DYNAMIC HONEYPOT CONFIGURATION FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

THESIS

Kyle A. Girtz

AFIT-ENG-MS-16-M-253

AFIT-ENG-MS-16-M-253

DYNAMIC HONEYPOT CONFIGURATION 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 Cyber Operations

Kyle A. Girtz, B.S.E.E

March 2016

AFIT-ENG-MS-16-M-253

DYNAMIC HONEYPOT CONFIGURATION FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

THESIS

Kyle A. Girtz, B.S.E.E

AFIT-ENG-MS-16-M-253

Abstract

Attacks on industrial control systems and critical infrastructure are on the rise. Important systems and devices like programmable logic controllers are at risk due to outdated technology and ad hoc security measures. To mitigate the threat, honeypots are deployed to gather data on malicious intrusions and exploitation techniques.

While virtual honeypots mitigate the unreasonable cost of hardware-replicated honeypots, these systems often suffer from a lack of authenticity due to proprietary hardware and network protocols. In addition, virtual honeynets utilizing a proxy to a live device suffer from performance bottlenecks and limited scalability.

This research develops an enhanced, application layer emulator capable of alleviating honeynet scalability and honeypot inauthenticity limitations. The proposed emulator combines protocol-agnostic replay with dynamic updating via a proxy. The result is a software tool which can be readily integrated into existing honeypot frameworks for improved performance.

The proposed emulator is evaluated on traffic reduction on the back-end proxy device, application layer task accuracy, and byte-level traffic accuracy. Experiments show the emulator is able to successfully reduce the load on the proxy device by up to 98% for some protocols. The emulator also provides equal or greater accuracy over a design which does not use a proxy. At the byte level, traffic variation is statistically equivalent while task success rates increase by 14% to 90% depending on the protocol. Finally, of the proposed proxy synchronization algorithms, templock and its minimal variant are found to provide the best overall performance.

AFIT-ENG-MS-16-M-253

To my family.

I would not be who I am without you.

Stay off the paved roads.

Acknowledgements

I would like to thank Capt Phillip Warner for laying a solid foundation in his research. His previous work and assistance throughout this process were invaluable. Thank you for your dedication to a job well done.

I would also like to thank Dr. Barry Mullins, my advisor, for giving me the opportunity to pursue a graduate degree and for his guidance throughout my career at AFIT.

Finally, I would like to thank Capt Joseph Hall and Capt Michael Todd for their assistance and peer reviews.

Kyle A. Girtz

Table of Contents

Page

Page

List of Figures

List of Tables

List of Acronyms

CIP	Common Industrial Protocol
CUT	component under test
DoS	Denial of Service
ENIP	EtherNet Industrial Protocol
EtherNet/IP	EtherNet Industrial Protocol
FTP	File Transfer Protocol
GUI	graphical user interface
HI	high-interaction
HMI	human-machine interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
HVAC	Heating, Ventilation, and Air Conditioning
ICS	Industrial Control Systems
IDS	Intrusion Detection System
IED	Intelligent Electronic Device
IP	Internet Protocol
IPv4	Internet Protocol version 4

IPv6	Internet Protocol version 6
ISO-TSAP	ISO Transport Service Access Point
IT	Information Technology
LI	low-interaction
MAC	Media Access Control
MTU	Master Terminal Unit
OS	Operating System
pcap	packet capture
p-tree	protocol tree
PLC	programmable logic controller
RTU	Remote Terminal Unit
SCADA	Supervisory Control and Data Acquisition
SNMP	Simple Network Management Protocol
SUT	system under test
TCP	Transmission Control Protocol
VM	virtual machine
XML	Extensible Markup Language

DYNAMIC HONEYPOT CONFIGURATION FOR PROGRAMMABLE LOGIC CONTROLLER EMULATION

I. Introduction

1.1 Background

Technological advancement on a societal scale requires stable underlying infrastructure to generate and distribute electricity, gas, water, communications, commercial goods, and other necessities. In the United States and similar modern societies, critical infrastructure is automated and controlled by computer networks known as Industrial Control Systems (ICS). Historically, ICS networks were isolated and designed for robustness rather than security [1]. Today, security requirements match those of availability as ICS networks have become increasingly interconnected and exposed to the Internet [2]. The programmable logic controller (PLC), a common ICS device, is particularly important to the security of ICS networks [3]. These devices contain custom programming to act as data collection and actuator control points. Modern malware, such as Stuxnet, has successfully compromised PLCs with physically destructive results [2, 4]. To complicate matters, the need for uninterrupted service makes it very difficult to update or patch ICS systems with the same frequency of traditional Information Technology (IT) systems [5]. For this reason, security is shifting to the use of sensors within the system tasked with maintaining an acceptable device state [6, 7].

One deception-based technology commonly employed for network state detection and threat analysis is the *honeypot* [8]. A honeypot is a bait device added to a network to attract attackers and collect suspicious traffic [9]. The quality of a honeypot framework depends on the level of authenticity and flexibility provided. A low-interaction (LI) honeypot provides low authenticity by only emulating specified services. A high-interaction (HI) honeypot provides very high authenticity at the cost of flexibility by replicating an entire computer system, either on physical hardware or through a virtual machine (VM). Deceptive security schemes, and honeypots in particular, have been shown to effectively identify new attacks and their origins [10, 11].

1.2 Motivation

Several scalability difficulties accompany the application of honeypot technology to PLCs. PLCs can be very expensive, rendering hardware replication and basic HI honeypots impractical. In addition, many ICS devices use proprietary networking protocols and hardware. Without *a priori* knowledge of a protocol, emulation cannot be easily scripted for a LI honeypot. Reverse engineering is possible but will typically yield a custom, inflexible solution. This time-consuming process would have to be repeated for every new device or protocol.

In spite of the difficulties, there are effective techniques for creating ICS honeypots. Frameworks such as Honeyd allow LI honeypots to be configured to respond to the network layer of the traditional protocol stack [12]. This method has also been automated to create dynamic Honeyd configurations based on the state of a network [13, 14, 15, 16]. While some of these solutions include application layer configuration, this is not always possible in the presence of proprietary protocols. Automated protocol reverse-engineering can help to emulate unexpected and unknown protocols [17]. ScriptGen automatically creates Honeyd configurations [18] while the extended ScriptGenE can emulate the protocol directly by replaying captured conversations in context [19]. A final technique is to proxy traffic to a single physical device. Scalability is still questionable as the back-end device can be easily overwhelmed by large amounts of traffic.

Hybrid honeypots consists of some combination of the above methods with the goal of achieving both authenticity and flexibility in one system [8]. Typically this involves the simultaneous use of HI and LI honeypots. A simple example is Honeyd+, an extension of Honeyd with a PLC proxy [20]. The front-end Honeyd instance handles simple network traffic while application layer requests are sent to a real PLC. More complex hybrid systems like SGNET [21] and GQ [22] have been tested on IT networks, but neither framework is open-source. In addition, there is no guarantee that IT honeypots will be effective or scalable on ICS networks for the reasons discussed above.

There continues to be a need for an open-source, hybrid honeypot framework designed and optimized for ICS networks. This research develops a hybrid system by integrating a honeypot proxy with automatic protocol reverse engineering and dynamic replay.

1.3 Research Goals

The goal of this research is to develop and test a hybrid honeypot configuration with protocol emulation via dynamic replay and incremental updating. The developed framework is an extension of the ScriptGenE software suite [19]. Initial emulation based on training captures should accurately reproduce, in context, some of the traffic generated by the PLC. Untrained traffic is forwarded to a real device via proxy in order to improve replay capabilities. The emulator should not proxy all requests, just unrecognized requests. The updated emulator should accurately respond to unrecognized requests with accurate responses rather than a default error message.

