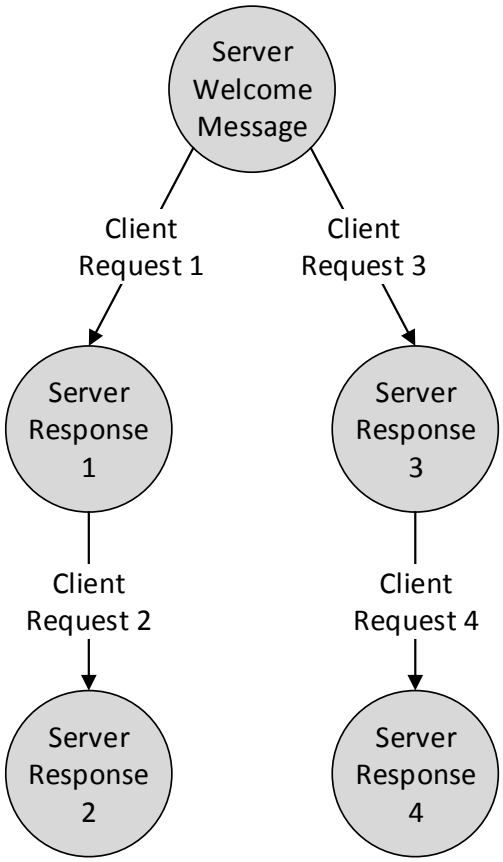

Figure 7. Protocol Tree Design

An example HTTP protocol tree can be seen in Figure 8. This example displays the information the protocol tree stores for emulation. In this example, the honeypot starts in the welcome state accepting new connections. After the client has connected, the honeypot receives the client request of *GET /index.html HTTP/1.1*. The honeypot matches this request to the protocol tree and follows the matching edge to the next node containing the *HTTP/1.0 200 OK* server response. After this transaction, the honeypot stays in this node/state until the client disconnects or sends another message. If the client sends another request, the honeypot backtracks up the tree to look for nodes nodes corresponding to the client request. Even though HTTP is a stateless protocol, the tree ScriptGenE creates to represent it is not.

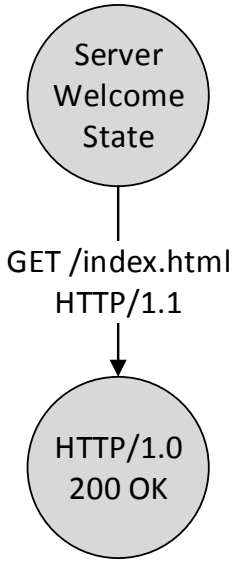

Figure 8. HTTP Protocol Tree Example

2.3.5 Synchronization Algorithms.

Synchronization Algorithms are utilized in the ScriptGenE Framework to ensure the back-end PLC is in the same state as the honeypot. This can be thought of ensuring both the back-end PLC and the honeypot are at the same point in the

protocol tree, even though the PLC does not have a protocol tree. The honeypot only needs to ensure synchronization when it receives a client request not found in the protocol tree, also referred to in this research as Not-In-Tree requests. To respond to the request appropriately, the honeypot must proxy the request to back-end PLC after synchronization.

Figure 9 displays a transaction between a client and the honeypot where the first four client requests are in the protocol tree, while the next five client requests are not. The black bars signify the honeypot has established a connection with the back-end PLC. There are four Synchronization Algorithms tested in this research:

1. Catchup. This algorithm waits until a Not-In-Tree client request is received by the honeypot before it synchronizes the back-end PLC. Because of this design, the honeypot must send all previous client requests to the PLC. Unlike any of the other algorithms, the honeypot disconnects from the back-end PLC after every response. The example scenario in Figure 9 displays the design of Catchup and the disconnection after every Not-In-Tree Request.

2. Lockstep. This algorithm is always synchronized with the physical device. As soon as the honeypot receives a client request, it proxies the request to the back-end PLC. If the client request is In-Tree, the honeypot responds to the client before receiving the response from the back-end PLC. Figure 9 displays the constant connection of Lockstep.

3. Latelock. This algorithm works the same as the Catchup Algorithm; it waits until it receives a Not-In-Tree client request before proxying to the back-end PLC. The difference is it does not disconnect from the back-end PLC after receiving the response to the proxied client request, but instead it stays in Lockstep with the PLC. In Figure 9, after the first Not-In-Tree request, the Latelock Algorithm stays in connection with the back-end PLC.

4. Triggerlock. This algorithm waits to synchronize with the back-end PLC until the client request is a certain depth (TriggerDepth) from the root node in the protocol tree. At Triggerdepth, the Triggerlock Synchronization Algorithm begins to synchronize with the back-end PLC. The expected values for TriggerDepth are integers from zero to the maximum depth of the protocol tree. This design ensures the honeypot will never need to proxy more than the predetermined depth upon receiving a Not-In-Tree request. An example of this algorithm in action is displayed in Figure 9. With a TriggerDepth setting of 3, the Triggerlock Algorithm waits until after it receives the third message from the client before establishing a connection with the back-end PLC.

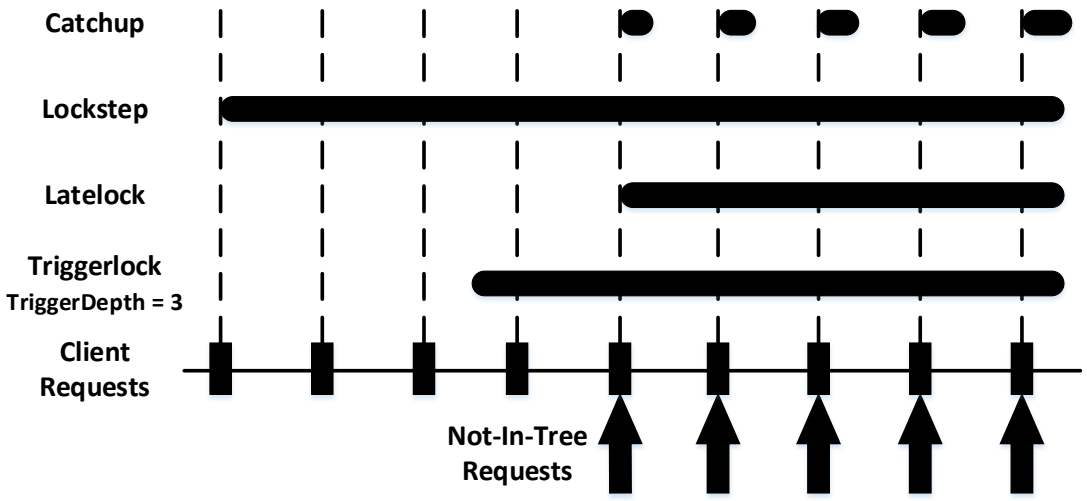

Figure 9. Synchronization Algorithm Example Adapted From [1]

2.4 Chapter Summary

Chapter II examines ICSs, the importance of security for these systems, and the various ways honeypots help fill in the security gaps increasing over time. ICSs often control the nation's critical infrastructure and because of this importance they need

to be protected from inevitable external threats arising from connecting devices to the Internet. Only in the last decade has awareness of the threats to ICSs increased, yet most of these systems have been in use before security was considered important for ICSs. Unlike traditional IT devices which have an expected life span of 5-6 years, an ICS device is expected to last for 25 years before needing to be replaced. This long life means devices with built-in security may not replace existing devices for many years to come, so measures must be taken to secure the devices externally.

This need for external security is where honeypots can excel. They offer a great degree of authenticity, flexibility, affordability, and scalability. While neither a LI or HI honeypot can offer all of these traits, a combination of the two can. These hybrid honeypots can be taken a step further by adding protocol-agnostic replay software which can remove the need for honeypots to be manually configured or even need to know which protocols will be used on the network. ScriptGenE is one of these honeypot frameworks and is the honeypot this research seeks to improve.

III. Research Methodology

3.1 Goals

ScriptGenE is designed to take as input a network communication protocol in the form of a network packet capture and create a protocol tree to emulate the protocol. The ultimate goal of this research is to then have ScriptGenE evaluate the protocol tree to determine the best Synchronization Algorithm for each protocol tree. The best algorithm reduces latency, while simultaneously reduces the load on the back-end PLC. Specifically, this research seeks to answer the following questions:

1. Given a protocol tree, is there a best Synchronization Algorithm yielding minimized latency and minimized load on the back-end PLC?

2. Does the shape of the protocol tree impact which Synchronization Algorithm should be used?

3. Is there an optimal Triggerdepth setting for the Triggerlock Algorithm based on the tree shape or size?

Chapter III discusses the methodology of the experiments run in order to answer these questions.

3.2 Approach

The ScriptGenE framework used in this research is designed to emulate the observed network traffic of a specified device. The framework creates protocol trees using the observed network traffic and uses these protocol trees to respond to client requests. This research furthers the ScriptGenE Framework by evaluating the protocol tree to select a Synchronization Algorithm best suited for the protocol. Testing

this extension to the ScriptGenE Framework involves not only creating new protocol tree replications of commonly used PLC protocols like ENIP, STEP7, and HTTP; but also creating new protocol trees representing various possible protocols. The three real world PLC protocols are selected because of the different structures of protocol trees they produce when run through ScriptGenE. The other protocol trees are produced to cover a range of trees between the three protocols for more variability in the experiment.

Each experimental run accepts all eleven protocol trees and executes four tests on all trees. These four tests can be thought as two distinct groups (tests 1 and 2, and tests 3 and 4). Tests 1 and 2 seek to answer the first two research questions discussed in Section 3.1, while tests 3 and 4 seek to answer the third research question also discussed in Section 3.1.

Tests 1 and 3 determine the latency and back-end PLC load for the case when the honeypot has the client request in the protocol tree (In-Tree). Tests 2 and 4 determine the latency and back-end PLC load for the case when the honeypot does not have the client request in the protocol tree (Not-In-Tree) and has to forward the request to the back-end PLC in order to properly respond to the client. All experimental runs are randomized by randomly selecting the client request from each of the protocol trees. The four tests are run utilizing every Synchronization Algorithm for every tree to ensure all possible configurations are tested.

3.3 System Boundaries

The main System Under Test (SUT) is the ScriptGenE Framework. The Component Under Test (CUT) is the ScriptGenE.py file, responsible for creating and analyzing the protocol tree. Figure 10 displays the SUT and CUT along with the parameters and metrics. This experiment works by running the eleven generated protocol trees

through 5000 different randomized scenarios using each of the four Synchronization Algorithms. The number 5000 is selected to ensure the confidence intervals of the means is small enough the mean are comparable. Each group of two tests, outlined in Section 3.2, produces the evaluation metrics of response time and load on the back-end PLC. Since there are two groups each with two performance metrics, there are four total performance metrics for this experiment.

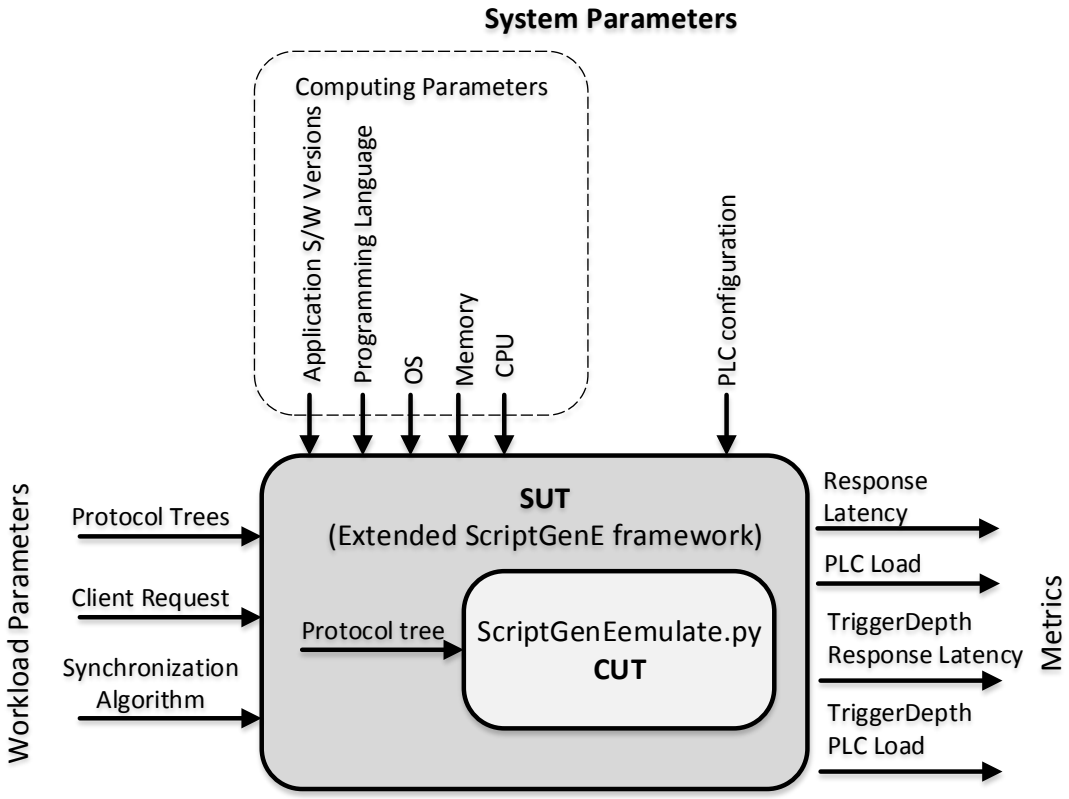

Figure 10. System Under Test

3.4 System Parameters

The System Parameters are inputs capable of affecting the performance metrics and because of this possible impact, these parameters are kept at one constant value.

These constants limit the scope of the experiment and focus in on the actual impact the factors are having on the performance metrics. Sections 3.4.1 and 3.4.2 contain the experimental variables held constant. They are presented along with the reason why they are kept constant.

3.4.1 Network Traffic Collection System.

The equipment utilized to collect the network traffic, which is then used to create the protocol trees and the average times to respond to requests, must be kept constant so accurate comparisons can be made between protocol trees and Synchronization Algorithms. Figure 11 displays network traffic collection setup. Below are the specifications of the laptop and the two PLCs used to collect the network traffic:

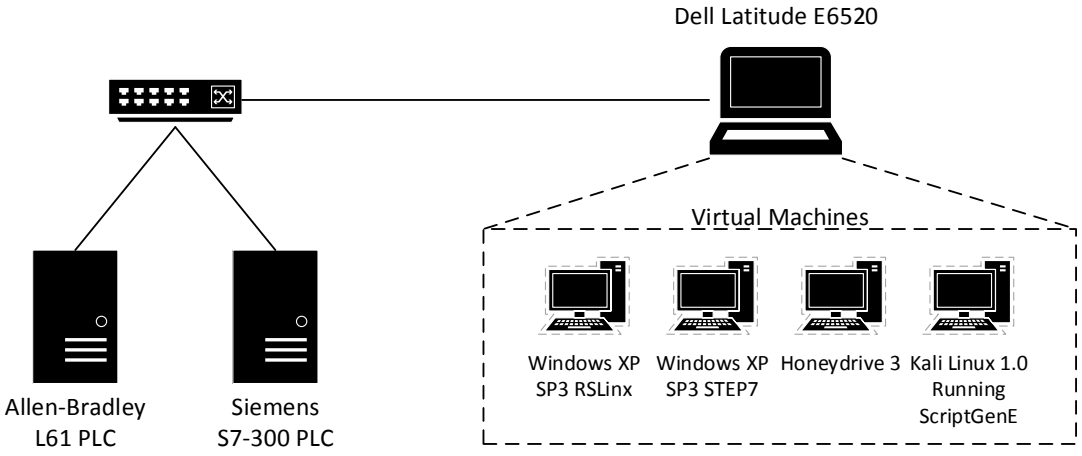

Figure 11. Network Traffic Collection Setup

Dell Latitude E6520 Laptop

- Microsoft Windows 7 Service Pack 1
- 2.2GHz Intel Core i7-2720QM processor
- 8GB RAM

- VMware Workstation version 12.0.0 build-2985596

Allen-Bradley ControlLogix5561 (L61) PLC

- Firmware version 19.015
- Slot 0 - L61 Controller with mode set to `REM Run` (remote `Run`)
- Slot 1 - 1756-EWEB EtherNet/IP ENBT

Siemens SIMATIC S7-300 PLC

- Firmware version 2.6
- Slot 2 - CPU 315-2 Controller with one Ethernet port
- Slot 4 - Discrete I/O (DI16xDC24V)
- Slot 5 - Discrete I/O (DO16xDC24V/0.5A)
- Slot 6 - Discrete I/O (DO16xAC120V/230V/1A)
- Slot 7 - Discrete I/O (DO16xRel. AC120V/230V)
- Slot 8 - Analog I/O (AI8xTC)
- Slot 9 - Analog I/O (AI8x16Bit)

3.4.2 Protocol Trees.

This experiment tests eleven protocol trees and are the same trees throughout the entire experiment. This sameness is to ensure the findings for one algorithm's impact on a certain tree can be compared to another algorithm's impact on the same tree.