1.4 Approach

An automatic protocol emulation tool is developed from the base components of the ScriptGenE framework [19]. This tool consists of server replay based on previously captured training data with the addition of a novel *dynamic update* function. The developed tool is intended to improve the configuration of Honeyd+ [20] to provide a flexible and robust hybrid honeypot framework for ICS.

1.4.1 Protocol emulation.

The automatic emulation element of the developed framework extends the replay feature of the ScriptGenE framework. ScriptGenE, itself an extension of the Honeyd script-creation tool ScriptGen [18, 23], is a suite of tools capable of building a generalized protocol tree (p-tree) from a set of network traces and replaying the protocol through contextual emulation [19]. This research focuses on the replay capability. Using previously collected and processed p-tree data, observed pieces of the protocol can be replayed. In the event that unexpected traffic arrives, there are several response strategies. ScriptGenE can backtrack to a point higher in the tree or send default error messages. The new dynamic update function allows ScriptGenE to proxy the new request to a real PLC, observe and forward the response back to the client, and update its protocol tree to handle future requests of this type. This function is intended to replace the default error messages when backtracking does not occur.

1.4.2 Honeynet configuration.

The developed tool is designed to be added to a complete Honeyd+ configuration. Honeyd+ is an extension of Honeyd that provides search-and-replace functionality to proxy replies from a back-end PLC [20]. The Honeyd+ framework is designed to be a production-level, PLC honeypot deployed on the Linux Raspberry Pi platform. A key weakness to this framework is poor performance by the back-end PLC in the presence of heavy traffic. The developed emulation tool reduces this load by emulating the back-end PLC as an intermediary. In this configuration all Honeyd+ proxies point to the emulation device, which then proxies directly to the PLC for dynamic updates.

While the tool is designed to supplement the Honeyd+ framework, it can be deployed independently as a self-contained honeypot or as a component of other honeynet frameworks. Care must be taken when using the tool independently as it provides limited honeypot functionality by exclusively emulating the application layer.

1.4.3 Experimentation.

All experimental network traffic in this research is generated by two PLCs. One device is an Allen-Bradley ControlLogix L61 PLC with an attached EtherNet Industrial Protocol (EtherNet/IP) ENBT module. This device is interrogated using the RSLinx software by Rockwell Automation and `wget` to query the module information and webserver respectively. The other device, a Siemens S7-300 PLC with discrete and analog input/output modules, is interrogated using Siemens' SIMATIC STEP7 software for module browsing.

A single experiment is developed to test both devices and all three primary protocols over a series of randomly ordered tasks. Hypertext Transfer Protocol (HTTP) and EtherNet Industrial Protocol (ENIP) are tested on the L61 while ISO Transport

Service Access Point (ISO-TSAP) is tested on the S7-300. The experiments leverage a complete initial protocol tree capable of full task replay for each protocol. These protocol trees are randomly corrupted and loaded into an emulator. The relevant tasks are performed on the emulator to determine how well it can compensate for the gaps in the tree with its proxy function. The tasks are repeated for multiple corrupted trees and proxy synchronization algorithms. All experimental tasks, identified by the tuple (protocol, algorithm, tree), are performed in a random order.

During each experimental task run, packet captures are collected for the client and proxy connections separately. These captures are inspected to determine PLC load and emulator authenticity. PLC load is measured by calculating the fraction of client requests that are forwarded to the PLC. Authenticity is measured by comparing the byte differences in the client conversation with a reference trace for each task.

In addition to trace comparisons, each task is determined to pass or fail based on the overall results of the conversation. The visual presence of modules is interpreted as a pass (success) for ENIP and ISO-TSAP while the correct number and size of files downloaded by `wget` indicates a pass (success) for HTTP. Control of these queries and pass/fail determination is performed by OpenCV image recognition and the graphical user interface (GUI) automation software SikuliX [24].

1.5 Assumptions and Limitations

Many of the underlying limitations of the ScriptGenE framework exist as a result of the reliance on previously collected network traces [19]. This research develops a proof of concept extension to ScriptGenE which aims to remove trace-based limitations through the use of a proxy-and-update mechanism. However, some of the other limitations of ScriptGenE persist and are left to be resolved as future work.

1.5.1 Limitations of network trace-based approaches.

Trace-based approaches are only capable of emulating observed traffic. This is particularly important for field variations within similar messages. Having never seen a value vary, the emulator will fail to recognize its ability to vary. This problem is not solved by a proxy update. A single proxy response has no baseline with which to be compared, so no field variation can be detected. This research assumes that training traces may be missing packets from a connection. The missing packets may consist of any traffic except for the Transmission Control Protocol (TCP) connection handshakes. Updates are made when entirely new traffic is encountered. This could be a new conversation context or a significant deviation from a known conversation.

An additional limitation of trace-based approaches involves emulating encrypted traffic. If training traffic contains no dynamic fields the traffic may be replayed as is. When updating through a proxy, encryption issues can potentially be solved by establishing a new encrypted session with the back-end device. This allows the emulator to observe dynamic fields during decryption prior to forwarding. However, if encrypted training data contains dynamic fields and emulation is attempted independent of a proxy connection, the emulation will be incorrect.

1.5.2 Network protocols involved.

This research provides a proof of concept proxy-and-update extension to the trace-based training and replay functionality of ScriptGenE. As ScriptGenE has only been tested on HTTP, ENIP, and ISO-TSAP, testing of this extension is also limited to these protocols. Additionally, the extended ScriptGenE framework operates exclusively at the application level. While many extensions and applications are possible, this research intends to add the emulator to an existing Honeyd+ network which handles all non-application layer traffic [20]. Finally, the original ScriptGenE framework is limited to supporting TCP and Internet Protocol version 4 (IPv4) traffic. This remains true for this research.

1.5.3 Limited set of tasks.

This research tests three protocols on two PLCs. Each protocol is tested with a single task. In reality, there is a wide variety of potential tasks and conversations, some very complex. In addition, production environments produce aggregated traffic with specific characteristics. This proof of concept research does not attempt to simulate complex conversations or environments. The test tasks are generated in a controlled environment in order to create an effective experiment for examining the extended functionality of ScriptGenE.

1.5.4 Limited configuration setup.

A single hardware and firmware configuration is used for each PLC during experimental testing. Manufacturers such as Allen-Bradley and Siemens produce a variety of PLC models and modules resulting in a wide range of hardware configurations. In addition, there are numerous firmware versions for each product. Any of these factors may affect traffic generation. For this reason, pre-trained emulation is assumed to be valid for the exact configuration used to generate the training data. Proxy responses are assumed to be valid only for the exact configuration found in the back-end target. It is possible that poor emulator configuration could result in training data and proxy data originating from different PLC configurations. Ensuring this does not happen or assuming appropriate risk is the responsibility of the system administrator. In most practical situations, extensive training data and relatively similar PLC configurations mitigate this risk.

1.5.5 Timing.

When a client connects to the ScriptGenE emulator some of its traffic may be proxied. In this configuration there are many relevant timing considerations. This research does not address timing issues such as discrepancies between those emulated responses that required proxying and those that did not. It is also possible that emulation of training data (no proxy required) may not match the timing with which the captured conversation occurred. These timing discrepancies will not affect the content of the conversation and are also ignored in this research.

1.6 Thesis Overview

Chapter II contains an overview of ICS technology and related work on honeypots and their applications. Chapter III provides a description of the developed emulator configuration. Chapter IV details the experimental design with results in Chapter V. Chapter VI presents research conclusions and suggestions for future work.