3.5 Factors

Just as it is important for certain variables to be held constant in the experiment, it is important certain variables are purposefully changed to measure their impact

on the system. The following four factors are intentionally varied so their impact on the system can be measured using the performance metrics discussed in Section 3.6. Each factor is presented with measurement units, the expected range of values, and the reason for being included.

3.5.1 Protocol/Tree Structure.

This factor is measured as a categorical variable. It keeps track of the protocol being emulated by the honeypot. The expected range of values for this factor are the eleven different protocol trees in this experiment: the three PLC protocol trees and the eight other hybrid trees designed to be a blend of multiple protocols. The three PLC protocols used in this research are: ENIP, HTTP, and STEP7. Figure 12 displays these three protocol trees. Tree 1 has a depth of one because it is a stateless protocol and previous requests do not impact the responses. On the other hand, Trees 2 and 3 are more vertical than Tree 1 because they are created from stateful protocols.

The other eight protocol trees are shown in A and are created for this research to simulate various other protocols ScriptGenE may encounter. They are designed to look different from the three PLC protocols to determine how changing the shape of and size of the trees impacts the performance metrics. Trees 4 and 5 are designed to be vertical in shape but with more width than the ENIP and STEP7 trees. Tree 6 is twice as wide as Tree 4 but still taller than it is wide. Trees 7 is designed to be equally tall as it is wide. Tree 8 and Tree 9 are wider than they are tall, but still taller than the stateless protocol HTTP. Tree 10 is a pyramid with a depth of five, where every node has two children. Tree 10 is created to determine the impact of a non-rectangularly shaped tree. Tree 11 is also non-rectangular, but an inverted

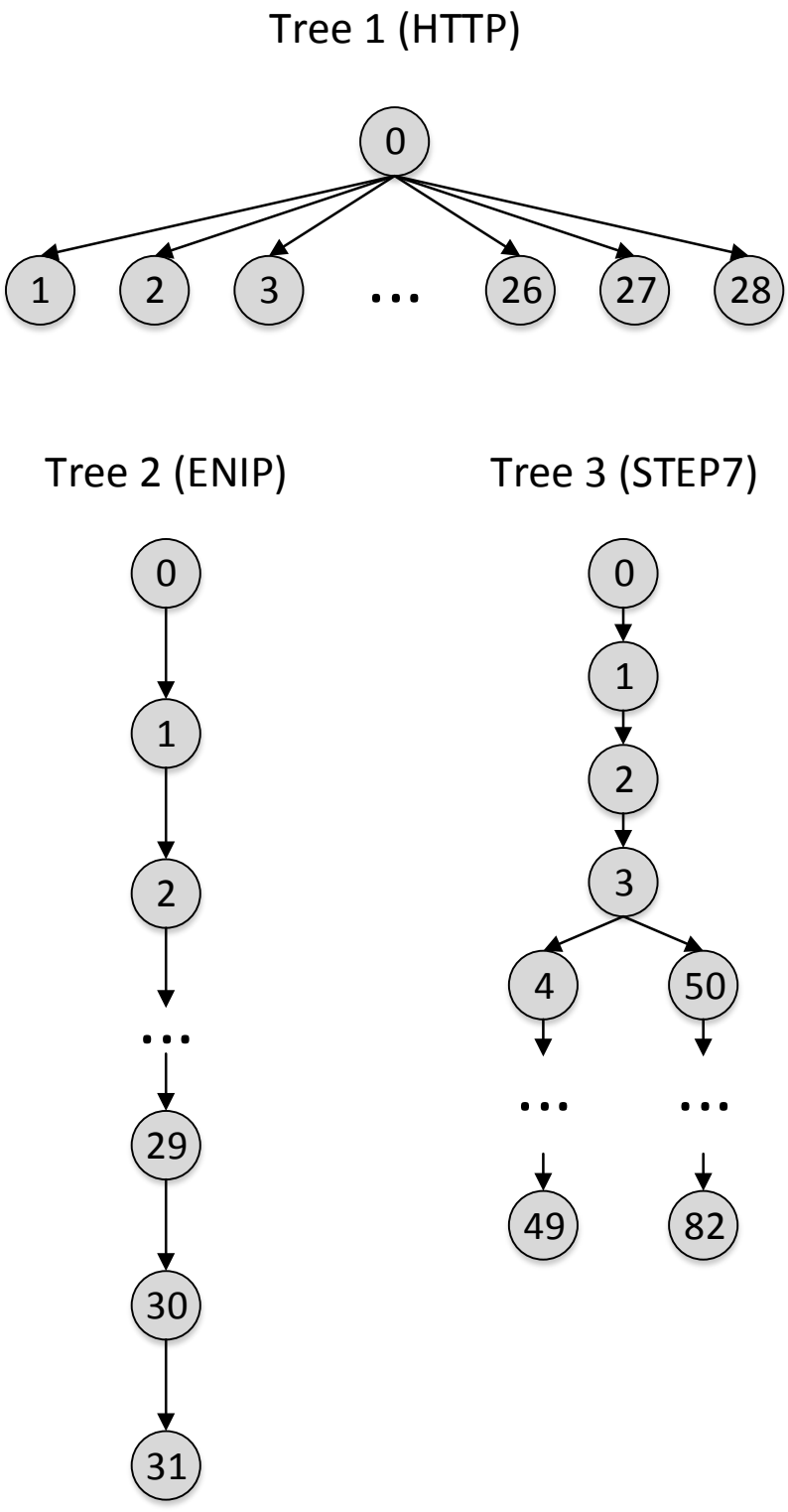

Figure 12. Protocol Trees 1, 2, and 3

pyramid of Tree 10. For a better understanding of these two protocol trees, Table 1 lists the number of nodes at every tree depth for Tree 10 and Tree 11.

Table 1. Tree 10 and 11: Nodes per Depth

	Tree 10	Tree 11
Tree Depth	Number of Nodes	
0	1	1
1	2	32
2	4	16
3	8	8
4	16	4
5	32	2

This factor is important for the experiment because the honeypot is supposed to be protocol agnostic in that it can emulate any protocol without previous knowledge of the protocol. It is supposed to monitor traffic into and out of the production device and emulate the traffic without knowing anything about the actual production device. The ScriptGenE Framework must be able to handle various different protocols and their resulting protocol trees regardless of how different they look from previously used protocol trees. The goal of this research is for the honeypot to evaluate the protocol tree and select a Synchronization Algorithm based on the protocol tree; therefore, varying the trees allows for a more thorough test of the Synchronization Algorithms.

3.5.2 Client Request.

This factor is measured as a categorical variable. It keeps track of whether the client request is in the protocol tree or whether the client request is not in the protocol tree and needs to be proxied to the back-end PLC. This factor is referred to throughout Chapters III and IV as In-Tree or Not-In-Tree to delineate data for when the protocol tree contains the client request or does not contain the request respectively.

This factor is important because the Synchronization Algorithms behave differently based on whether the request is In-Tree or Not-In-Tree.

3.5.3 Synchronization Algorithm.

This factor is measured as a categorical variable. It measures which Synchronization Algorithm is being used by the honeypot to ensure the back-end PLC is in the same state as the honeypot before the client request is proxied. The expected values for this variable are the various Synchronization Algorithms being used. The four Synchronization Algorithms are Catchup, Lockstep, Latelock, and Triggerlock. These four algorithms are defined in Section 2.3.5.

3.6 Performance Metrics

Four performance metrics are used in this experiment to properly test the impact the four different Synchronization Algorithms have on the honeypot. The four metrics revolve around the purpose of the honeypot, which is to accurately and quickly respond to client requests while minimizing the load on the back-end PLC.

This section presents a variable with the measurement units, the expected range of values, and the reason for being included as a performance metric.

3.6.1 Response Latency.

The first metric is total response latency. This metric is measured in seconds and measures the total time it takes for the honeypot to send a response to the user after the honeypot receives the request. The expected values for this metric range from 0.1 to 0.5 seconds. The total response latency is important for this experiment because a PLC used in industry is a real-time system and is designed to have a low response time. For the honeypot to be more convincing in emulating the PLC, the response

latency must be as low as possible. The typical PLC response time is less than 0.1 seconds, which was found by examining pcap files of PLC network traffic.

3.6.2 PLC Load.

The second metric is total load on the back-end PLC. This metric is measured in back-end PLC request from the honeypot per client request received by the honeypot. This variable measures the efficiency of the Synchronization Algorithm. This calculation is done by taking the number of requests the honeypot has to proxy to the back-end PLC and divides this number by total number of client requests. The range of values expected for this metric are 0-10 proxies per client request. The expected value can be larger than one because the Catchup Algorithm may require multiple interactions with the back-end PLC to synchronize. This metric should not go above one for any Synchronization Algorithm except Catchup because all other algorithms only synchronize once. Catchup synchronizes every time it receives a Not-In-Tree client request. This variable is important to the experiment because there may be honeypot configurations with many honeypots connected to one production device. In this configuration, the load on the back-end PLC device should be as small as possible. Thus, allowing the back-end PLC to be available when a honeypot needs to proxy a request.

3.6.3 TriggerDepth Response Latency.

The total response latency for every TriggerDepth is the same as the first metric of total response latency except this is measuring the results of the second group of experiments on the Triggerlock Algorithm. Similar to previous measurements, this metric is measured in seconds. Expected values for this metric is a range of values from 0.1 to 0.5 seconds.

3.6.4 TriggerDepth PLC Load.

This metric is the total load on back-end PLC for every TriggerDepth. This metric is also the same as the metric from the first group of experiments except this measures the load for the second groups of experiments. This metric is measured in the number of client requests proxied to the back-end PLC. The expected values are a range from 0 to the depth of the protocol tree in test.

3.7 Experimental Assumptions and Limitations

As with any experiment or simulation, assumptions must be made since this experiment is executing in a simulated environment. The following is a list of assumptions for this experiment:

3.7.1 Assumptions.

3.7.1.1 Honeypot to PLC Communication Time.

The time it takes for the honeypot to communicate with the back-end PLC device is constant. This time is not the total time the honeypot takes to request a response from the PLC device, just the time it takes for the packets to travel between the honeypot and the physical device.

3.7.1.2 Honeypot to Client Communication Time.

The time it takes for the honeypot to communicate with the client is constant. This time only represents the time it takes for the packets to travel between honeypot and client.

3.7.1.3 Packet Size.

The size of all responses to client requests are the same and the time for honeypot and the PLC to receive and respond to these requests are consistent. This sameness allows the experiment to narrow in on the difference between the four Synchronization Algorithms by eliminating the variance in response sizes and times.

3.7.1.4 Correct Responses.

The response the client receives from the honeypot, whether forwarded or directly answered from the protocol tree, is correct. This sameness allows the experiment to focus on the performance metrics of latency time and back-end PLC load as opposed to whether the ScriptGenE Framework is functioning as expected.

3.7.2 Limitations.

To accurately compare the impact the factors have on the performance metrics, the test measurements are run in a closed network to limit error in latency measurements. This closed network is a limitation because the system is designed to work on an open, public-facing network, but it is necessary to limit the error due to other traffic on the network when taking precise time measurements.

Another limitation of this experiment is response times are based on the averages of the three real-world protocols being tested. These response time averages are a limitation because ScriptGenE may encounter other protocols, which may produce different results. However, these three protocols are included because they are commonly used in ICS and ICS is the emphasis this research.

3.8 Analysis Techniques

All of the data obtained through experimentation is processed and analyzed using the GNU package R scripting language [36]. The R scripts written for this research, as seen in Appendix B, take raw test output data as input and create more readable statistics like the mean, confidence intervals, and graphs.

3.9 Experimental Setup

3.9.1 Experimental Scripts.

The following Scripts are used to setup and conduct the experiment:

- ScriptGenE.py
  - This program accepts the network packet capture and the IP address of the production device to be emulated. ScriptGenE imports the Python class ptree from the script ProtocolTree.py. ScriptGenE uses this class to create the protocol tree. Then it uses this tree to become the honeypot. This program is used to create the pcap files needed to calculate the response time averages.
- TreeBuilder.py
  - This program extends the Python class ptree from the script ProtocolTree.py. TreeBuilder.py both replicates the three production protocol trees and creates the eight hybrid protocol trees designed to cover a range of tree shapes between the three production protocols.
- TreeAnalyser.py

- This script is the main test program. It takes the protocol trees as inputs and outputs Comma-Separated Values (CSV) files. Each CSV file contains the latency times and back-end PLC loads for each of the 5000 test runs on a protocol tree using the four Synchronization Algorithms and various TriggerDepths. After exporting these files, the program performs a system call to run the PostExperiment.R file.

- PostExperiment.R

  - This R-file takes the CSV files as input and outputs the graphs and tables used in this paper. PostExperiment.R can be found in Appendix B.

3.9.2 Configuring and Running The Experiment.

As mentioned in Section 3.2, there are two groups of tests. Each test runs twice, once with the assumption the protocol tree contains the client's request and does not need to proxy the request to the back-end PLC (In-Tree). The second run assumes the protocol tree does not contain the client's request and therefore needs to proxy the request to the back-end PLC (Not-In-Tree).

The first step in the experiment is to construct the eleven protocol trees of varying sizes. The first three trees are formed by running the observed network traffic from previous research [1] through ScriptGenE.py using the HTTP, ENIP, and STEP7 protocols. The protocol trees used in the experiment are then created for the three production protocols using the TreeBuilder.py file. These new protocol trees are identical in size and shape to the original protocol tree, except the new tree only contains the identifiers of each node in the tree and does not contain protocol specific data. The removal of protocol specific data allows for the different protocol trees to be better compared against each other since their only difference now is tree structure and not content. An example of the difference between a ScripGenE.py generated

tree and a TreeBuilder.py tree is shown in Figure 13. After the three production protocol trees are created, the TreeBuilder.py file creates the eight other protocol trees. Figure 14 displays the process of creating the eleven protocol trees.

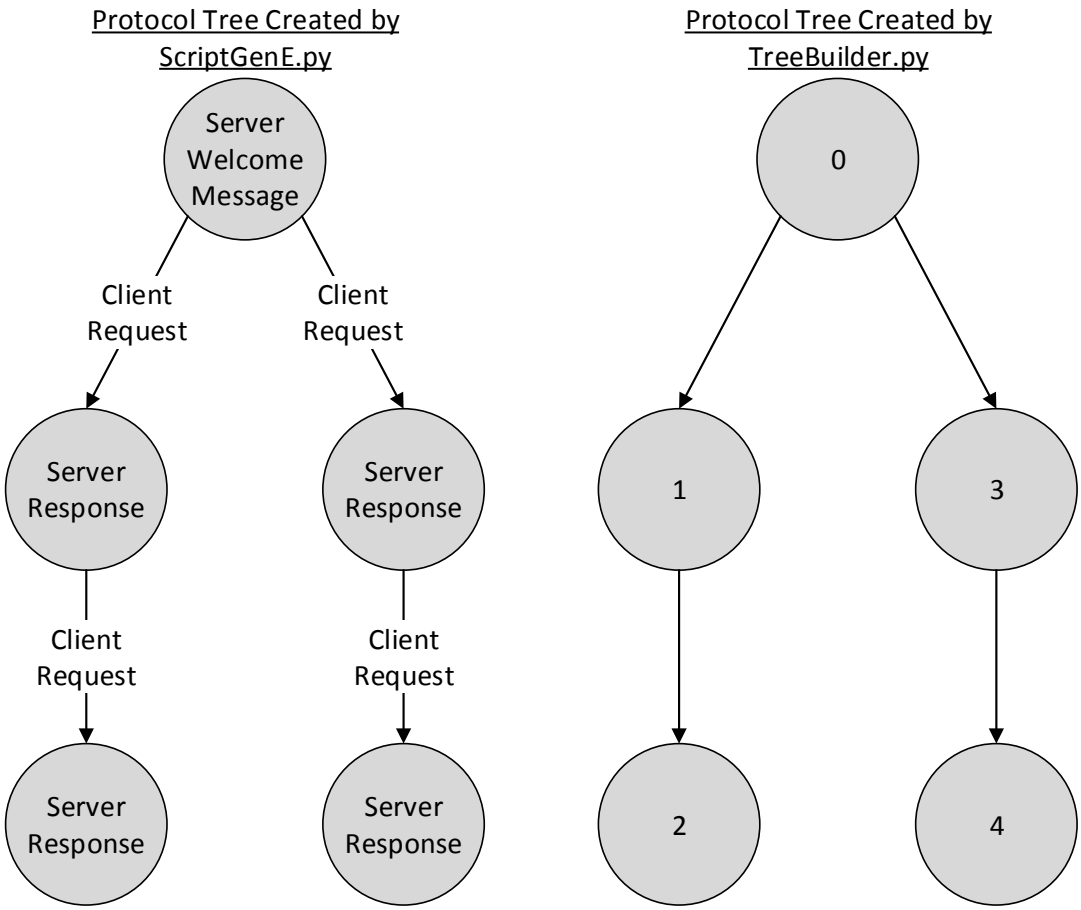

Figure 13. Protocol Trees Generated by ScriptGenE.py and TreeBuilder.py

The next step of the experiment is to get a protocol tree through one run of the experiment to determine the response latency and back-end PLC load for each of the first three Synchronization Algorithms: Catchup, Lockstep, and Latelock. After those values are calculated, the experiment then calculates the response latency and PLC load for each of the TriggerDepths for the Triggerlock Algorithm.

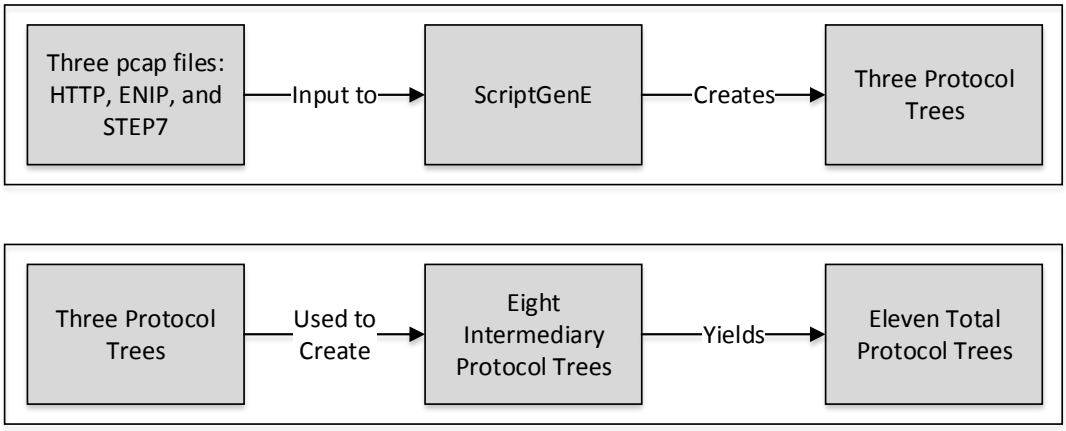

Figure 14. Creation of the Eleven Protocol Trees

To calculate these values, the test selects a random node (using a random number generator in Python) from the protocol tree and uses this node as the simulated client request. Figure 15 is a diagram of the experiment. In this figure, the red nodes signify the randomly selected node. The response latency and back-end PLC load is then calculated with the assumption the node was actually in the protocol tree. Next, the response latency and back-end PLC load must be calculated with the assumption the node was not in the protocol tree. As mentioned, this step involves selecting another random node (again using a random number generator in Python) between the client's requested node and the root node. This randomly selected node will always be between the client's requested node and the root node. In Figure 15, the blue node signifies this randomly selected node. This newly selected node is then used as the last node the protocol tree contained. In this scenario, the honeypot must proxy requests from the last known node up to the client's requested node.

The single run of the test outlined in the previous paragraph is executed for each protocol tree to finish a single iteration of the experiment. There are 5000 iterations completed during the experiment. The number 5000 is selected because of initial results collected during the pilot study. Initially, the experiment only consisted of

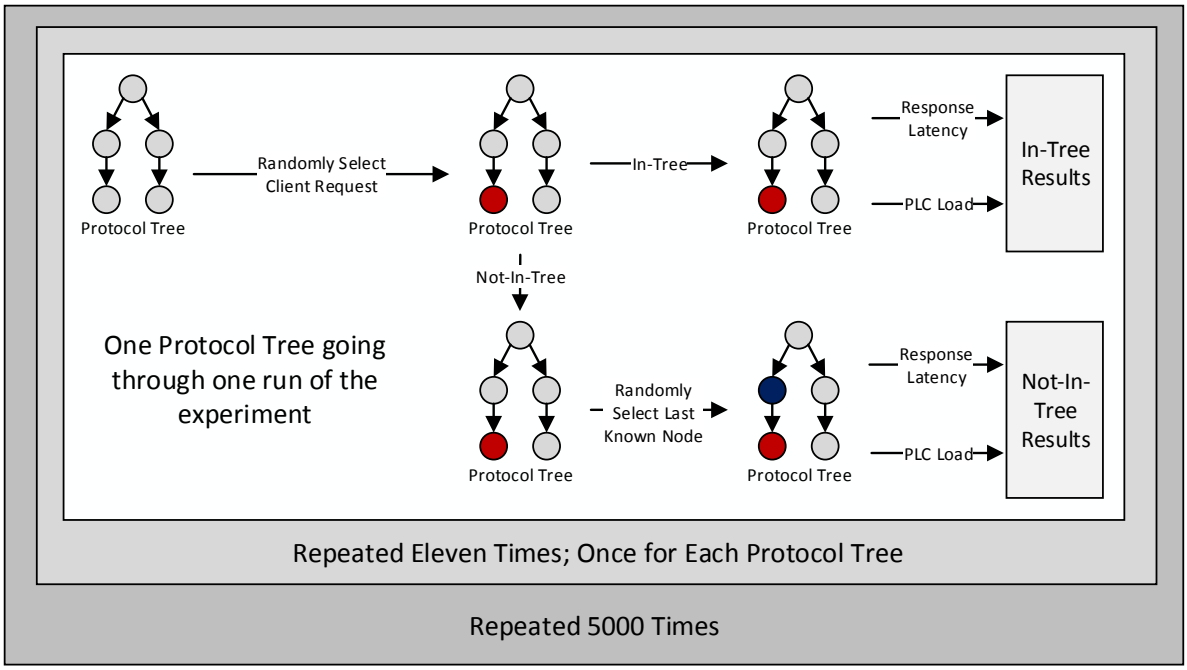

Figure 15. Diagram of the Experiment

500 iterations, but the results produced were not statistically distinguishable from each other because the confidence intervals overlapped. This problem was remedied by increasing the iterations by a factor of ten.

The total experimental data consists of 5000 iterations, each with random nodes being selected as the client's requested nodes and honeypot's last known nodes. So four Synchronization Algorithms producing results for latency and load across both the In-Tree and Not-In-Tree scenarios creates 16 data values for one protocol tree. After all eleven protocol trees are processed, there are 176 data points. The total experiment after running 5000 times produces a total of 880,000 data points for analysis.

When the values are calculated and the program has run through 5000 randomized tree configurations for each protocol tree, the data is exported in CSV files. These

CSV files are used as input arguments to the PostExperiment.R script to create tables and graphs used to better understand the raw data.

3.10 Methodology Summary

This chapter discusses the experimental methodology used to test the efficiency of the four Synchronization Algorithms. It outlines the goals of the research and how the experiment is going to achieve those goals. The two groups of tests used to answer the three research questions are described. The performance metrics of response latency and back-end PLC load are used in this experiment to show how one algorithm compares to another when dealing with the eleven different protocol trees. The next chapter describes the results of this experiment and how these findings impact the ScriptGenE Framework going forward.

IV. Results and Analysis

4.1 Overview

Chapter IV presents and analyzes the results from the experiment described in Chapter III. This chapter presents each of the Performance Metrics from Section 3.6 and performs both an observational and interpretive form of analysis. Observational analysis looks at the data and describes the trends while commenting on the performance in reference to the research hypothesis and the research questions. Interpretive analysis attempts to explain why the data looks the way it does and what it means in respect to the research hypothesis and research questions.

4.2 Metric 1 - Impact of Algorithm on Latency

Low latency is critical to honeypot effectiveness because if the honeypot takes too long to respond to the client's request, the client recognizes it is communicating with a honeypot instead of the real production device. This section explores the results from the experiment in regards to how each algorithm influences the latency between client request and honeypot response.

4.2.1 In-Tree Results.

This section describes results when the request is found in the tree. Since the tree contains all the responses and does not need to proxy client requests to the back-end PLC, all the response times are the same between the algorithms. These similarities can be seen in Figure 16. The response time for every algorithm across every protocol tree is 0.169 seconds. The confidence intervals for this figure are not included because they are all zero and they do not visually differentiate themselves from the plot points on the graph.

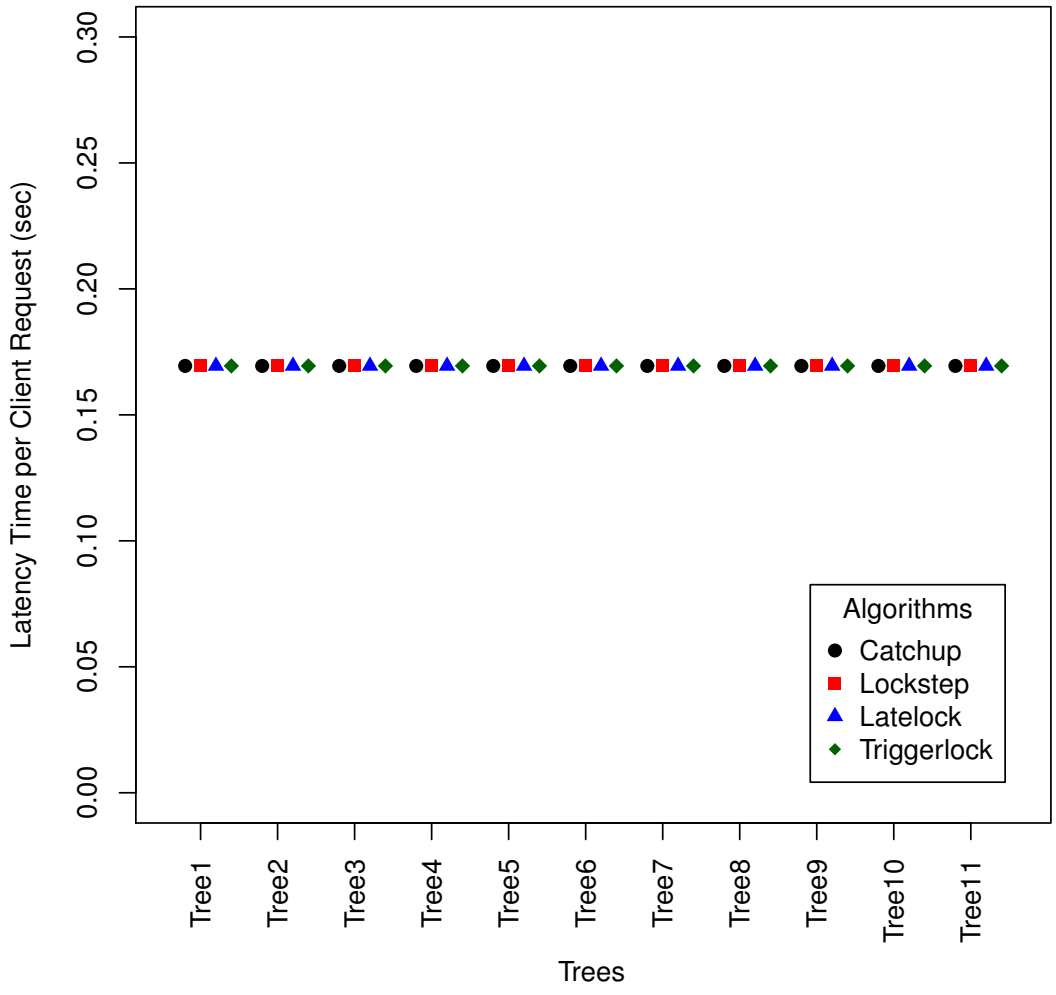

Figure 16. Metric 1 - Latency Times: In-Tree Scenario

Table 2 contains the latency times for every algorithm tested against all eleven trees. The columns Triggerlock Low and Triggerlock High in this table and Tables 3, 5, and 7 display the lowest and highest latency times or PLC loads for the Triggerlock Algorithm among the possible TriggerDepths.

Table 2. Metric 1 - Latency Times: In-Tree Scenario

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree2	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree3	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree4	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree5	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree6	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree7	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree8	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree9	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree10	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209
Tree11	0.1694209	0.1694209	0.1694209	0.1694209	0.1694209

4.2.2 Not-In-Tree Results.

Since the response to the client's request is not in the tree, the honeypot is required to proxy the communication to the back-end PLC. As shown in Figure 17, this proxying can cause the response time of the honeypot to vary based on the implemented Synchronization Algorithm. The Triggerlock Algorithm in the graphs for this chapter are represented by two green diamonds connected by a line. These two diamonds represent the highest and lowest values found when changing the TriggerDepth. For example, the highest diamond in Figure 17 comes from the TriggerDepth being set to the maximum depth of the tree, while the lowest diamond represents the value when the TriggerDepth value is set to one. The line connecting the two diamonds represents all the other values found by setting the TriggerDepth to values between 1 and the maximum tree depth.

For every tree, except Tree 1, the Catchup Algorithm is slower than the other algorithms due to the disconnection of the honeypot to the back-end PLC after every response. Similarly, for every tree except Tree 1, the Lockstep Algorithm is faster than the Latelock Algorithm in responding to client requests. The reason Tree 1 is the same for every algorithm in this scenario is because Tree 1 is the HTTP protocol tree. HTTP is a stateless protocol, which means the response to the current request does not depend on the previous client requests. Because previous requests do not matter, the protocol tree for Tree 1 is a root starting node with an edge for every known client request connecting the root to the appropriate response nodes. Since Tree 1 has a depth of one and the response to the client request is not in the tree because of the testing scenario, the time it takes every algorithm to respond is the same.

As with Figure 16, the confidence intervals for Figure 17 are not included because they are not visually different from the plot points on the graph. Table 3 shows every algorithm's response time. The standard error for each algorithm for every tree is included in Table 4. The upper and lower confidence intervals can be found by adding and subtracting respectively the standard error to the average latency time.

Due to the nature of the Triggerlock Algorithm, it can act similarly to the Lockstep Algorithm or the Latelock Algorithm based on its configuration of the TriggerDepth. The Triggerlock Low column in the Table 3 is identical to the Lockstep column. This indistinguishability comes from setting the TriggerDepth to one. When the first client request is received, the honeypot begins synchronizing with the back-end PLC. This is the same way the Lockstep Algorithm works. So with a TriggerDepth of one, the Triggerlock Algorithm produces results identical to the Lockstep Algorithm for the latency times in the Not-In-Tree Scenario.