II. Background and Related Research

2.1 Overview

This chapter provides a brief background on the characteristics and importance of Industrial Control Systems. In particular, a discussion of ICS security reveals that traditional Information Technology security solutions are insufficient to properly address complex, decentralized networks of relatively primitive ICS systems. Honeypots are presented as a critical security measure to compensate for this inherent insecurity. Relevant application layer networking protocols and honeypot basics provide a foundation for the final survey of modern honeypot technology.

2.2 Background

2.2.1 Industrial Control Systems.

In order to flourish, every society depends on policy, infrastructure, and the complex interactions between economic and political systems. In the United States, *critical infrastructure*, as defined by Presidential Policy Directive 21, currently consist of sixteen sectors including the chemical, energy, food, water, and nuclear industries as well as services such as health care, finance, transportation, communications, and information technology [25]. Many of these sectors rely on ICS to perform Supervisory Control and Data Acquisition (SCADA) and other functions. ICS is a blanket term for all digital and analog computer systems providing autonomous sensing and control of physical processes that is reliable, safe, and efficient [1].

Figure 1 shows an abstracted ICS system and its three primary components: the human-machine interface (HMI), control loop, and remote diagnostics and maintenance [1]. The HMI and remote maintenance components provide interfaces for operators (locally or remotely) to observe and modify the control loop state and functionality. The control loop itself achieves autonomy via a feedback loop consisting of sensor and actuator subcomponents. The sensors provide physical data to the controller so that it may decide how best to command the actuators. A simple example is temperature control within a building. A thermometer (the sensor) provides temperature data to the controller which triggers either the heater or air conditioning (the actuators). The desired temperature can be set via an HMI like the thermostats commonly found in homes. The HMI is a convenient abstraction of the control loop; a homeowner does not have to manually turn the heater on and off to maintain the desired temperature.

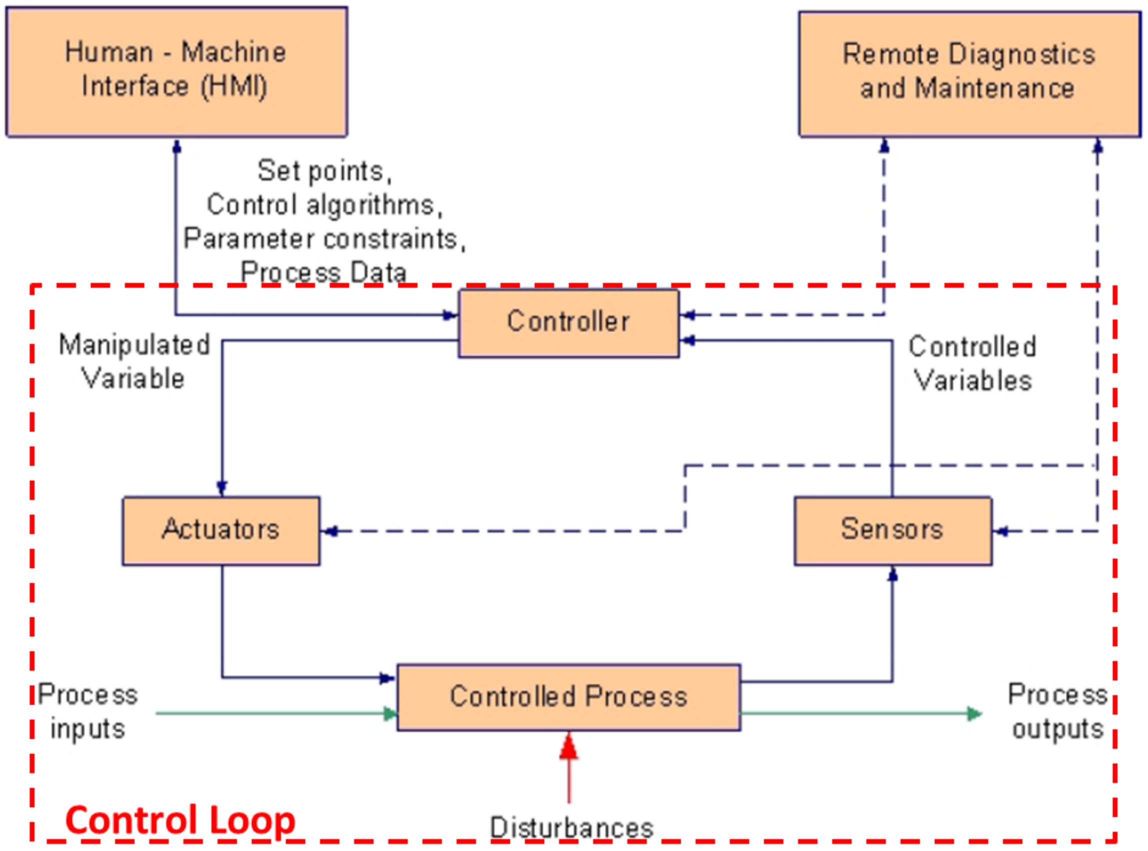


Figure 1. ICS block diagram [1]

A more realistic ICS implementation would look something like the system shown in Figure 2 [1]. These systems are often composed of multiple, interconnected networks possibly spread over large geographic areas. For this reason, the control center consolidates all observation and command capabilities for the convenience of the operators. Control centers typically contain the primary HMI and a Master Terminal Unit (MTU). The MTU communicates with scattered Remote Terminal Units (RTUs) in a master/slave type relationship. The RTU collects data from sensors (not shown). Other ICS devices include the Intelligent Electronic Device (IED) and the PLC. An IED performs basic data processing locally before reporting to the control center. A PLC is a flexible device that may act as an RTU or provide more complex control functionality.

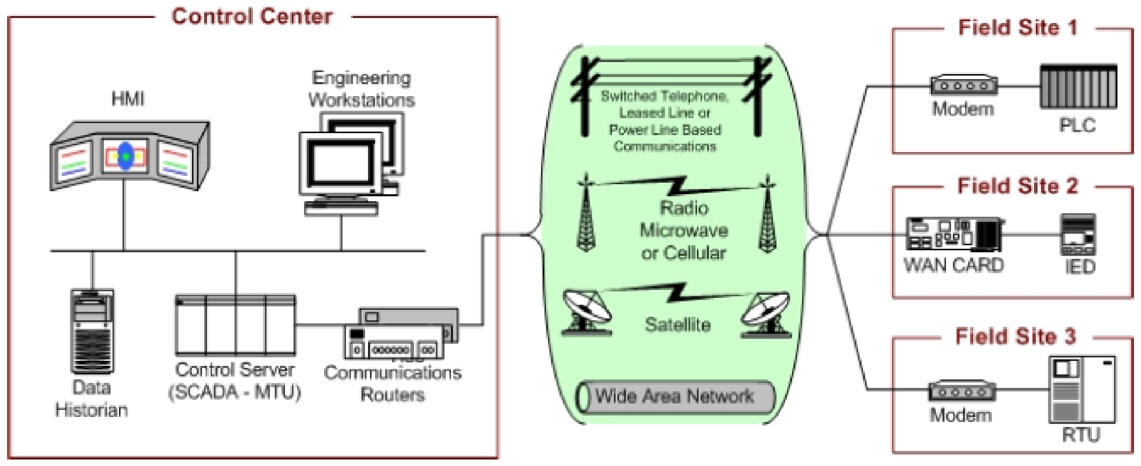


Figure 2. General SCADA layout [1]

As noted, the PLC is a unique device, being remotely deployed and capable of both data collection and programmable control through ladder logic applications [1]. PLC hardware and firmware is usually vendor specific and modular. The modules allow for flexible configuration of the hardware by providing extra analog or digital input/output or communications functionality. Most PLCs will have at least one controller module where ladder logic applications are deployed.

2.2.2 ICS security.

Traditional ICS networks are dedicated to data collection and control functions. No external connections were available; practical security was accomplished through lack of access [1]. However, as the Internet grew into its present-day, ubiquitous state, ICS networks were increasingly exposed to the outside world in order to facilitate efficient productivity and remote management [1]. This trend is expected to continue as the dependence of society on increasingly complex control systems rises [1, 2]. Shodan, a database and search engine for Internet-facing embedded devices, provides ready access to many Internet-facing PLCs, SCADA devices and servers, and other building controls such as Heating, Ventilation, and Air Conditioning (HVAC) systems. Researchers have used Shodan to find, catalog, and create visualizations of thousands of these devices [26, 27]. Further work has shown that an exposed PLC will respond to unauthenticated queries with codes that may reveal the function and industry sector of the device [28]. These results are troubling; if researchers can identify vulnerable devices, there is no reason to doubt malicious actors are capable of attacking them.

Indeed, ICS attacks are becoming more common and dangerous [1, 2]. The well-known worm Stuxnet exploited PLCs in 2010 as part of what appears to be a targeted attack on Iranian nuclear enrichment facilities [4]. The observable effects of the attack included physical destruction of centrifuge equipment. Similarly, ICS threat model studies have shown that application of Denial of Service (DoS) and integrity attacks against a PLC in chemical production plants may lead to unsafe states such as undesired shutdowns or explosions [2].

Another troubling reality of ICS systems is that destructive physical effects can be achieved without actually exploiting the target device. With little or no remote authentication, legitimate commands sent at the wrong time or in the wrong way can cause a system to physically damage its environment. This inherent insecurity was powerfully demonstrated at Digital Bond's Project Basecamp [3]. The target, an Allen-Bradley ControlLogix PLC, was provided to a simulated attacker. The attacker was able to cause a DoS, trojanize the firmware, and leak arbitrary information with *legitimate* traffic designed for ease of use. These intrusions could potentially modify the behavior of equipment or entirely shut it down had this simulation been a live attack.

A natural question to ask at this point is "How often are PLCs attacked due to resources like Shodan?" The answer is unclear at the moment as recent experiments provide conflicting and unsatisfactory results. In 2013, Kyle Wilhoit performed two studies in which he seeded Google and Shodan with ICS honeypot information and identified all targeted attacks [29, 30]. The studies, one lasting 28 days and the second lasting four months, resulted in 12 and 74 attacks respectively. However, there is some question as to whether these attacks actually targeted the PLCs or only the HMIs [28]. More questions arise when these experiments are compared to a study by Bodenheim in which four PLCs were deployed with Internet-facing Internet Protocol (IP) addresses [31]. The PLC configurations varied so that some were easier to identify by Shodan than the others. After 55 days of exposure, the absence of any direct attacks caused Bodenheim to conclude that Shodan did not significantly impact the targeting of PLCs on the Internet.

Regardless of the targeting mechanism, the key to securing modern ICS networks lies in tightened defenses rather than obscurity. How can PLCs be kept from exploitation once exposed? The difficulty lies in the dedicated, trust-based protocols ICSs used before merging with the Internet. ICS design was not intended for a massively interconnected environment hosting malicious actors. Fundamentally, ICS and IT networks have different historical development paths and design goals [1].

There are a couple of key differences that make securing ICS more difficult than IT systems. One factor is age. IT equipment has an average lifespan of 3-5 years due to rapid technology development. ICS equipment often remains in use for 15-20 years or longer due to the specificity of its design [1]. This difference in design creates problems with applying modern IT security solutions to aging ICS equipment. For example, in one study a ping sweep, a technique commonly used to map IT networks, caused ICS equipment to malfunction in spectacular ways including randomly moving robotic arms, destroying circuit boards in production, and disrupting gas availability to customers of a local utility company [5].

A second difficulty involves the difference between security goals and safety goals. IT systems emphasize integrity and confidentiality in most settings while ICS networks value availability and safety. Where data matters most, businesses want to ensure the data is valid and private. When a network controls physical processes, safety overrides privacy. Additionally, availability is of utmost importance for most critical infrastructure ICS systems [1]. The lights must stay on and water must keep flowing without interruptions. This makes it undesirable to interrupt ICS for temporary patching or replacement; even a reboot may not be feasible. In summary, IT and ICS networks cannot be patched with new security measures in the same way [1].

A final difficulty is cost. While most businesses use IT networks to facilitate their service or production goals, in critical infrastructure an ICS system is the service being provided. The flexibility of IT infrastructure allows IT security solutions to be more flexible while industrial companies may have fewer options for securing legacy ICS equipment. Because nearly 90% of all critical infrastructure businesses using these control systems are privately owned, finances often overshadow security [1]. In the minds of many decision makers, the guaranteed age and availability costs of implementing security measures far outweigh the offhand risk of a potential attack.

Provided the difficulty of securing ICS, traditional defense mechanisms have proven rather ineffective. A common defense paradigm is the separation of corporate IT networks from ICS networks using a demilitarized zone as shown in Figure 3 [1, 5]. Critical control systems are cordoned off by firewalls in order to prevent potentially harmful traffic from entering the ICS space. Unfortunately, as time has shown, this defense fails to address the real problem: ICS systems are designed to trust inputs. Early control networks relied on local access and all commands were considered trustworthy. Today, there must be a paradigm shift toward "resilient control systems" [32].

Resiliency in ICS requires more than just a proper network configuration. The state of the system must be monitored at all times in conjunction with mechanisms to maintain a safe state close to operational normalcy even in the event of malicious or unexpected disturbances [32]. This approach is discussed by Carcano et al. as an extension of an Intrusion Detection System (IDS) for industrial networks [7]. The idea is to collect data of normal operations and so called *critical states* before creating metrics to accurately distinguish between the two. During operation, the IDS can then monitor the state of the system rather than simply inspect traffic for signatures. This method allows the IDS to determine when a system is approaching a critical state and take protective measures before damage is done.

A variety of other creative defenses have been proposed as well. McQueen and Boyer propose using a collection of deceptive methods simultaneously to protect data and confuse attackers [10]. IDS based systems and honeypots have been found to improve traditional security paradigms significantly [6, 11]. Honeypots in particular provide powerful possibilities for state awareness. The next section addresses honeypot technology in detail.