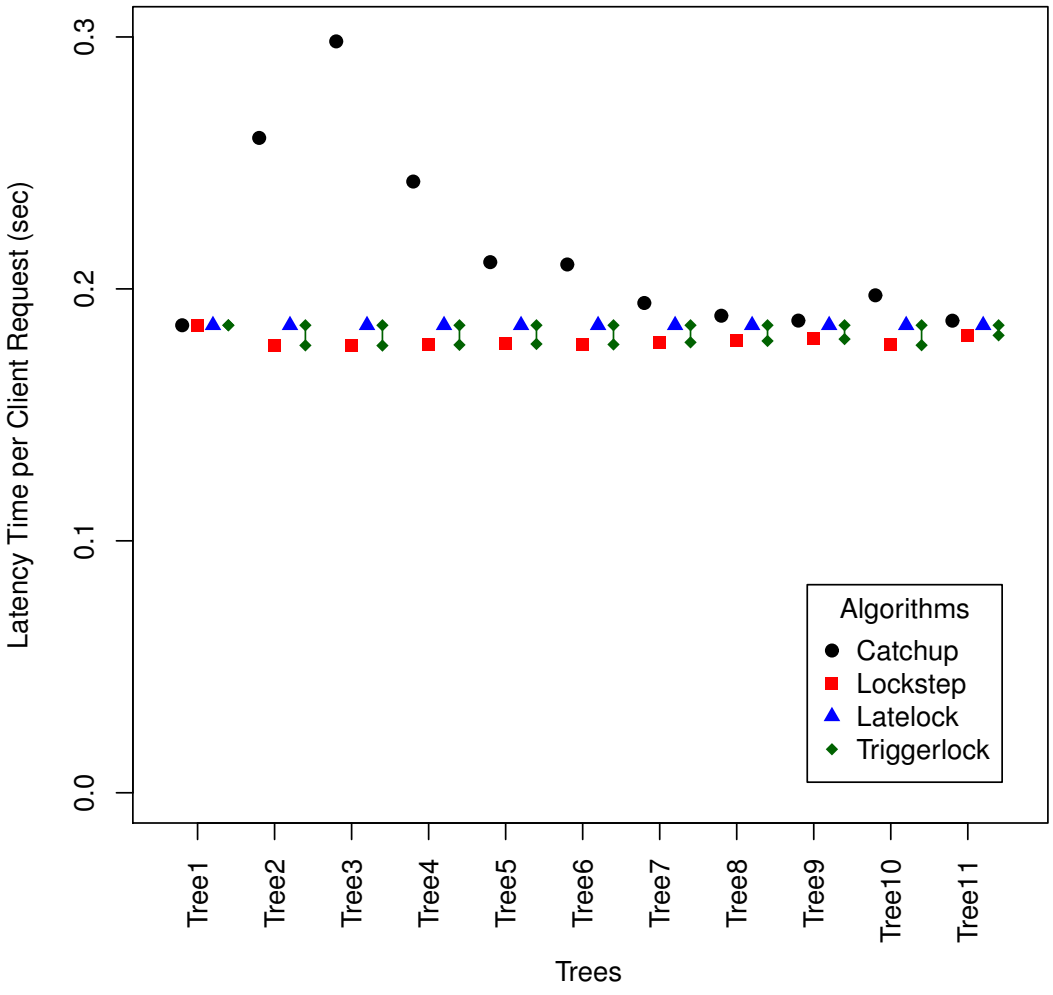

Figure 17. Metric 1 - Latency Times: Not-In-Tree Scenario

A similar comparison can be made to the Triggerlock High column and the Latelock column. This time, the similarity comes from setting the TriggerDepth to the maximum depth of the protocol tree. With this TriggerDepth, the Triggerlock Algorithm waits until the last possible client request to synchronize, thus the algorithm is never synchronized when it receives the unknown client request. Because the Triggerlock Algorithm is never synchronized under this TriggerDepth, it works the same way as the Latelock Algorithm. This is why the latency times are the same between Triggerlock High and Latelock in this Not-In-Tree Scenario.

Table 3. Metric 1 - Latency Times: Not-In-Tree Scenario

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	0.1855669	0.1855669	0.1855669	0.1855669	0.1855669
Tree2	0.2599264	0.1776914	0.1855669	0.1776914	0.1855669
Tree3	0.2982410	0.1776178	0.1855669	0.1776178	0.1855669
Tree4	0.2426074	0.1778855	0.1855669	0.1778855	0.1855669
Tree5	0.2106105	0.1782425	0.1855669	0.1782425	0.1855669
Tree6	0.2096852	0.1780515	0.1855669	0.1780515	0.1855669
Tree7	0.1944022	0.1788921	0.1855669	0.1788921	0.1855669
Tree8	0.1893496	0.1794502	0.1855669	0.1794502	0.1855669
Tree9	0.1874140	0.1801859	0.1855669	0.1801859	0.1855669
Tree10	0.1974518	0.1777428	0.1855669	0.1777428	0.1855669
Tree11	0.1873834	0.1816839	0.1855669	0.1816839	0.1855669

Table 4. Metric 1 - Latency Times: Not-In-Tree Scenario Standard Error

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
Tree2	0.0017887	0.0001210	0.0000000	0.0001210	0.0000000
Tree3	0.0025646	0.0001226	0.0000000	0.0001226	0.0000000
Tree4	0.0013864	0.0001188	0.0000000	0.0001188	0.0000000
Tree5	0.0006887	0.0001149	0.0000000	0.0001149	0.0000000
Tree6	0.0006769	0.0001153	0.0000000	0.0001153	0.0000000
Tree7	0.0003160	0.0001147	0.0000000	0.0001147	0.0000000
Tree8	0.0001809	0.0001144	0.0000000	0.0001144	0.0000000
Tree9	0.0001125	0.0001139	0.0000000	0.0001139	0.0000000
Tree10	0.0002917	0.0000984	0.0000000	0.0000984	0.0000000
Tree11	0.0001438	0.0001181	0.0000000	0.0001181	0.0000000

4.3 Metric 2 - Impact of Algorithm on PLC Load

As mentioned in Section 3.6.2, the load placed on the back-end PLC is important for this experiment because the back-end PLC needs to be able to handle as many simultaneous requests as possible. It must support the number of honeypots designed to be using only one PLC.

4.3.1 In-Tree Results.

The algorithms requiring constant communication with the back-end PLC like Lockstep and Triggerlock (once the TriggerDepth is reached) impose a larger load than Catchup and Latelock. This difference is because Catchup and Latelock do not require PLC communication until the protocol tree does not have a response to the client's request. As seen in Figure 18, Catchup and Latelock remain at zero load per client request because they do not require the back-end PLC for the In-Tree scenario, since they have the answer in the protocol tree. However, the Lockstep and Triggerlock Algorithms have loads corresponding to how they are designed to function. For the Lockstep Algorithm, the load is consistently one request to the PLC per client request. Because of this design, Lockstep has the ability to respond the fastest to an unknown client request, but places the largest load on the back-end PLC. The Triggerlock Algorithm is displayed in Figure 18 as a range of values from the lowest to the highest possible load, based on the variable TriggerDepth settings. As with the previous figures, the confidence intervals for this figure are not included because they are small enough they are not visually different from the plot points on the graph.

Comparisons can be drawn between the Lockstep and Triggerlock Algorithm with a certain TriggerDepth setting. For the In-Tree Scenario measuring the PLC load, Triggerlock places an identical load on the PLC as Lockstep when the TriggerDepth

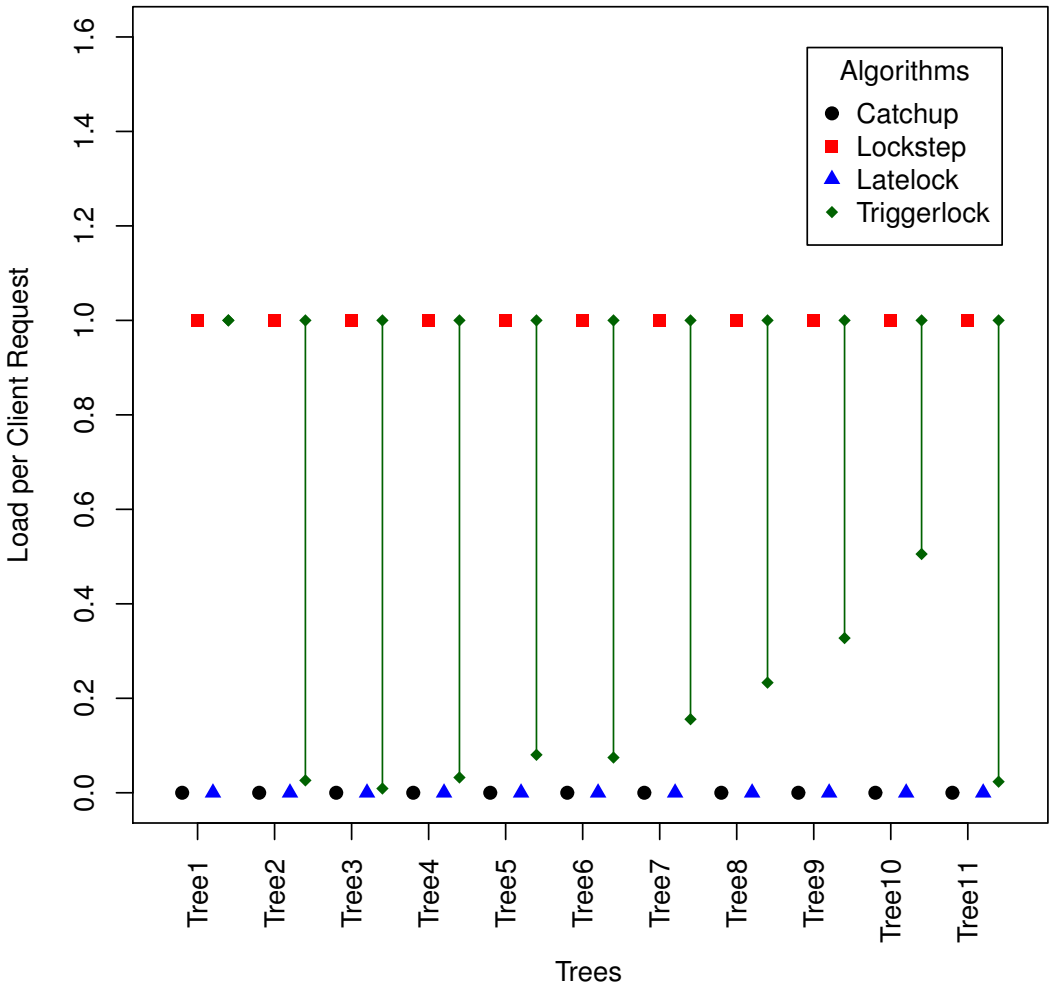

Figure 18. Metric 2 - PLC Loads: In-Tree Scenario

is one. As explained in Section 4.2.2, the Triggerlock Algorithm with a TriggerDepth setting of one proxies all incoming client requests to the back-end PLC just like the Lockstep Algorithm. Table 5 shows how Triggerlock High (e.g., Triggerlock with a TriggerDepth of one) produces the same results as Lockstep. This table also gives the specific values for the Triggerlock Algorithm, with the TriggerDepth setting of the maximum depth per protocol tree. The reason the Triggerlock Low Algorithm does not yield the same results as the Latelock Algorithm is because even with a TriggerDepth of the maximum protocol tree depth, the Triggerlock Algorithm has to proxy all previous client requests if the client reaches the maximum depth. The Triggerlock Algorithm with a TriggerDepth of n begins to synchronize with the back-end PLC once the honeypot receives the nth request, even if n is equal to the maximum depth of the protocol tree. Since the Triggerlock Algorithm has reached the maximum depth of its current protocol tree, it does not mean the client will not send another request. If the client does send another request, the honeypot should be synchronized with the PLC. This is why the Triggerlock Low column in Table 5 is not zero like the Latelock column. The standard error for each algorithm for every tree is included in Table 6.

Table 5. Metric 2 - PLC Loads: In-Tree Scenario

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	0.0000	1.0000	0.0000	1.0000	1.0000
Tree2	0.0000	1.0000	0.0000	0.0310	1.0000
Tree3	0.0000	1.0000	0.0000	0.0120	1.0000
Tree4	0.0000	1.0000	0.0000	0.0376	1.0000
Tree5	0.0000	1.0000	0.0000	0.0882	1.0000
Tree6	0.0000	1.0000	0.0000	0.0822	1.0000
Tree7	0.0000	1.0000	0.0000	0.1660	1.0000
Tree8	0.0000	1.0000	0.0000	0.2450	1.0000
Tree9	0.0000	1.0000	0.0000	0.3406	1.0000
Tree10	0.0000	1.0000	0.0000	0.5192	1.0000
Tree11	0.0000	1.0000	0.0000	0.0280	1.0000

Table 6. Metric 2 - PLC Load: In-Tree Scenario Standard Error

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	0.0000000	0.0000000	0.0000000	0.0000000	0.0000000
Tree2	0.0000000	0.0000000	0.0000000	0.0048046	0.0000000
Tree3	0.0000000	0.0000000	0.0000000	0.0030184	0.0000000
Tree4	0.0000000	0.0000000	0.0000000	0.0052733	0.0000000
Tree5	0.0000000	0.0000000	0.0000000	0.0078613	0.0000000
Tree6	0.0000000	0.0000000	0.0000000	0.0076142	0.0000000
Tree7	0.0000000	0.0000000	0.0000000	0.0103145	0.0000000
Tree8	0.0000000	0.0000000	0.0000000	0.0119226	0.0000000
Tree9	0.0000000	0.0000000	0.0000000	0.0131374	0.0000000
Tree10	0.0000000	0.0000000	0.0000000	0.0138504	0.0000000
Tree11	0.0000000	0.0000000	0.0000000	0.0045732	0.0000000

4.3.2 Not-In-Tree Results.

As seen in Figure 19, Latelock, Lockstep, Triggerlock Low, and Triggerlock High have the same back-end PLC load. This similarity is because they all have to proxy every client request, since this scenario guarantees an unknown request. However, the Catchup Algorithm has a significantly higher back-end PLC load for every protocol tree, except Tree 1, because of its design to break the connection after every request it proxies. This design causes exponential increases in the load on the back-end PLC. This design is why Tree 1 is the same load as the other algorithms. Figure 19 has the confidence intervals displayed for the Catchup Algorithm because unlike the previous graphs the standard error is large enough for the deeper protocol trees like Trees 2, 3, and 4, the intervals are visibly different from the average load plot points. This variance is due to the large amount of load placed on the back-end PLC the farther away the client request is from the root node.

Table 7 displays the exact values for the back-end PLC load per client request. As previously stated, the Catchup Algorithm is the only algorithm that produces a load other than one.