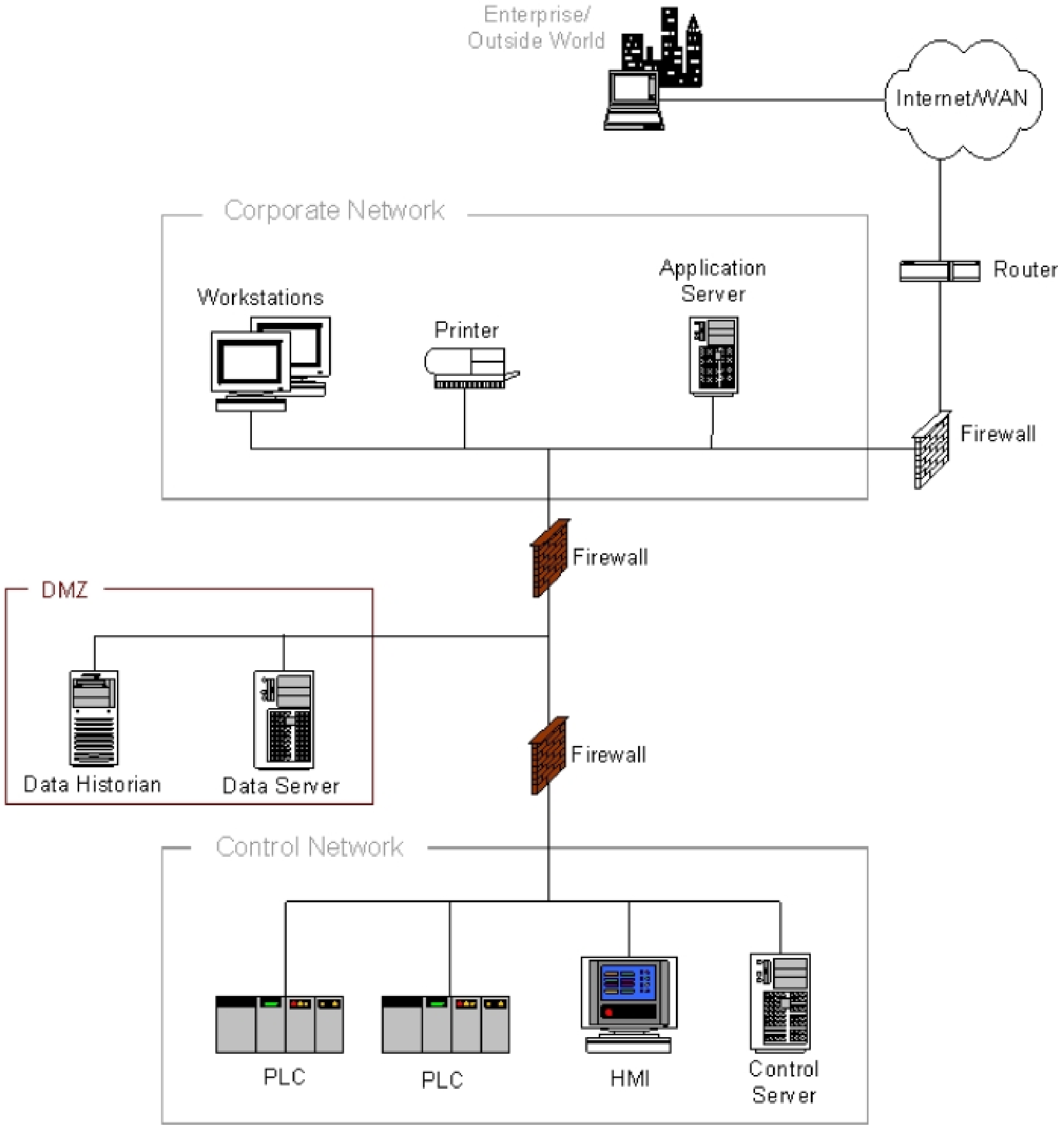


Figure 3. ICS network configuration recommended by NIST [1]

2.2.3 Application layer protocols.

A basic understanding of networking protocols is critical for any honeypot discussion. This section introduces the relevant application layer protocols for this research. The interested reader is referred to [33] for more information on networking protocols in general.

A common stack-based model of Internet protocol layers is shown in Figure 4. Each layer below the application layer provides a network service with the end goal of reliably delivering application layer data from a process on one machine to a specific process on another machine [33]. The application layer contains the most variability as any custom application may define its own application layer protocol. This research specifically addresses impersonating a device by replicating network traffic at the application layer. The application layer is the most difficult and interesting layer to emulate because of its potential variability and the high correlation of packet information to process functionality. The rest of this section introduces the application layer protocols used in this research.

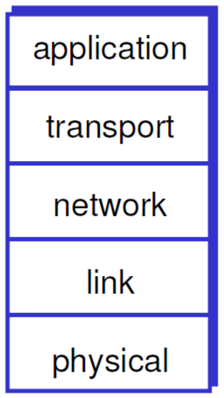


Figure 4. Internet Protocol stack [33]

Arguably the most well-known application layer protocol is HTTP. As defined in RFC 2616, HTTP is a generic, stateless protocol primarily used to transfer hypertext over TCP port 80 [34]. Hypertext consists of a descriptive language such as Hypertext

Markup Language (HTML) transferred by HTTP as a normal text block. There are several types of HTTP requests and responses; the most common are the GET request and OK response used to retrieve a web page.

While HTTP represents a loose request/response behavior ideal for web pages, most PLCs also use a primary control protocol for configuration and status updates. These protocols tend to maintain a context state and are complex in comparison to HTTP. One standard used by many devices and manufacturers is the combination of EtherNet/IP and Common Industrial Protocol (CIP). These protocols were developed to consolidate the diverse command-and-control protocols used in automated ICS networks [35]. The object-oriented design allows for flexibility in additions or subtractions from a network as well as effective communication between different networks. ENIP and CIP operate over TCP port 44818 and are used extensively by Rockwell Automation for Allen-Bradley devices.

A second control protocol, designed and used by Siemens, is ISO-TSAP. Much like ENIP and CIP, ISO-TSAP encapsulates the proprietary Siemens control protocol PROFINET to allow it to operate over Ethernet connections rather than the original specialized buses. ISO-TSAP communicates over TCP port 102.

2.2.4 Honeypots.

A *honeypot* is a system installed on a network which the administrators expect to be probed, attacked, or compromised [9]. Honeypots are useful for identifying the presence and methods of attackers on a network and can be configured for unique applications like worm detection, adversary distraction, spam prevention and many more. Honeypots have several advantages over traditional signature-based IDSs including fewer false positives, the ability to interact with attackers at the application-level, and zero-day attack detection [8]. If the honeypot system is configured to be completely passive, it can be assumed that any traffic it receives will be suspect at best and malicious at worst. Application level interaction allows honeypots to investigate and log traffic even if that traffic is encrypted during transmission. Finally, honeypots can potentially detect a zero-day attack (i.e. exploiting a vulnerability never discovered or patched) because it does not depend on identified signatures but rather on current operational behavior. A new attack can be identified in real time by its effects on the honeypot system.

Because a honeypot is a passive device designed for information gathering, its value depends entirely on the information gathered [9]. Honeypots can be classified into several categories. Physical honeypots are actual hardware machines with their own IP addresses. Virtual honeypots are software-simulated machines configured to behave similarly to the target system to some degree. The degree of interaction of a honeypot can be either high or low depending on how much of the target system the honeypot is able to replicate. The following sections discuss and provide examples of each honeypot type.

All honeypots are evaluated based on three operational characteristics: performance, fidelity, and security [8]. Performance refers to the honeypot's ability to handle heavy traffic or multiple virtual devices simultaneously. Fidelity indicates how close the honeypot appears to mimic the functionality of the real target device. Security describes how vulnerable the system is in the event an attacker obtains access to the honeypot and attempts to exploit it to attack the network further.

As a powerful example of the benefits of honeypot information, a team of researchers conducted a three year experiment gathering Internet traffic with honeypots [11]. The resulting data set, "Kyoto 2006+", reveals the approximate global state of malware on the Internet. Almost half of the collected sessions were attacks. About one percent of these attack sessions (425,719) contained unknown attacks that did not trigger the IDS. This is equivalent to an average of 428 unknown attacks every day while the honeypots were operational. In addition, the data set revealed the most common countries of origin, target ports, and shellcodes used as attack payloads. This information shows the power honeypots can offer security experts a realistic understanding of current attack methodologies and tools.