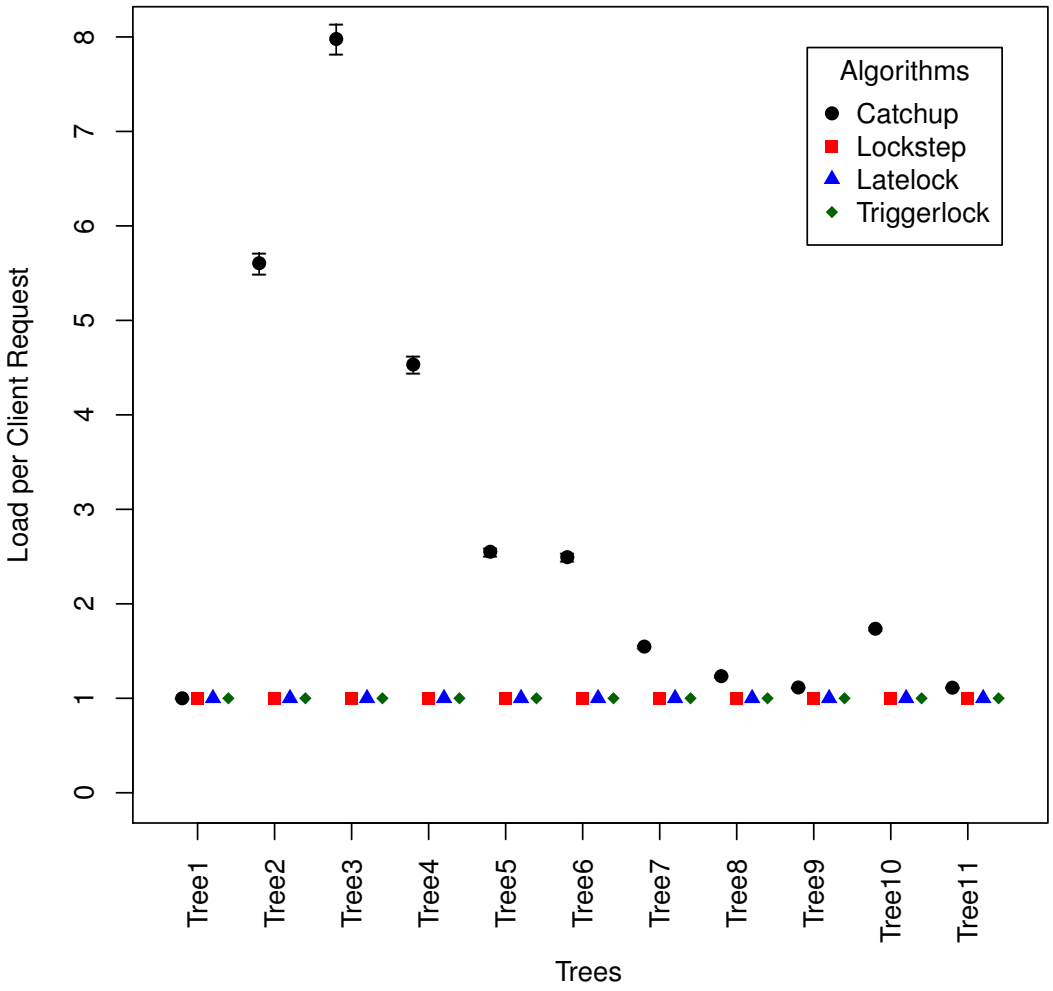

Figure 19. Metric 2 - PLC Loads: Not-In-Tree Scenario

Table 7. Metric 2 - PLC Loads: Not-In-Tree Scenario

Tree	Catchup	Lockstep	Latelock	Triggerlock Low	Triggerlock High
Tree1	1.0000000	1.0000000	1.0000000	1.0000000	1.0000000
Tree2	5.6054474	1.0000000	1.0000000	1.0000000	1.0000000
Tree3	7.9784536	1.0000000	1.0000000	1.0000000	1.0000000
Tree4	4.5327983	1.0000000	1.0000000	1.0000000	1.0000000
Tree5	2.5510753	1.0000000	1.0000000	1.0000000	1.0000000
Tree6	2.4937642	1.0000000	1.0000000	1.0000000	1.0000000
Tree7	1.5472167	1.0000000	1.0000000	1.0000000	1.0000000
Tree8	1.2342833	1.0000000	1.0000000	1.0000000	1.0000000
Tree9	1.1144000	1.0000000	1.0000000	1.0000000	1.0000000
Tree10	1.7360933	1.0000000	1.0000000	1.0000000	1.0000000
Tree11	1.1125066	1.0000000	1.0000000	1.0000000	1.0000000

4.4 Metric 3 - Impact of TriggerDepth on Latency

For the next two sections, the discussion shifts from all the Synchronization Algorithms to just discussing the Triggerlock Algorithm and how varying the TriggerDepth setting impacts the latency and back-end PLC load. For this discussion on latency, the In-Tree Scenario results are not discussed because every TriggerDepth yields the same value of 0.169421 seconds. The depth the honeypot synchronizes with the back-end PLC does not impact the response latency when the honeypot's protocol tree already contains the response to the client's request. Instead, this section focuses on the impact the varying TriggerDepth settings have on the response latency when the protocol tree does not contain the response to the client's request.

4.4.1 Not-In-Tree Response Times.

Across all tests, as the TriggerDepth increases so does the response time. These results are in line with the expected results based on the design of the Triggerlock Algorithm. When the exchange between the client and the honeypot progresses past the TriggerDepth, the back-end PLC is now synchronized and can instantly respond if the honeypot needs to proxy. When the exchange between the client and the honeypot

has not yet reached the TriggerDepth and the honeypot receives a request that it does not have in its tree, it must synchronize all previous messages to the back-end PLC before sending the unknown request. In the tests having a larger TriggerDepth, there is an increase in the penalty when the honeypot does not know how to respond to a client's request. This penalty comes from all previous requests having to be proxied to the PLC. When only considering response latency in the TriggerDepth setting for the Triggerlock Algorithm, the lower the TriggerDepth, the faster the response time. Figure 20 shows the Not-In-Tree response times for Tree 2 as the TriggerDepth increases. Table 8 gives the specific averages for every TriggerDepth setting along with the standard error for each average.

From the slowest to fastest average response times in Figure 20, there is a speedup of 0.004186 seconds or 4.39% of the slowest time. The reason for this small percentage in speedup is because of the large amount of time it takes the honeypot to respond to the client's request even when the request is in the protocol tree. It takes about 10 times as long for the honeypot to reply to a client's request than it takes the back-end PLC to reply to the honeypot.

4.5 Metric 4 - Impact of TriggerDepth on PLC Load

4.5.1 In-Tree Results.

The results show the greater the TriggerDepth, the less amount of load on the back-end PLC. These results make sense when considering the design of the Triggerlock Synchronization Algorithm. The algorithm waits to proxy all incoming client requests until the predetermined TriggerDepth is reached. Once this depth is reached, the algorithm looks identical to lockstep with every request being proxied regardless of whether the protocol tree contains the request or not. Any load on the back-end PLC when the protocol tree contains the response to the client query is superfluous.

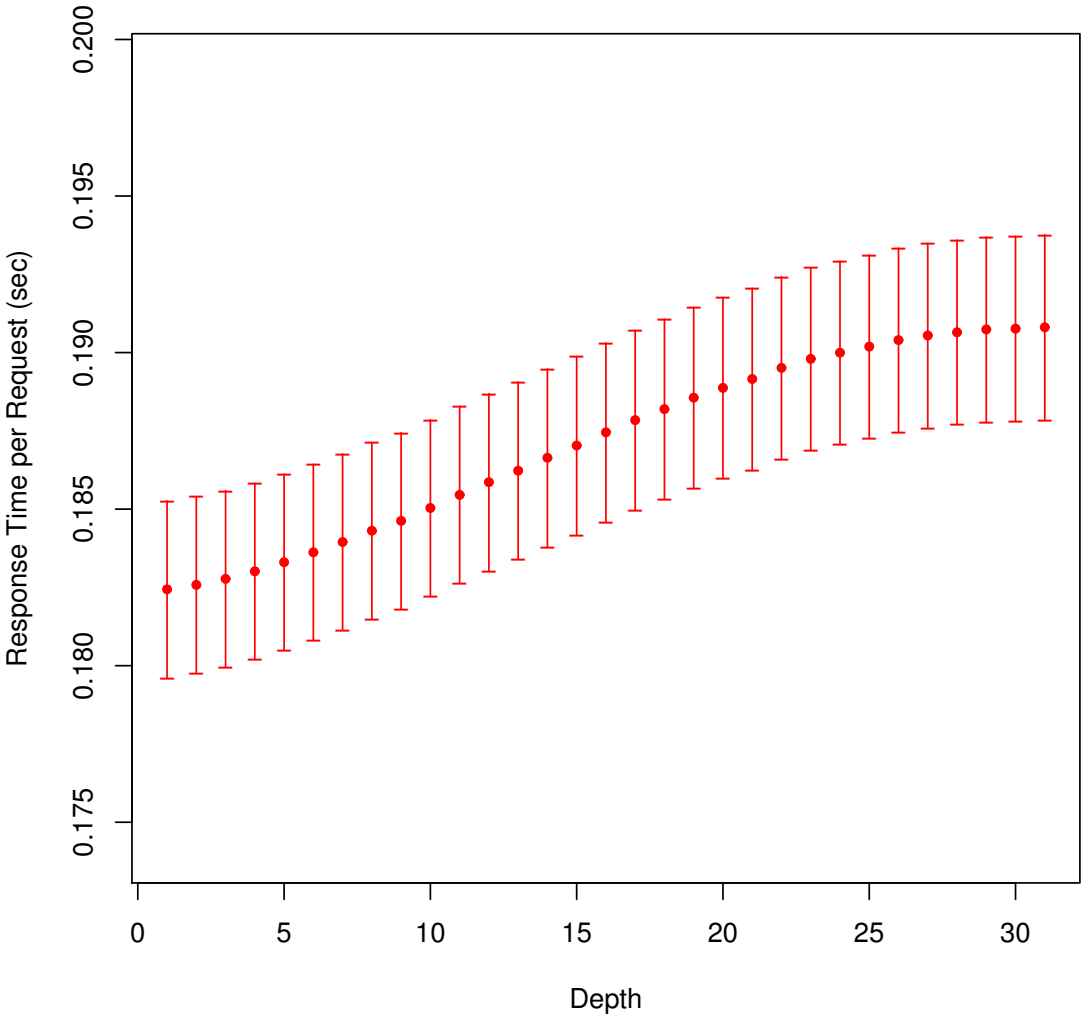

Figure 20. Metric 3 - Tree 2 TriggerDepth Latency Times: Not-In-Tree Scenario

Table 8. Metric 3 - Tree 2 TriggerDepth Latency Times: Not-In-Tree Scenario

TriggerDepth	Time	Standard Error
Depth1	0.1824371	0.0028257
Depth2	0.1825813	0.0028225
Depth3	0.1827725	0.0028194
Depth4	0.1830131	0.0028165
Depth5	0.1833048	0.0028143
Depth6	0.1836215	0.0028138
Depth7	0.1839502	0.0028138
Depth8	0.1843104	0.0028151
Depth9	0.1846254	0.0028166
Depth10	0.1850361	0.0028197
Depth11	0.1854548	0.0028241
Depth12	0.1858605	0.0028293
Depth13	0.1862280	0.0028342
Depth14	0.1866396	0.0028396
Depth15	0.1870305	0.0028457
Depth16	0.1874523	0.0028547
Depth17	0.1878424	0.0028641
Depth18	0.1881930	0.0028719
Depth19	0.1885605	0.0028808
Depth20	0.1888732	0.0028889
Depth21	0.1891565	0.0028963
Depth22	0.1895109	0.0029067
Depth23	0.1897997	0.0029154
Depth24	0.1899961	0.0029216
Depth25	0.1901911	0.0029284
Depth26	0.1903995	0.0029361
Depth27	0.1905457	0.0029416
Depth28	0.1906470	0.0029456
Depth29	0.1907403	0.0029493
Depth30	0.1907645	0.0029503
Depth31	0.1908082	0.0029521

The forwarding is only done in hopes of faster response times upon receiving Not-In-Tree request, which is impossible during the In-Tree Scenario. Figure 21 shows not only does the load decrease as the TriggerDepth increases, but it decreases in an exponential manner. Using Figure 21, increasing the TriggerDepth from 4 to 5 results in a load decrease of 0.1208 requests, which is less than the load decrease from changing TriggerDepth from 24 to 25 of 0.7824. The average load for every TriggerDepth in Tree 2 can be found in Table 9.

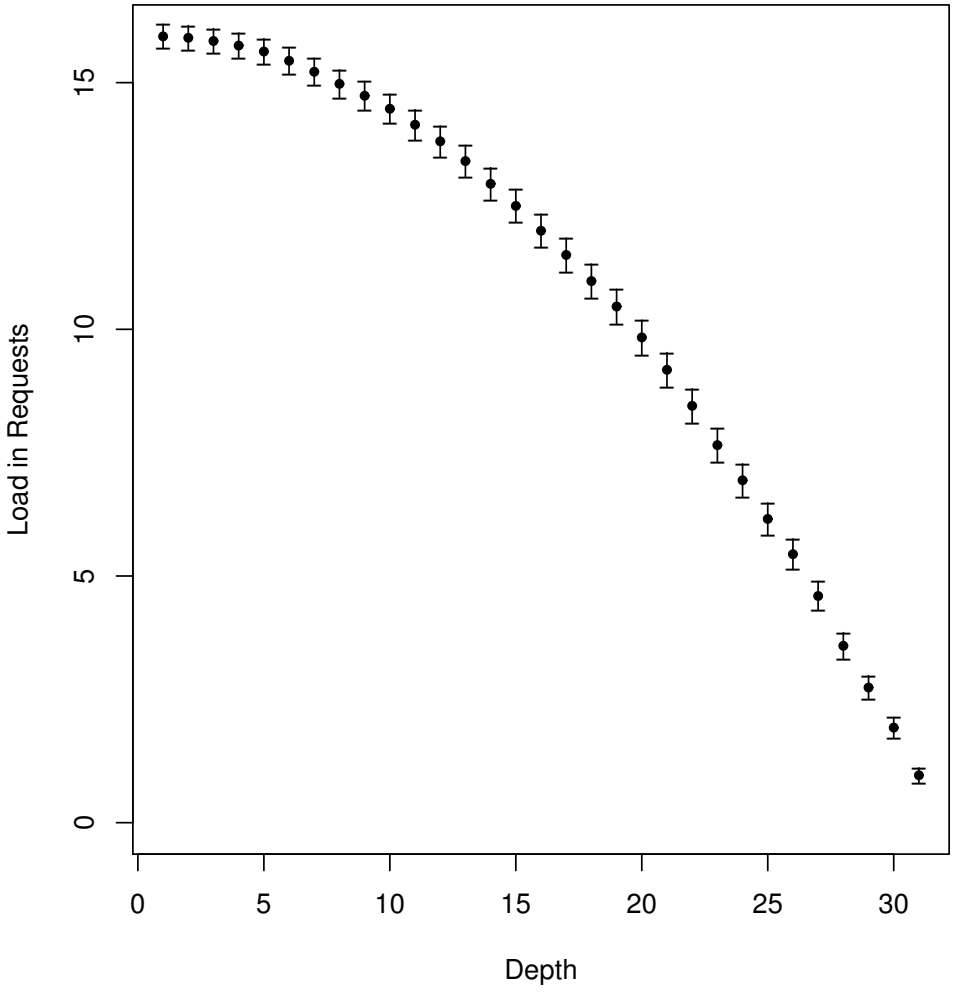

Figure 21. Metric 4 - Tree 2 TriggerDepth PLC Load: In-Tree Scenario

Table 9. Metric 3 - Tree 2 TriggerDepth PLC Load: In-Tree Scenario

TriggerDepth	Load	Standard Error
Depth 1	15.9378	0.2465903
Depth 2	15.9080	0.2480185
Depth 3	15.8444	0.2509329
Depth 4	15.7514	0.2549726
Depth 5	15.6306	0.2599094
Depth 6	15.4436	0.2670204
Depth 7	15.2204	0.2748279
Depth 8	14.9754	0.2826616
Depth 9	14.7338	0.2897036
Depth 10	14.4710	0.2966625
Depth 11	14.1470	0.3043751
Depth 12	13.8104	0.3114973
Depth 13	13.4096	0.3189332
Depth 14	12.9494	0.3262558
Depth 15	12.5014	0.3322425
Depth 16	11.9974	0.3377337
Depth 17	11.5078	0.3418886
Depth 18	10.9774	0.3451429
Depth 19	10.4626	0.3471088
Depth 20	9.83560	0.3480086
Depth 21	9.17960	0.3472946
Depth 22	8.44880	0.3445578
Depth 23	7.65240	0.3392785
Depth 24	6.93940	0.3324198
Depth 25	6.15700	0.3224082
Depth 26	5.44200	0.3107793
Depth 27	4.59440	0.2935741
Depth 28	3.58460	0.2675454
Depth 29	2.73900	0.2397807
Depth 30	1.92700	0.2056948
Depth 31	0.96100	0.1489426

The back-end PLC load is not influenced by the time it takes the honeypot to respond to the client, these results represent the impact the Synchronization Algorithms have on load even in an optimized version of the ScriptGenE Framework.

4.6 Results Summary

Chapter IV presents and analyzes the results from the experiment described in Chapter III. This chapter presents each of the Performance Metrics from Section 3.6 and performs both an observational and interpretive form of analysis.

V. Conclusions

5.1 Introduction

ScriptGenE, as it previously existed, only used one Synchronization Algorithm for all protocol trees - Latelock. While there was an option to set the Sychronization Algorithm to either Catchup, Lockstep, Latelock, or Triggerlock for each individual protocol tree, there was not a way for the administrator setting up the honeypot to know which algorithm works best for the given protocol being implementing. This is the problem this research set out to solve.

5.2 Research Conclusions

This research seeks to solve the research problem by asking the following three questions:

1. Given a protocol tree, is there a Synchronization Algorithm yielding minimized latency and minimized load on the back-end PLC?

2. Does the shape of the protocol tree impact the best Synchronization Algorithm?

3. Is there an optimal setting for the Triggerlock TriggerDepth based on the tree shape and size?

This research answers the three questions through two empirical experiments. The first experiment sought to find the optimal Synchronization Algorithm for each tree and answer questions one and two. This was done by comparing each algorithm's latency times and load on the back-end PLC. These results showed that the Catchup Algorithm is the worst case for almost every protocol tree. Results also showed the Lockstep and Latelock Algorithms have their pros and cons. The Lockstep Algorithm

is able to respond to unknown responses with the lowest latency, but in order to do so it must place the largest load on the back-end PLC. The Latelock Algorithm only has a large latency for the first unknown client request, then the latency drops to the lowest value. However, the algorithm displaying the best results based on the give and take of latency time and back-end PLC load is the Triggerlock Algorithm. These results give the answer to the first research question of: yes, given a protocol tree, there is a Synchronization Algorithm which yields minimized latency and load on the back-end PLC. The answer to the second research question is also yes, because as the experiment has shown, the shape of the tree determines which algorithm should be utilized.

The prospective results of the Triggerlock Algorithm led to the third research question and the second experiment. The second experiment set out to determine how changing the Triggerdepth impacted the latency time and the back-end PLC load. These results show in the Triggerlock Algorithm, as the TriggerDepth increased, the Latency Time also increased, but the back-end PLC load decreased. These results were expected because of how the Triggerlock Algorithm is designed to work, but the more interesting results from this experiment, were the load on the back-end PLC decreased exponentially as the TriggerDepth was increased. These results answered the third research question.

Through experimentation all the original research questions are answered and it is shown for almost all trees (ten out of eleven) the best algorithm is Triggerlock. The only exception is stateless trees, which all have a depth of one, where the best algorithm is Catchup because of its instant disconnection from the back-end PLC after receiving the response to the forwarded request. Additionally, through answering the three research questions, the research hypothesis was proven correct.

Selecting a Synchronization Algorithm based on the shape and size of the protocol tree will yield faster latency response times and a smaller load on the back-end PLC than the existing default Synchronization Algorithm setting of Latelock.

5.3 Significance of Research

This research helps further improve the ScriptGenE Framework previously researched by Warner and Girtz. ScriptGenE now analyzes the protocol tree as part of the initial honeypot setup process. After the protocol tree is created, ScriptGenE analyzes the shape and size of the tree. Then sets the Synchronization Algorithm to the corresponding algorithm based on the results from this research. The changes made to ScriptGenE based on this research improve the software by decreasing the latency times when the honeypot receives a request from the client it does not have in the protocol tree and decreasing the load on the back-end PLC when the protocol tree contains the response to the client request.

5.4 Limitations of Research

One of the main limitations to this research is the amount of speedup possible from reducing the latency. As previously discussed in Chapter IV, the time it takes to send a request and receive a response from the back-end PLC takes about a tenth of the time it takes the honeypot to receive a request from the client and respond to the request. Because of this time difference there is not a large penalty to having to proxy a large amount of requests in order to synchronize the honeypot with the back-end PLC. Any speed up from a Synchronization Algorithm leveraging the constant connection to the back-end PLC is limited. While it does provide a speedup to the average latency time, it will not halve the latency time or perform a speedup of greater

than 100%. If the ScriptGenE Framework is optimized further and the time difference between the honeypot response and the back-end PLC response is lessened, then the impact of this research is increased because the time to synchronize the honeypot to the back-end PLC is heavily weighted in the results of this research.

5.5 Future Work

- A possible follow-on to this research would be to make the ScriptGenE Framework faster at responding to client's requests when the protocol tree contains the response. This research could be done by optimizing the tree search and optimizing the software to reduce the amount of code needed to respond to the client's requests.

- Another possible follow-on could be to test the ScriptGenE Framework with non-PLC protocols to truly test the protocol-agnosticism. This research could be done by selecting various protocols and testing their accuracy in responses against the actual systems they are emulating.

- A final possible follow-on could be to test the impact of having multiple ScriptGenE honeypots all using the same back-end PLC. The goal of this test would be to see how much load is placed on the back-end PLC per ScriptGenE instance and to test whether the PLC can keep track of every connection accurately.

5.6 Chapter Summary

The problem this research attempted to solve was finding an optimal Synchronization Algorithm to synchronize a honeypot running the ScriptGenE Framework with a back-end PLC. Through the formulation and use of an experimental methodology, a more efficient ScriptGenE Honeypot Framework has been created.