2.2.4.1 High-interaction honeypots.

A HI honeypot is a conventional computer system that operates normally without emulating any functionality [8]. Through physical hardware or a VM, it provides real services and genuine interaction to an attacker. This allows the honeypot to achieve maximal authenticity by mirroring the target device exactly. Because the system serves no real purpose, any traffic or interaction is suspect. Typically the honeypot will contain monitoring and logging tools for detection and analysis of suspected traffic. The primary drawback to HI honeypots is the lack of flexibility and scalability. Creating a *honeynet* (i.e. network of honeypots) may involve prohibitively large cost or significant maintenance efforts than would otherwise be practical.

Another problem with HI is the risk of compromise. If an attacker hijacks a honeypot, he may use it as a launch pad to attack valuable systems on the same network. For this reason, the honeynet is normally isolated from the real network by a Honeywall [36]. A Honeywall server acts like a complex firewall that can capture, control, and analyze network traffic coming from the honeynet [8].

One particularly interesting feature of HI honeypots is the potential for 0-day exploit detection. Using a technique called *dynamic taint analysis* the system *taints* all incoming data in order to determine where and how it influences the system [8]. Even if the exploit method is unknown, this technique will reveal the nature, mechanism, and results of the attack. The most prolific dynamic taint analysis honeypot package is Argos, a virtual HI honeypot [8].

2.2.4.2 Low-interaction honeypots.

In contrast with HI honeypots, LI honeypots do not provide an entire, functional computer system for attackers to interact with. Instead, they emulate only specific services, network stacks, or other aspects of a real system [8]. The benefit of this limitation is the simplicity and ease of configuration. LI honeypots often require very little maintenance. Because they do not emulate a full machine, LI honeypots are typically not vulnerable to compromise, making them safer to deploy alongside a production network. The obvious downside to less interaction is the potential loss of authenticity. A LI honeypot should provide "just enough" interaction to trick an attacker or automated tool [8].

There is a wide variety of specialized LI honeypot packages. Each package simulates a particular component of a computer system and is intended to detect specific types of attacks. Some of the popular packages are LaBrea, Tiny Honeypot, Nepenthes/dionaea, Google Hack Honeypot, PHP.HoP, and Honeyd [9]. LaBrea and Tiny Honeypot are general networking honeypots which provide relatively primitive TCP interactions. The Google Hack Honeypot and PHP.HoP are both web service-based honeypots. Nepenthes and its successor dionaea [37] allow a honeynet to collect and analyze malware based on honeypot traffic [38].

One of the most popular LI honeypot frameworks is Honeyd [12]. Honeyd is not actually a single honeypot on its own; it is a framework for creating virtual networks of virtual honeypots [9]. Honeyd allows the user to create arbitrarily many virtual LI honeypot devices and virtually network them together to consume unused IP space on a real network. A more detailed discussion of Honeyd is provided in Section 2.2.4.5.

2.2.4.3 Active honeypots.

The traditional concept of a honeypot consists of a passive machine, physical or virtual, waiting to be contacted by malicious traffic. All of the above examples fit this description. However, given the recent rise in client-side exploits on programs such as browsers, there is a need for client honeypots. These honeypots are often active in the sense that they initiate interactions to simulate the end-user [8]. When traffic is received in response to the honeypots probing, normal honeypot analysis through logging, monitoring, or dynamic taint analysis can be used to determine its effects.

Another potential feature of active honeypot technology is the addition of “bait” to a passive honeynet. One tool based on this idea is called Beeswarm [39]. This system is configured like a normal honeynet with additional “drone” machines which periodically communicate with honeypots and leak credentials as bait for an attacker. When the credentials are used, the attacker may be discovered.

2.2.4.4 Hybrid honeypots.

As discussed above, HI honeypots provide a high level of fidelity at the cost of scalability where LI honeypots provide excellent flexibility and scalability at the (potential) cost of fidelity [8]. Hybrid honeypot systems attempt to merge the two into a cohesive system where the strengths of each type counter the weaknesses of the other. These hybrid systems often consist of LI front ends capable of transferring filtered traffic to HI honeypots.

One solution following this model is Collapsar [40]. Collapsar allows a single honeypot framework to span multiple networks through the use of traffic *redirectors*. These devices redirect traffic from unused IP space on the separate networks back to a front-end gateway. This gateway filters traffic into and out of the Collapsar core where HI honeypots are running on VMs. This solution achieves promising performance and scalability with the primary drawback of potentially increased latencies tipping off an attacker.

Another fascinating example of extreme scalability is Potemkin [41]. In order to determine the nature of attack traffic, Potemkin attempts to virtualize the entire Internet. This is accomplished through a concept known as the *honeyfarm*. A gateway router performs most of the analysis and filtering work while a group of physical servers provide a collection of HI honeypots interaction. Potemkin's key feature is resource management; it only creates a lightweight, cloned VM on an IP address when the gateway receives traffic intended for that address. These temporary VMs are terminated after the traffic is processed. Studies have shown a handful of servers can run over 64,000 honeypots using Potemkin [41]. In order to maintain high virtualization and prevent security issues, the gateway also starts up new VMs for malicious outbound traffic which is reflected back into the honeyfarm. In this way, traffic interactions and causality can be studied in a safe, contained virtual environment.

A final hybrid solution is Honeybrid, an open-source project that attempts to directly combine high and low-interaction honeypots with an emphasis on fidelity over scalability [42, 43]. The architecture consists of a front-end gateway that routes traffic to either high or low-interaction honeypots on the back end based on a decision engine. While the flexibility of Honeybrid allows any high and low-interaction honeypots to be plugged into the architecture, the preliminary experiments used Honeyd as a front end and Qemu and Nepenthes as back-end honeypots.

2.2.4.5 Honeyd.

As mentioned above, Honeyd is not a single honeypot; it is a framework for creating virtual networks of honeypots [9, 12]. Honeyd allows the user to create arbitrarily many virtual LI honeypot devices and virtually network them together to consume unused IP space on a real network. In addition, Honeyd allows for great flexibility through service scripts, allowing the virtual honeypots to run any kind of service or protocol the user desires. To increase authenticity, Honeyd will also project operating systems using signatures from the same databases scanning tools reference. For even greater flexibility and authenticity, Honeyd allows the user to install subsystems [8]. These are external applications that run as a component of the honeypot.

Conceptually, Honeyd allows a network administrator to fill in the unused portions of network IP space with a virtual web of devices, each of which appears to run a real, chosen operating system and provides real, flexible services and functions. In addition, the topology of this virtual network can be specified and is projected realistically. The overall view an attacker observes seems to be a real network of devices indistinguishable from their physical counterparts.

The primary drawback to Honeyd is the need for custom application and service scripts [8]. These must be created for each new application and, once installed, are static until the configuration is modified. This means that changes in the network require changes to Honeyd. New services on a network may require the creation of a brand new service script. Both of these tasks may be time intensive on a complicated or rapidly changing network. If the honeynet cannot be easily maintained, its use will diminish entirely.

2.3 Related Research

2.3.1 Manually configured ICS honeypots.

Due to the substantial differences between traditional IT hardware and ICS devices such as PLCs, deploying a honeypot on an ICS network can be challenging. There have been several attempts to create an ICS honeypot. The key feature of each project is the manual configuration and lack of flexibility. Each design emulates a particular device or a specific set of protocols.

A few small, open-source projects have been published online. The Conpot project is a LI honeypot capable of emulating a chosen PLC [44]. Service scripts for Modbus and Simple Network Management Protocol (SNMP) have been released to supplement the network stack of the actual device. Similarly, Digital Bond provides a honeynet configuration consisting of two VMs: a honeywall server as a gateway and a simulated PLC with exposed Modbus, File Transfer Protocol (FTP), Telnet, SNMP, and HTTP services [45]. The user does have the option of replacing the simulated PLC with a real device for increased authenticity.

Another design that uses a proxy for improved authenticity is Winn's extension of Honeyd [20]. The resulting framework, dubbed Honeyd+, is designed to be a cheap, production-level honeypot framework. It can be deployed on a Linux Raspberry Pi board and configured to proxy to a physical PLC at a remote location. With the Honeyd platform as a stable foundation, many honeynets can be deployed at various geographical locations while each emulates the same live PLC, which is used for proxy requests at the application layer. An example configuration is shown in Figure 5 where three honeypot sites proxy to a single corporate office with a live PLC.

Honeyd+ also improves upon Honeyd by adding a search-and-replace function to the web pages retrieved by the proxy. This ensures that an attacker cannot identify a honeypot by a discrepancy in the IP or Media Access Control (MAC) address on its web page.

One research endeavor, CryPLH, attempted to create a custom ICS honeypot from an Ubuntu VM [46]. The goal was to create an authentic honeypot that is easy to configure and can be modified to emulate similar PLCs relatively quickly. The design itself consists of a stripped down VM configured to look exactly like a

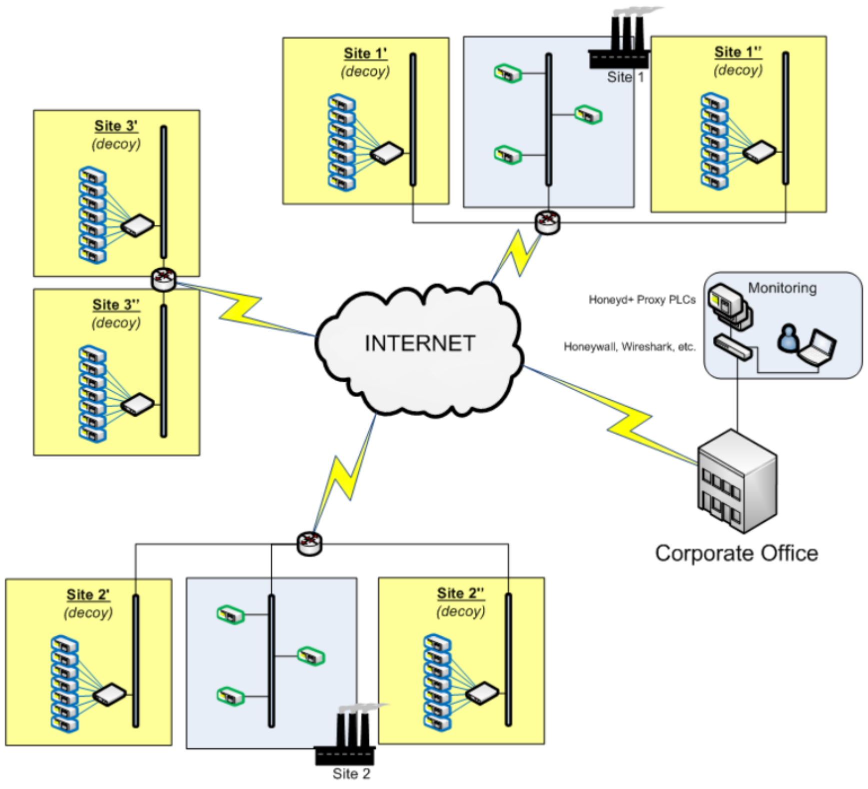


Figure 5. Example Honeyd+ production configuration [20]

Siemens Simatic 300(1) PLC. Using `iptables` as a firewall/filter the VM is able to provide a variety of services including HTTP and HTTP Secure (HTTPS), SNMP, and the ISO-TSAP protocol used by Siemens for their STEP7 programming software. This design requires specific manual configuration for each service provided by the honeypot and as such is very inflexible.

A similar design based on custom configurations of Linux is the highly portable ICS honeypot created by Jaromin on a Gumstix device [47]. Just like CryPLH, this honeypot provides chosen services through manually configured firewall rules and custom scripts. The honeypot emulates a single specific device, a Koyo DirectLOGIC 405 PLC, with the HTTP and Modbus services. Although this honeypot performs well, it has limited applicability because the Gumstix hardware is restricted to a single PLC configuration and each service must be manually customized.

Occasionally, research is conducted into the nature of the results obtained from honeypots rather than improving the technology itself. The studies by Kyle Wilhoit described in Section 2.2.2 deployed several ICS honeypots of varying types and functions around the world in order to determine the nature of ICS attacks [29, 30]. The honeypots used in these experiments were all manually configured using Honeyd and specific service scripts such as HTTP, FTP, and Modbus.

2.3.2 Dynamic honeynets.

In order to increase flexibility and reduce workload on administrators, it is beneficial to automate the creation of a honeypot network. A *dynamic honeynet* is a system with some degree of automation involved in the configuration process. Typically these systems only automate the configuration of the transport layer of the network stack and below; application layer protocols must be handled either manually or with some other strategy, if the application layer is emulated at all. These systems tend to sacrifice fidelity of interaction for performance and scalability with the goal of deceiving network scanning tools or detecting patterns of traffic.

One creative dynamic system attempted to capture Internet Protocol version 6 (IPv6) address scans to determine if the advent of IPv6 is providing attackers a fruitful avenue of attack [48]. The system uses custom VM images of common Operating Systems (OSs) such as Windows and Linux as manually configured honeypots. Based on the common practice of creating IPv6 addresses from MAC addresses, the system assigns the closest matching honeypot image to an incoming IPv6 probe. This means any unused IPv6 space can potentially be bound to a honeypot during a scan. The dynamic characteristic is seen in the binding of appropriate images to IP addresses where the images are all manually configured.

As discussed above, Honeyd is a common LI honeypot framework capable of emulating the transport layer and below. It is possible to create a dynamic honeynet using Honeyd by automating the host generation process based on observed network traffic. The simplest method is to quietly observe, without generating traffic, until the network is fully mapped. Hieb used this technique, called *passive sniffing*, with `p0f` and `tcpdump` to sniff traffic in order to determine the OS and ports to deploy within a Honeyd honeynet [15].

A less stealthy, but faster and more reliable, method of sniffing network traffic is *active sniffing*. Rather than wait for target traffic, an active system generates traffic in order to observe desired responses. This approach risks honeypot exposure if an attacker observes the active scans. Hecker, Hay, and Nance converted the passive system created by Hieb above into an active one by replacing `p0f` and `tcpdump` with the scanning tool `Nmap` [49].

Some systems combine the strengths of both techniques by using passive and active sniffing together. Kuwatly et al. use `p0f`, `tcpdump`, and `Nmap` [16]. Hecker and Hay extended their previous work to include varying levels and mixtures of passive and active sniffing [14]. The lowest noise level, passive only, uses `p0f` and `tcpdump`. The highest noise level uses `Nmap` and `xprobe2` to scan the target network in addition to the passive tools. Intermediate levels use combinations of these tools.

A more sophisticated system, designed by Vollmer and Manic for ICS, leverages `Ettercap` to perform passive mapping of a network [13]. Using ports, OS fingerprints, and MAC addresses the honeynet can be configured with the correct VM images running the most probable services. All of the configuration data is stored in an Extensible Markup Language (XML) document which allows for easy modification should the network change. This system was integrated into an Automatic Intelligent Cyber-Sensor which acts as a combination honeypot/IDS capable of detecting over 99% of anomalous traffic [6].

2.3.3 Automatic protocol emulation.

In contrast to the dynamic honeynets described in the last section, it is also possible to automate the process of configuring application layer scripts for Honeyd. These dynamic honeypots achieve higher levels of fidelity with less human interaction. The idea is to observe port numbers and protocol traffic in order to determine what well-known protocols are being used. With this information, configuration of those protocols or services can be automated through the use of preconfigured scripts. The techniques for sensing the network are numerous and often application specific.