Appendix A. Protocol Trees

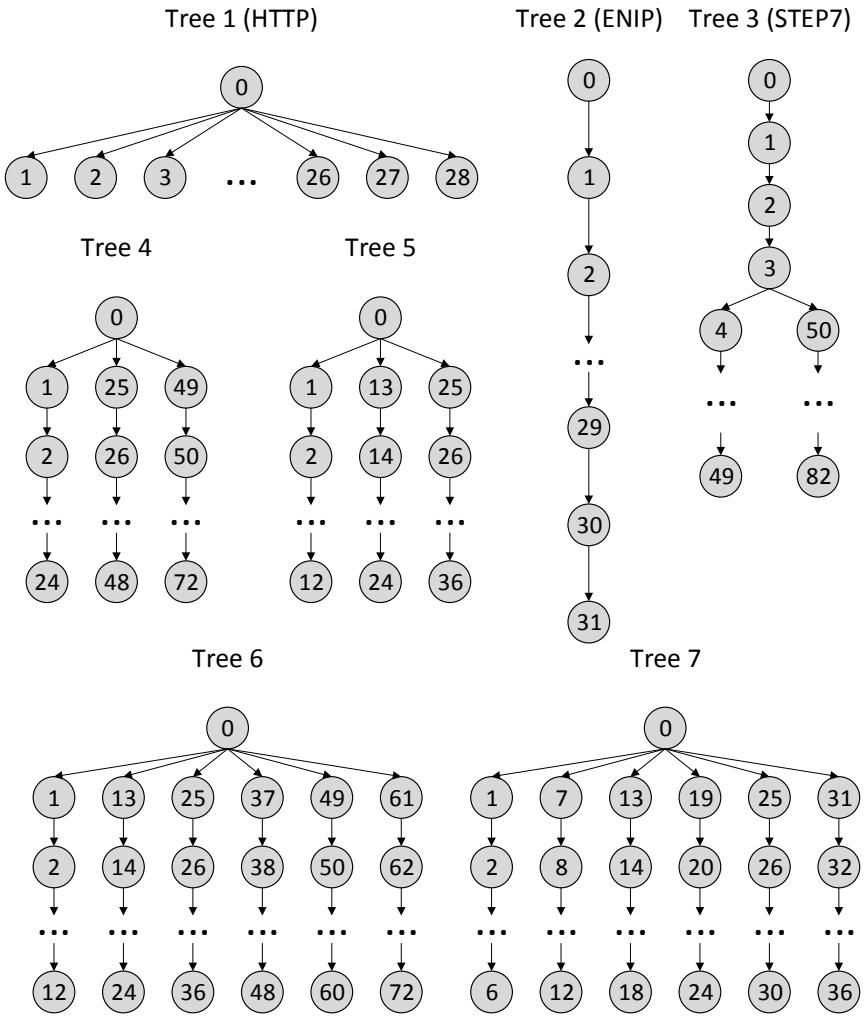

Figure 22. Protocol Trees 1 through 7

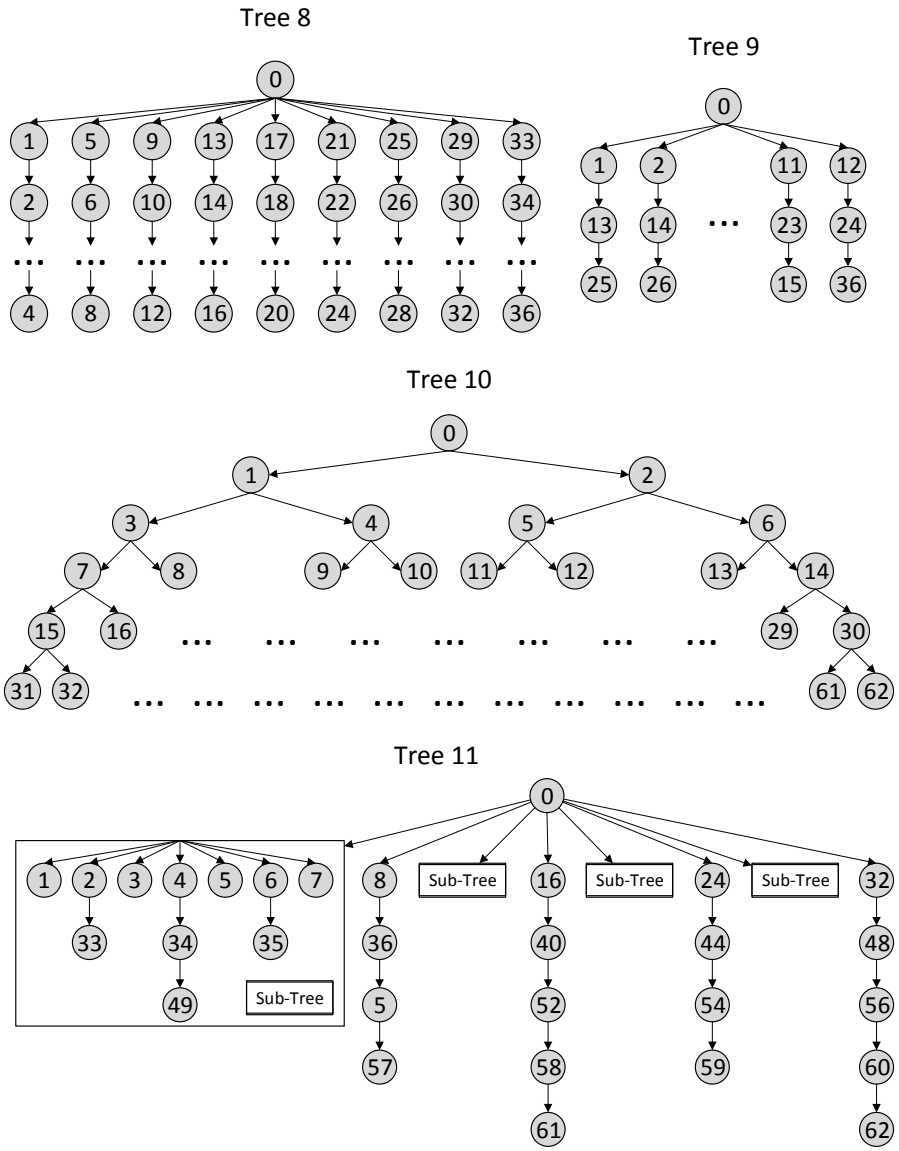

Figure 23. Protocol Trees 8 through 11

Appendix B. R Code

```
1   trigFinder<-function(tree)
2   {
3       library(iterators)
4
5       treeIterator = iter(tree)
6       counter = 0
7       trigCols = c()
8       trigErrs = c()
9       depth = tree$Depth[1]
10      tableSize = (depth*2)
11
12      # Iterate through the triggerdepth results to
        find standard error for the means
13      for(i in 1:tableSize)
14      {
15          col = nextElem(treeIterator)
16          trigCols = c(trigCols, mean(col))
17          trigErrs = c(trigErrs,
            (1.96*(sd(col)/sqrt(length(col)))))
18      }
19
20      trigTimeErr = trigErrs[1:depth]
21      trigLoadErr = trigErrs[(depth+1):(depth*2)]
22
23      trigTime = trigCols[1:depth]
24      trigLoad = trigCols[(depth+1):(depth*2)]
25
26      results = list('time' = trigTime, 'load' =
        trigLoad, 'timeErr' = trigTimeErr, 'loadErr' =
        trigLoadErr)
27      return(results)
28  }
29
30  trigMaker<-function(trigIn, trigNotIn, preface, depth,
    treeName)
31  {
32      trigInTime = trigIn$time
33      trigInLoad = trigIn$load
34      trigInLoadErr = trigIn$loadErr
35
36      trigNotTime = trigNotIn$time
37      trigNotLoad = trigNotIn$load
38      trigNotTimeErr = trigNotIn$timeErr
39
```

```
40          # create matrix of triggerdepth means with
            standard errors for export to CSV
41          table = matrix(c(trigInTime, trigInLoad,
            trigInLoadErr, trigNotTime, trigNotLoad,
            trigNotTimeErr), ncol = 6, byrow = FALSE)
42          rows = c()
43          for(i in 1:(depth))
44          {
45                name = paste0('Depth', i)
46                rows = c(rows, name)
47          }
48
49          rownames(table) = rows
50          colnames(table) = c('InTreeTime', 'InTreeLoad',
            'InTreeLoadErr', 'NotFoundTime', 'NotFoundLoad',
            'NotFoundTimeErr')
51          filename = paste0(preface, 'results/http/',
            treeName, '/Trig.csv')
52          write.csv(table, file = filename)
53    }
54
55    trigGrapherLoad<-function(filename, tree, titleName)
56    {
57          setEPS()
58          postscript(filename, width = 7, height = 7)
59
60          # set the border parameters to ensure enough
            space for labels
61          par(mar = c(5,5,4,5)+0.1)
62
63          # create x variable, which is a set of integers
            starting at 1 to the max triggerdepth
64          x = c()
65          depth = tree$depth
66          for(i in 1:(depth))
67          {
68                x = c(x, i)
69          }
70
71          # TriggerDepth Load Graph
72
73          y1 = tree$InTreeLoad
74          y1.1 = tree$InTreeLoad + tree$InTreeLoadErr
75          y1.2 = tree$InTreeLoad - tree$InTreeLoadErr
76
77          plot(x, y1,
```

```
78                  xlab = 'Depth',
79                  ylab = 'Load in Requests',
80                  cex = 1,
81                  pch = 20,
82                  ylim = c(0,max(y1)))
83            points(x, y1.1, col = 'black', pch = "-")
84            points(x, y1.2, col = 'black', pch = "-")
85            segments(x, y1.1, x, y1, col = 'black')
86            segments(x, y1.2, x, y1, col = 'black')
87
88            dev.off()
89      }
90
91      #### ^ Load graph ^ ######## v Time graph v ####
92
93      trigGrapherTime<-function(filename, tree, titleName)
94      {
95            setEPS()
96            postscript(filename, width = 7, height = 7)
97
98            # create x variable, which is a set of integers
              starting at 1 to the max triggerdepth
99            x = c()
100           depth = tree$depth
101           for(i in 1:(depth))
102           {
103                 x = c(x, i)
104           }
105
106           # TriggerDepth Time Graph
107
108           y1 = tree$NotFoundTime/(depth/2)
109           y1.1 = (tree$NotFoundTime +
              tree$NotFoundTimeErr)/(depth/2)
110           y1.2 = (tree$NotFoundTime -
              tree$NotFoundTimeErr)/(depth/2)
111
112           ylimiter = max(y1) - min(y1)
113
114           plot(x, y1,
115                 xlab = 'Depth',
116                 ylab = 'Response Time per Request (Sec)',
117                 cex = 1,
118                 pch = 20,
119                 col = 'red',
```

```
120                 ylim =
                    c((min(y1)-ylimiter),(max(y1)+ylimiter)))
121           points(x, y1.1, col = 'red', pch = "-")
122           points(x, y1.2, col = 'red', pch = "-")
123           segments(x, y1.1, x, y1, col = 'red')
124           segments(x, y1.2, x, y1, col = 'red')
125
126           dev.off()
127     }
128
129     # creates a column in the final csv for each tree
130     columnBuilder<-function(tree)
131     {
132           CatchupTimes = tree$CatchupTimes/tree$ExpDepth
133           LockstepTimes = tree$LockstepTimes/tree$ExpDepth
134           LatelockTimes = tree$LatelockTimes/tree$ExpDepth
135           TriggerlockLowTimes =
              tree$TriggerlockTimes1/tree$ExpDepth
136           CatchupTime = mean(CatchupTimes)
137           LockstepTime = mean(LockstepTimes)
138           LatelockTime = mean(LatelockTimes)
139           TriggerlockLowTime = mean(TriggerlockLowTimes)
140           TriggerlockHighTime = mean(TriggerlockHighTimes)
141           CatchupLoads = tree$CatchupLoads/tree$ExpDepth
142           LockstepLoads = tree$LockstepLoads/tree$ExpDepth
143           LatelockLoads = tree$LatelockLoads/tree$ExpDepth
144           TriggerlockHighLoads =
              tree$TriggerlockLoads1/tree$ExpDepth
145           CatchupLoad = mean(CatchupLoads)
146           LockstepLoad = mean(LockstepLoads)
147           LatelockLoad = mean(LatelockLoads)
148           TriggerlockHighLoad = mean(TriggerlockHighLoads)
149           TriggerlockLowLoad = mean(TriggerlockLowLoads)
150           CatchupTimeErr = (1.96*sd(CatchupTimes)/
              sqrt(length(tree$CatchupTimes)))
151           LockstepTimeErr = (1.96*sd(LockstepTimes)/
              sqrt(length(tree$CatchupTimes)))
152           LatelockTimeErr = (1.96*sd(LatelockTimes)/
              sqrt(length(tree$CatchupTimes)))
153           TriggerlockLowTimeErr =
              (1.96*sd(TriggerlockLowTimes)/
              sqrt(length(tree$CatchupTimes)))
154           TriggerlockHighTimeErr =
              (1.96*sd(TriggerlockHighTimes)/
              sqrt(length(tree$CatchupTimes)))
```

```
155             CatchupLoadErr = (1.96*sd(CatchupLoads)/
                sqrt(length(tree$CatchupTimes)))
156             LockstepLoadErr = (1.96*sd(LockstepLoads)/
                sqrt(length(tree$CatchupTimes)))
157             LatelockLoadErr = (1.96*sd(LatelockLoads)/
                sqrt(length(tree$CatchupTimes)))
158             TriggerlockLowLoadErr =
                (1.96*sd(TriggerlockLowLoads)/
                sqrt(length(tree$CatchupTimes)))
159             TriggerlockHighLoadErr =
                (1.96*sd(TriggerlockHighLoads)/
                sqrt(length(tree$CatchupTimes)))
160             Depth = tree$Depth[1]
161
162             column = c(CatchupTime, LockstepTime,
                LatelockTime, TriggerlockLowTime,
                TriggerlockHighTime,
163                   CatchupLoad, LockstepLoad, LatelockLoad,
                      TriggerlockLowLoad, TriggerlockHighLoad,
164                   CatchupTimeErr, LockstepTimeErr,
                      LatelockTimeErr, TriggerlockLowTimeErr,
                      TriggerlockHighTimeErr,
165                   CatchupLoadErr, LockstepLoadErr,
                      LatelockLoadErr, TriggerlockLowLoadErr,
                      TriggerlockHighLoadErr,
166                   Depth)
167
168             return(column)
169       }
170
171       grapher<-function(filename, tree, titleName, type)
172       {
173             setEPS()
174             postscript(filename, width = 7, height = 7)
175
176             # set graph border parameters
177             par(mar = c(6,5,4,2)+0.1)
178
179             # set different x variables for each algorithm,
                so they can appear next to each other not on top
                of each other
180             # x, x2, x3, and x4 are Catchup, Lockstep,
                Latelock, and Triggerlock respectively
181             x = c(.7, 1.7, 2.7, 3.7, 4.7, 5.7, 6.7, 7.7, 8.7,
                9.7, 10.7)
```

```
182             x2 =
                c(.9,1.9,2.9,3.9,4.9,5.9,6.9,7.9,8.9,9.9,10.9)
183             x3 = c(1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1, 8.1,
                9.1, 10.1, 11.1)
184             x4 = c(1.3, 2.3, 3.3, 4.3, 5.3, 6.3, 7.3, 8.3,
                9.3, 10.3, 11.3)
185
186             if(type == 'Times')
187             {
188                   y1 = tree$CatchupTime
189                   y1.2 = tree$CatchupTime +
                      tree$CatchupTimeErr
190                   y1.3 = tree$CatchupTime -
                      tree$CatchupTimeErr
191                   y2 = tree$LockstepTime
192                   y2.2 = tree$LockstepTime +
                      tree$LockstepTimeErr
193                   y2.3 = tree$LockstepTime -
                      tree$LockstepTimeErr
194                   y3 = tree$LatelockTime
195                   y3.2 = tree$LatelockTime +
                      tree$LatelockTimeErr
196                   y3.3 = tree$LatelockTime -
                      tree$LatelockTimeErr
197                   y4H.2 = tree$TriggerlockHighTime +
                      tree$TriggerlockHighTimeErr
198                   y4H = tree$TriggerlockHighTime
199                   y4L = tree$TriggerlockLowTime
200                   y4L.3 = tree$TriggerlockLowTime -
                      tree$TriggerlockLowTimeErr
201                   lab = 'Latency Time per Client Request
                      (sec)'
202             } else{
203                   y1 = tree$CatchupLoad
204                   y1.2 = tree$CatchupLoad +
                      tree$CatchupLoadErr
205                   y1.3 = tree$CatchupLoad -
                      tree$CatchupLoadErr
206                   y2 = tree$LockstepLoad
207                   y2.2 = tree$LockstepLoad +
                      tree$LockstepLoadErr
208                   y2.3 = tree$LockstepLoad -
                      tree$LockstepLoadErr
209                   y3 = tree$LatelockLoad
210                   y3.2 = tree$LatelockLoad +
                      tree$LatelockLoadErr
```

```
211                 y3.3 = tree$LatelockLoad -
                    tree$LatelockLoadErr
212                 y4H.2 = tree$TriggerlockHighLoad +
                    tree$TriggerlockHighLoadErr
213                 y4H = tree$TriggerlockHighLoad
214                 y4L = tree$TriggerlockLowLoad
215                 y4L.3 = tree$TriggerlockLowLoad -
                    tree$TriggerlockLowLoadErr
216                 lab = 'Load per Client Request'
217           }
218
219           # Used for post python script editing
220           preface =
              'Desktop/ScriptGenE_Framework_v1.1_201603/'
221
222           # Custom settings for each of the four main graphs
223           if(filename == paste0(preface,
              'results/InTreeGraphTimes.eps'))
224           {
225                 yrange = c(0, .05, 0.1, 0.15, 0.2, 0.25,
                    0.3)
226
227                 plot(x, y1,
228                       xlab = '', ylab = lab,
229                       pch = 19, ylim = c(min(yrange),
                          max(yrange)),
230                 xlim = c(0.5,11.5), axes = FALSE)
231                 points(x2, y2, col = 'red', pch = 15)
232                 points(x3, y3, col = 'blue', pch = 17)
233                 points(x4, y3, col = 'darkgreen', pch = 18,
                    cex = 1.25)
234
235                 legend('bottomright', inset = .05, title =
                    'Algorithms',
236                       c('Catchup', 'Lockstep', 'Latelock',
                          'Triggerlock'),
237                       pch = c(19, 15, 17, 18),
238                       col = c('black', 'red', 'blue',
                          'darkgreen'))
239
240           } else if(filename == paste0(preface,
              'results/NotInTreeGraphTimes.eps'))
241           {
242                 yrange = c(0, 0.1, 0.2, 0.3)
243
244                 plot(x, y1,
```

```
245                         xlab = '', ylab = lab,
246                         pch = 19, ylim = c(min(yrange),
                            max(yrange)),
247                   xlim = c(0.5,11.5), axes = FALSE)
248                   points(x2, y2, col = 'red', pch = 15)
249                   points(x3, y3, col = 'blue', pch = 17)
250                   points(x4, y4H.2, col = 'darkgreen', pch =
                      18)
251                   points(x4, y4L.3, col = 'darkgreen', pch =
                      18)
252                   segments(x4, y4H.2, x4, y4L.3, col =
                      'darkgreen')
253
254                   legend('bottomright', inset = .05, title =
                      'Algorithms',
255                         c('Catchup', 'Lockstep', 'Latelock',
                            'Triggerlock'),
256                         pch = c(19, 15, 17, 18),
257                         col = c('black', 'red', 'blue',
                            'darkgreen'))
258
259             } else if(filename == paste0(preface,
                'results/InTreeGraphLoads.eps'))
260             {
261                   yrange = c(0, 0.2, 0.4, 0.6, 0.8, 1, 1.2,
                      1.4, 1.6)
262
263                   plot(x, y1,
264                         xlab = '', ylab = lab,
265                         pch = 19, ylim = c(min(yrange),
                            max(yrange)),
266                   xlim = c(0.5,11.5), axes = FALSE)
267                   points(x2, y2, col = 'red', pch = 15)
268                   points(x3, y3, col = 'blue', pch = 17)
269                   points(x4, y4H.2, col = 'darkgreen', pch =
                      18)
270                   points(x4, y4L.3, col = 'darkgreen', pch =
                      18)
271                   segments(x4, y4H.2, x4, y4L.3, col =
                      'darkgreen')
272
273                   legend('topright', inset = .05, title =
                      'Algorithms',
274                         c('Catchup', 'Lockstep', 'Latelock',
                            'Triggerlock'),
275                         pch = c(19, 15, 17, 18),
```

```
276                         col = c('black', 'red', 'blue',
                            'darkgreen'))
277
278               } else if(filename == paste0(preface,
                  'results/NotInTreeGraphLoads.eps'))
279               {
280                     yrange = c(0, 1, 2, 3, 4, 5, 6, 7, 8)
281
282                     plot(x, y1,
283                           xlab = '', ylab = lab,
284                           pch = 19, ylim = c(min(yrange),
                              max(yrange)),
285                     xlim = c(0.5,11.5), axes = FALSE)
286                     points(x, y1.2, col = 'black', pch = "-")
287                     points(x, y1.3, col = 'black', pch = "-")
288                     segments(x, y1.2, x, y1, col = 'black')
289                     segments(x, y1.3, x, y1, col = 'black')
290                     points(x2, y2, col = 'red', pch = 15)
291                     points(x3, y3, col = 'blue', pch = 17)
292                     points(x4, y4L.3, col = 'darkgreen', pch =
                        18)
293
294                     legend('topright', inset = .05, title =
                        'Algorithms',
295                           c('Catchup', 'Lockstep', 'Latelock',
                              'Triggerlock'),
296                           pch = c(19, 15, 17, 18),
297                           col = c('black', 'red', 'blue',
                              'darkgreen'))
298               }
299
300               axis(1, at = x2, labels = c('Tree1', 'Tree2',
                  'Tree3', 'Tree4', 'Tree5', 'Tree6', 'Tree7',
                  'Tree8', 'Tree9', 'Tree10', 'Tree11'), las = 2)
301               axis(2, yrange, las = 3)
302               mtext('Trees', side = 1, line = 4)
303               box()
304               dev.off()
305         }
306
307         r_experiment<-function(type)
308         {
309               if(type == 'Python')
310               {
311                     preface = ''
312               } else {
```

```
313                 preface =
                    'Desktop/ScriptGenE_Framework_v1.1_201603/'
314             }
315             tree1InTree <- read.csv(paste0(preface,
                'results/http/tree01/In.csv'))
316             tree2InTree <- read.csv(paste0(preface,
                'results/http/tree02/In.csv'))
317             tree3InTree <- read.csv(paste0(preface,
                'results/http/tree03/In.csv'))
318             tree4InTree <- read.csv(paste0(preface,
                'results/http/tree04/In.csv'))
319             tree5InTree <- read.csv(paste0(preface,
                'results/http/tree05/In.csv'))
320             tree6InTree <- read.csv(paste0(preface,
                'results/http/tree06/In.csv'))
321             tree7InTree <- read.csv(paste0(preface,
                'results/http/tree07/In.csv'))
322             tree8InTree <- read.csv(paste0(preface,
                'results/http/tree08/In.csv'))
323             tree9InTree <- read.csv(paste0(preface,
                'results/http/tree09/In.csv'))
324             tree10InTree <- read.csv(paste0(preface,
                'results/http/tree10/In.csv'))
325             tree11InTree <- read.csv(paste0(preface,
                'results/http/tree11/In.csv'))
326
327             tree1NotInTree <- read.csv(paste0(preface,
                'results/http/tree01/NotIn.csv'))
328             tree2NotInTree <- read.csv(paste0(preface,
                'results/http/tree02/NotIn.csv'))
329             tree3NotInTree <- read.csv(paste0(preface,
                'results/http/tree03/NotIn.csv'))
330             tree4NotInTree <- read.csv(paste0(preface,
                'results/http/tree04/NotIn.csv'))
331             tree5NotInTree <- read.csv(paste0(preface,
                'results/http/tree05/NotIn.csv'))
332             tree6NotInTree <- read.csv(paste0(preface,
                'results/http/tree06/NotIn.csv'))
333             tree7NotInTree <- read.csv(paste0(preface,
                'results/http/tree07/NotIn.csv'))
334             tree8NotInTree <- read.csv(paste0(preface,
                'results/http/tree08/NotIn.csv'))
335             tree9NotInTree <- read.csv(paste0(preface,
                'results/http/tree09/NotIn.csv'))
336             tree10NotInTree <- read.csv(paste0(preface,
                'results/http/tree10/NotIn.csv'))
```

```
337             tree11NotInTree <- read.csv(paste0(preface,
                'results/http/tree11/NotIn.csv'))
338
339             writefileInTree = paste0(preface,
                'results/InTreeResults.csv')
340             writefileNotInTree = paste0(preface,
                'results/NotInTreeResults.csv')
341
342             # Create the eleven In-Tree columns for the
                eleven trees
343             colBuilder = columnBuilder(tree1InTree)
344             colBuilder = c(colBuilder,
                columnBuilder(tree2InTree))
345             colBuilder = c(colBuilder,
                columnBuilder(tree3InTree))
346             colBuilder = c(colBuilder,
                columnBuilder(tree4InTree))
347             colBuilder = c(colBuilder,
                columnBuilder(tree5InTree))
348             colBuilder = c(colBuilder,
                columnBuilder(tree6InTree))
349             colBuilder = c(colBuilder,
                columnBuilder(tree7InTree))
350             colBuilder = c(colBuilder,
                columnBuilder(tree8InTree))
351             colBuilder = c(colBuilder,
                columnBuilder(tree9InTree))
352             colBuilder = c(colBuilder,
                columnBuilder(tree10InTree))
353             colBuilder = c(colBuilder,
                columnBuilder(tree11InTree))
354
355             # create In-Tree matrix for export to CSV
356             InTreeTable = matrix(colBuilder, ncol=21,
                byrow=TRUE)
357             rownames(InTreeTable) = c('Tree1', 'Tree2',
                'Tree3', 'Tree4', 'Tree5', 'Tree6', 'Tree7',
                'Tree8', 'Tree9', 'Tree10', 'Tree11')
358             colnames(InTreeTable) = c('CatchupTime',
                'LockstepTime', 'LatelockTime',
                'TriggerlockLowTime', 'TriggerlockHighTime',
359                   'CatchupLoad', 'LockstepLoad',
                      'LatelockLoad', 'TriggerlockLowLoad',
                      'TriggerlockHighLoad',
```

```
360                 'CatchupTimeErr', 'LockstepTimeErr',
                    'LatelockTimeErr', 'TriggerlockLowTimeErr',
                    'TriggerlockHighTimeErr',
361                 'CatchupLoadErr', 'LockstepLoadErr',
                    'LatelockLoadErr', 'TriggerlockLowLoadErr',
                    'TriggerlockHighLoadErr',
362                 'Depth')
363
364           write.csv(InTreeTable, file = writefileInTree)
365
366           # Create eleven Not-In-Tree columns for the
              eleven trees
367           colBuilder = columnBuilder(tree1NotInTree)
368           colBuilder = c(colBuilder,
              columnBuilder(tree2NotInTree))
369           colBuilder = c(colBuilder,
              columnBuilder(tree3NotInTree))
370           colBuilder = c(colBuilder,
              columnBuilder(tree4NotInTree))
371           colBuilder = c(colBuilder,
              columnBuilder(tree5NotInTree))
372           colBuilder = c(colBuilder,
              columnBuilder(tree6NotInTree))
373           colBuilder = c(colBuilder,
              columnBuilder(tree7NotInTree))
374           colBuilder = c(colBuilder,
              columnBuilder(tree8NotInTree))
375           colBuilder = c(colBuilder,
              columnBuilder(tree9NotInTree))
376           colBuilder = c(colBuilder,
              columnBuilder(tree10NotInTree))
377           colBuilder = c(colBuilder,
              columnBuilder(tree11NotInTree))
378
379           # Create the matrix for export to CSV
380           NotInTreeTable = matrix(colBuilder, ncol=21,
              byrow=TRUE)
381           rownames(NotInTreeTable) = c('Tree1', 'Tree2',
              'Tree3', 'Tree4', 'Tree5', 'Tree6', 'Tree7',
              'Tree8', 'Tree9', 'Tree10', 'Tree11')
382           colnames(NotInTreeTable) = c('CatchupTime',
              'LockstepTime', 'LatelockTime',
              'TriggerlockLowTime', 'TriggerlockHighTime',
383                 'CatchupLoad', 'LockstepLoad',
                    'LatelockLoad', 'TriggerlockLowLoad',
                    'TriggerlockHighLoad',
```

```
384                 'CatchupTimeErr', 'LockstepTimeErr',
                    'LatelockTimeErr', 'TriggerlockLowTimeErr',
                    'TriggerlockHighTimeErr',
385                 'CatchupLoadErr', 'LockstepLoadErr',
                    'LatelockLoadErr', 'TriggerlockLowLoadErr',
                    'TriggerlockHighLoadErr',
386                 'Depth')
387
388         write.csv(NotInTreeTable, file =
            writefileNotInTree)
389
390         # Format the raw Trigger depth data for the
            graphing function
391         trigTree1InTree = trigFinder(tree1InTree)
392         trigTree1NotInTree = trigFinder(tree1NotInTree)
393         trigMaker(trigTree1InTree, trigTree1NotInTree,
            preface, tree1InTree$Depth[1],
            tree1InTree$TreeName[1])
394
395         trigTree2InTree = trigFinder(tree2InTree)
396         trigTree2NotInTree = trigFinder(tree2NotInTree)
397         trigMaker(trigTree2InTree, trigTree2NotInTree,
            preface, tree2InTree$Depth[1],
            tree2InTree$TreeName[1])
398
399         trigTree3InTree = trigFinder(tree3InTree)
400         trigTree3NotInTree = trigFinder(tree3NotInTree)
401         trigMaker(trigTree3InTree, trigTree3NotInTree,
            preface, tree3InTree$Depth[1],
            tree3InTree$TreeName[1])
402
403         trigTree4InTree = trigFinder(tree4InTree)
404         trigTree4NotInTree = trigFinder(tree4NotInTree)
405         trigMaker(trigTree4InTree, trigTree4NotInTree,
            preface, tree4InTree$Depth[1],
            tree4InTree$TreeName[1])
406
407         trigTree5InTree = trigFinder(tree5InTree)
408         trigTree5NotInTree = trigFinder(tree5NotInTree)
409         trigMaker(trigTree5InTree, trigTree5NotInTree,
            preface, tree5InTree$Depth[1],
            tree5InTree$TreeName[1])
410
411         trigTree6InTree = trigFinder(tree6InTree)
412         trigTree6NotInTree = trigFinder(tree6NotInTree)
```

```
413             trigMaker(trigTree6InTree, trigTree6NotInTree,
                preface, tree6InTree$Depth[1],
                tree6InTree$TreeName[1])
414
415             trigTree7InTree = trigFinder(tree7InTree)
416             trigTree7NotInTree = trigFinder(tree7NotInTree)
417             trigMaker(trigTree7InTree, trigTree7NotInTree,
                preface, tree7InTree$Depth[1],
                tree7InTree$TreeName[1])
418
419             trigTree8InTree = trigFinder(tree8InTree)
420             trigTree8NotInTree = trigFinder(tree8NotInTree)
421             trigMaker(trigTree8InTree, trigTree8NotInTree,
                preface, tree8InTree$Depth[1],
                tree8InTree$TreeName[1])
422
423             trigTree9InTree = trigFinder(tree9InTree)
424             trigTree9NotInTree = trigFinder(tree9NotInTree)
425             trigMaker(trigTree9InTree, trigTree9NotInTree,
                preface, tree9InTree$Depth[1],
                tree9InTree$TreeName[1])
426
427             trigTree10InTree = trigFinder(tree10InTree)
428             trigTree10NotInTree = trigFinder(tree10NotInTree)
429             trigMaker(trigTree10InTree, trigTree10NotInTree,
                preface, tree10InTree$Depth[1],
                tree10InTree$TreeName[1])
430
431             trigTree11InTree = trigFinder(tree11InTree)
432             trigTree11NotInTree = trigFinder(tree11NotInTree)
433             trigMaker(trigTree11InTree, trigTree11NotInTree,
                preface, tree11InTree$Depth[1],
                tree11InTree$TreeName[1])
434
435             #########
436
437             filename = paste0(preface,
                'results/InTreeResults.csv')
438             inTree<-read.csv(filename)
439             filename = paste0(preface,
                'results/NotInTreeResults.csv')
440             notInTree = read.csv(filename)
441
442             inTreeTimesGraph = paste0(preface,
                'results/InTreeGraphTimes.eps')
```

```
443             notInTreetimesGraph = paste0(preface,
                'results/NotInTreeGraphTimes.eps')
444             inTreeLoadsGraph = paste0(preface,
                'results/InTreeGraphLoads.eps')
445             notInTreeLoadsGraph = paste0(preface,
                'results/NotInTreeGraphLoads.eps')
446
447             grapher(inTreeTimesGraph, inTree, 'In-Tree
                Algorithm Times', 'Times')
448             grapher(notInTreetimesGraph, notInTree,
                'Not-In-Tree Algorithm Times', 'Times')
449             grapher(inTreeLoadsGraph, inTree, 'In-Tree
                Algorithm Loads', 'Loads')
450             grapher(notInTreeLoadsGraph, notInTree,
                'Not-In-Tree Algorithm Loads', 'Loads')
451
452             #graphs for trigger depth
453             for (i in 1:11)
454             {
455                   if(i < 10)
456                   {
457                         trigGrapherLoad(paste0(preface,
                            'results/http/tree0', i,
                            '/trigGraphLoad.eps'),
458                         read.csv(paste0(preface,
                            'results/http/tree0', i,
                            '/Trig.csv')),
459                         paste0('Tree ', i, ' TriggerDepth'))
460
461                         trigGrapherTime(paste0(preface,
                            'results/http/tree0', i,
                            '/trigGraphTime.eps'),
462                         read.csv(paste0(preface,
                            'results/http/tree0', i,
                            '/Trig.csv')),
463                         paste0('Tree ', i, ' TriggerDepth'))
464                   } else {
465                         trigGrapherLoad(paste0(preface,
                            'results/http/tree', i,
                            '/trigGraphLoad.eps'),
466                         read.csv(paste0(preface,
                            'results/http/tree', i, '/Trig.csv')),
467                         paste0('Tree ', i, ' TriggerDepth'))
468
```

```
469                         trigGrapherTime(paste0(preface,
                            'results/http/tree', i,
                            '/trigGraphTime.eps'),
470                         read.csv(paste0(preface,
                            'results/http/tree', i, '/Trig.csv')),
471                         paste0('Tree ', i, ' TriggerDepth'))
472                 }
473             }
474         }
475         #r_experiment('')
476         r_experiment('Python')
```

Bibliography

1. Kyle A. Girtz. Dynamic honeypot configuration for programmable logic controller emulation. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2016. *https://www.afit.edu/docs/AFIT-ENG-MS-16-M-253.pdf*.

2. White House Office of the Press Secretary. Presidential Policy Directive 21. *Critical Infrastructure Security and Resilience*, 2013. Retrieved 23 June 2015 from *http://www.whitehouse.gov/the-press-office/2013/02/12/presidential-policy-directive-critical-infrastructure-security-and-resil*.

3. Keith A. Stouffer, Joseph A. Falco, and Karen A. Scarfone. Guide to Industrial Control Systems (ICS) security: Supervisory Control and Data Acquisition (SCADA) systems, Distributed Control Systems (DCS), and other control system configurations such as Programmable Logic Controllers (PLC). Technical Report SP 800-82, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011. Retrieved 23 June 2015 from *http://csrc.nist.gov/publications/nistpubs/800-82/SP800-82-final.pdf*.

4. Brendan Galloway and Gerhard P. Hancke. Introduction to Industrial Control Networks. *IEEE Communications Surveys & Tutorials*, 15(2):860–880, 2013.

5. Nicolas Falliere. Exploring Stuxnet's PLC infection process. Technical report, Symantec Official Blog, 2010. Retrieved 23 June 2015 from *http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process*.

6. Marshall Abrams and Joe Weiss. Malicious Control System Cyber Security Attack Case Study Maroochy Water Services, Australia. *MITRE Corp USA https://www.mitre.org/publications/technical-papers*, 253(August):73–82, 2008.

7. Yu-Lun Huang, Alvaro A. Cárdenas, Saurabh Amin, Zong-Syun Lin, Hsin-Yi Tsai, and Shankar Sastry. Understanding the physical and economic consequences of attacks on control systems. *International Journal of Critical Infrastructure Protection*, 2(3):73–83, 2009.

8. David P. Duggan, Michael Berg, John Dillinger, and Jason Stamp. Penetration testing of Industrial Control Systems. Technical Report 2005-2846P, Sandia National Laboratories, March 2005. Retrieved 26 October 2014 from *http://energy.sandia.gov/wp/wp-content/gallery/uploads/sand_2005_2846p.pdf*.

9. Miles A. McQueen and Wayne F. Boyer. Deception used for cyber defense of control systems. In *Human System Interactions, 2nd Conference on*, HSI'09, pages 624–631. IEEE, May 2009.

10. James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, Boston, MA, USA, 6th edition, 2013.

11. Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, 1999. Retrieved 23 June 2015 from *http://tools.ietf.org/html/rfc2616*.

12. Siemens. Step 7 professional manual, 2013. Retrieved 16 August, 2016 from *https://cache.industry.siemens.com/dl/files/956/67851956/att_61613/v1/readmestep7professional.pdf*.

13. Paul Brooks. Ethernet/IP - industrial protocol. In *Emerging Technologies and Factory Automation, Proceedings of the 8th IEEE International Conference on*, volume 2, pages 505–514. IEEE, October 2001.

14. Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004.

15. Jungsuk Song, Hiroki Takakura, Yasuo Okabe, Masashi Eto, Daisuke Inoue, and Koji Nakao. Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation. In *Building Analysis Datasets and Gathering Experience Returns for Security, Proceedings of the First Workshop on*, BADGERS '11, pages 29–36, New York, NY, USA, 2011. ACM.

16. Honeywall, 2009. Retrieved 22 June 2015 from *https://projects.honeynet.org/honeywall*.

17. Argos. Argos introduction, 2008. Retrieved 28 June 2016 from *http://www.few.vu.nl/argos/*.

18. Paul Baecher and Markus Koetter. Dionaea, 2013. Retrieved 23 June 2015 from *http://dionaea.carnivore.it*.

19. Dieter Joho. Active honeypots. Master's thesis, Department of Information Technology University of Zurich, Switzerland, Zurich, Switzerland, December 2004. Retrieved 30 October 2016 from *https://pdfs.semanticscholar.org/4771/1f4178c17f2943801bd230e655efd98e1c52.pdf*.

20. Johnny Vestergaard. Beeswarm, 2015. Retrieved 23 August 2015 from *http://www.beeswarm-ids.org*.

21. Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *SIGOPS Operating Systems Review*, 39(5):148–162, October 2005.

22. Xuxian Jiang, Dongyan Xu, and Yi-Min Wang. Collapsar: A VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *Journal of Parallel and Distributed Computing*, 66(9):1165–1180, 2006.

23. Robin G. Berthier. *Advanced Honeypot Architecture for Network Threats Quantification*. PhD thesis, University of Maryland, College Park, MD, USA, 2009. Retrieved 24 August 2015 from *http://drum.lib.umd.edu/bitstream/1903/9204/1/Berthier_umd_0117E_10310.pdf*.

24. Niels Provos. Honeyd (version 1.6d), 2013. Retrieved 23 August 2015 from *https://github.com/DataSoft/Honeyd*.

25. Michael M. Winn. Constructing cost-effective and targetable ICS honeypots suited for production networks. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2015. *oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA615223*.

26. Vishal Chowdhary, Alok Tongaonkar, and Tzi-cker Chiueh. Towards automatic learning of valid services for honeypots. In *Distributed Computing and Internet Technology, Proceedings of the First International Conference on*, ICDCIT'04, pages 469–469, Berlin, Heidelberg, 2004. Springer-Verlag. Retrieved 31 August 2015 from *http://seclab.cs.sunysb.edu/alok/papers/icdcit04.pdf*.

27. Xuxian Jiang and Dongyan Xu. BAIT-TRAP: a catering honeypot framework. Technical report, Purdue University, 2004. Retrieved 23 August 2015 from *http://friends.cs.purdue.edu/pubs/BaitTrap.pdf*.

28. M Zubair Rafique, Juan Caballero, Christophe Huygens, and Wouter Joosen. Network dialog minimization and network dialog diffing: two novel primitives for network security applications. In *Computer Security Applications Conference, Proceedings of the 30th Annual*, pages 166–175. ACM, 2014.

29. Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Network and Distributed System Security Symposium*, February 2006. Retrieved 1 September 2015 from *http://www.internetsociety.org/doc/protocol-independent-adaptive-replay-application-dialog*.

30. Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: an automated script generation tool for Honeyd. In *Computer Security Applications Conference, 21st Annual*, pages 202–214. IEEE, December 2005.

31. Corrado Leita, Marc Dacier, and Frederic Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In *Recent Advances in Intrusion Detection, Proceedings of the 9th International Conference on*, RAID'06, pages 185–205, Berlin, Heidelberg, 2006. Springer-Verlag.

32. Phillip C. Warner. Automatic configuration of programmable logic controller emulators. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2015. *oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA620212*.

33. Weidong Cui, Vern Paxson, and Nicholas Weaver. GQ: Realizing a system to catch worms in a quarter million places. Technical Report 06-004, International Computer Science Institute, September 2006. Retrieved 31 August 2015 from *http://www.icir.org/vern/papers/gq-techreport.pdf*.

34. Christian Kreibich, Nicholas Weaver, Chris Kanich, Weidong Cui, and Vern Paxson. GQ: Practical containment for measuring modern malware systems. In *Internet Measurement Conference, Proceedings of the 2011 ACM SIGCOMM Conference on*, IMC '11, pages 397–412, New York, NY, USA, 2011. ACM.

35. Corrado Leita and Marc Dacier. SGNET: A worldwide deployable framework to support the analysis of malware threat models. In *Dependable Computing Conference, Seventh European*, pages 99–109. IEEE, May 2008.

36. R: The R Project for statistical computing, 2015. Retrieved 23 June 2015 from *http://www.r-project.org*.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2017	Master's Thesis	Sept 2015 — Mar 2017

4. TITLE AND SUBTITLE

Synchronization Algorithms for
Programmable Logic Controller Emulation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

McCargar, Elwyn J., 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-17-M-050

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

14. ABSTRACT

This research develops a way to discover the optimal Synchronization Algorithm used to synchronize a back-end Programmable Logic Controller (PLC) with a honeypot based on the emulated protocol. Synchronization between the back-end PLC and the honeypot is important for the emulation of a device's protocol because some protocols are stateful and provide responses to requests based on previous requests. The honeypot needs to ensure the back-end PLC is in the same state before sending a request to the PLC. There are four Synchronization Algorithms under test: Catchup, Lockstep, Latelock, and Triggerlock. These four algorithms are each tested against eleven protocols. Through a full factorial experiment, it is shown for 91% of the protocols the best algorithm is Triggerlock. The only exception is the stateless protocol HTTP, where the best algorithm is Catchup because of the honeypot's instant disconnection from the back-end PLC after receiving the response to its request.

15. SUBJECT TERMS

SCADA, honeypot, programmable logic controller, industrial control systems, automation, emulator, protocol reverse engineering, synchronization algorithms

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 (ENG)
U	U	U	U	108	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x7979 Barry.Mullins@afit.edu