A good example of such a system is BAIT-TRAP, a “catering” honeypot framework designed by Jiang and Xu [50]. Their system contains a database of possible service scripts which can be deployed dynamically to a set of virtual or physical honeypots. The honeypot configuration at any moment depends on the state of the network and the protocols and ports observed.

A further step of flexibility is taken by Chowdhary et al. by combining observed protocol traffic and knowledge about the protocol to emulate it in a technique they refer to as *service mining* [51]. This technique avoids using a static, preconfigured script and emulates known services in context. Although it requires an initial database of service information, the system worked well in experimentation by accurately emulating FTP.

Fink did similar work in configuring known protocols, HTTP and TCP, based on observed traffic [52]. Her system is able to emulate the web interface of an arbitrary PLC by using `wget` and `tcpdump`. By extracting TCP field information from the `tcpdump` results during a web page request, the emulator can modify its own TCP functionality to mimic the target device.

An advanced technique for extracting useful protocol information from network traffic is *clustering*. This method groups similar packets together into groups. Rafique et al. used this technique to create algorithms for network dialog minimization and determining differences between dialogs [53]. These algorithms were shown to be useful for traffic verification through IDS signature generation.

2.3.4 Hybrid honeypots with replay.

In order to create increasingly flexible application layer emulators, the analysis of network traffic must be taken to the extreme: complete protocol reverse engineering and dynamic configuration of unknown services. To this end several tools have been designed to implement advanced protocol reverse engineering algorithms and adaptive replay.

2.3.4.1 Protocol-agnostic replay.

A straightforward program for replaying traffic is RolePlayer [54]. RolePlayer works as part of a proxying Honeyd instance. Application layer traffic is sent to a real device while RolePlayer observes the responses. After a given amount of traffic, RolePlayer can replay those packets in context. Should the program arrive at a state for which it has no response, it can replay the whole conversation back to the proxy device in order to learn how to respond. This can occur without the attacker detecting the switch.

A more sophisticated version of the replay idea is to actually create a brand new Honeyd service script from an observed network trace as is done in ScriptGen [18]. ScriptGen uses state machines to determine the structure of a traffic dialog without knowing anything about the protocol or its implementations on the server or client. This state machine can be simplified and converted into a Python script usable by Honeyd. A later version of ScriptGen improves performance in the presence of inter- and intra-protocol dependencies [23]. It also adds the ability to dynamically update ScriptGen's state machines based on proxy traffic similar to RolePlayer.

ScriptGenE, a further extension of ScriptGen by Warner, adds the ability to handle difficult cases such as unknown transitions and default responses during session replay [19]. The *protocol-agnostic* design is intended specifically for ICS systems where PLCs may use proprietary protocols. An extension of ScriptGen, the ScriptGenE framework constructs protocol trees as the finite state machine of a protocol. These could be converted into Honeyd scripts or ScriptGenE itself can access the trees as a subsystem of Honeyd in order to replay the conversations.

2.3.4.2 Hybrid honeypots.

The first fully hybrid honeypot to implement protocol-agnostic replay using RolePlayer was GQ [22, 55]. This system was designed to capture and analyze worms and similar malware. The system uses RolePlayer as part of a front-end controller tasked with filtering incoming traffic, containing dangerous outbound traffic, and assigning new honeypots to key traffic. The back-end honeyfarm has the common structure of modular VM honeypots ready to be deployed. These honeypots can be replaced due to their modularity. GQ has been maintained and improved in order to track new malware in a contained environment.

SGNET, like many of the honeypot frameworks discussed so far, follows this same strategy but distributes its sensors instead of having a centralized controller [21]. These ScriptGen-based sensors individually decide whether or not to route traffic back to a gateway over a custom HTTP-like protocol called *Peiros*. Within the gateway a private network of Argos-based *sample factories* and Nepenthes-based *shellcode handlers* determine the behavior of malicious code captured from traffic. Information on malware traffic patterns can be fed back to the sensors for improved filtering. An improvement on the malware containment feature called Mozzie also uses ScriptGen to emulate the parties during malware communication in order to study the interaction without releasing dangerous traffic [56].

A final hybrid honeypot system called AWESOME, or Automated Web Emulation for Secure Operation of a Malware-Analysis Environment, uses a collection of tools seen above [57]. Honeyd and Argos are used for LI and HI honeypots respectively while ScriptGen and Nitro are employed for replay and malware analysis. This system is intended to collect and analyze malware similar to SGNET.

2.4 Chapter Summary

This chapter examines the current state of ICS technology and security. These systems, though they are a key component of national critical infrastructure, are inherently insecure due to weak designs based on universal trust. State-based security paradigms are important for securing such a network. Honeypots are a flexible sensor technology shown to be useful in analyzing the state of a network.

The ideal honeypot has a high degree of flexibility, scalability, and authenticity. Though these factors often conflict with each other, creative combinations of LI and HI honeypots can achieve high levels of performance. However, configuration and maintenance are significant limitations. Hybrid honeypots such as SGNET, GQ, and AWESOME attempt to provide flexibility and automation to the process.

Being designed for wide scale malware analysis, most of these powerful hybrid systems have not been publicly released, nor have they been tested on ICS networks. Other solutions like Honeyd+ are easy to deploy but lack flexibility and performance guarantees. The need remains for an open-source, dynamic honeypot framework for ICS networks. This honeypot should be protocol-agnostic in order to effectively emulate the common proprietary ICS protocols. In addition, the honeypot should be easy to configure and deploy in a production environment.

III. Framework Design

3.1 Overview

This chapter details the design and development of an automatic, application layer PLC emulator. The emulator uses trace-based dynamic replay alongside a novel *incremental update* function via a proxy connection. This emulator is an extension of the ScriptGenE framework intended to be used as a removable component of a Honeyd+ honeynet configuration. The motivation for these choices is provided in Section 3.2. An in-depth look at the ScriptGenE framework is also provided in Section 3.4.2 as the starting point for emulator design. The resulting extended emulator provides flexible, automatic replay capability with limited dependence on the back-end PLC.

3.2 Motivation and Application

Winn's production honeynet framework, Honeyd+, achieves application level authenticity through a proxy to a back-end PLC [20]. Winn's configuration is illustrated on the left side of Figure 6. Honeyd+ is able to replace device-specific data in the proxy responses to create the illusion of multiple identical, but independent, PLCs. For maximum scalability, the framework is configured to allow many geographically-distributed Honeyd+ instances proxy access to a single back-end PLC. Winn found this configuration may create overwhelming traffic loads on the PLC. Alleviation is challenging as adding more PLCs or cutting back Honeyd+ instances each limit scalability due to cost and flexibility respectively. The ideal solution involves allowing an arbitrary number of Honeyd+ instances and a single back-end PLC while reducing the amount of traffic forwarded to the PLC.

One practical solution is an intermediary providing application layer emulation as depicted on the right side of Figure 6. This intermediary would replace the PLC as the target of Honeyd+ proxies thereby reducing traffic to the PLC. To maintain authenticity, the intermediary itself should be able to proxy to the back-end PLC. The emulation functionality of the intermediary would reduce the proxied traffic and alleviate the load on the PLC. This research provides a proof of concept version of this intermediary by extending the stand-alone functionality of the ScriptGenE replay component. ScriptGenE was chosen as the emulator framework because it provides an automatic replay function based on training captures. This allows the emulator to immediately generate some of the PLC traffic without proxying. The extensions to ScriptGenEreplay described in this chapter add the ability to proxy unknown messages and dynamically update the protocol tree during emulation.

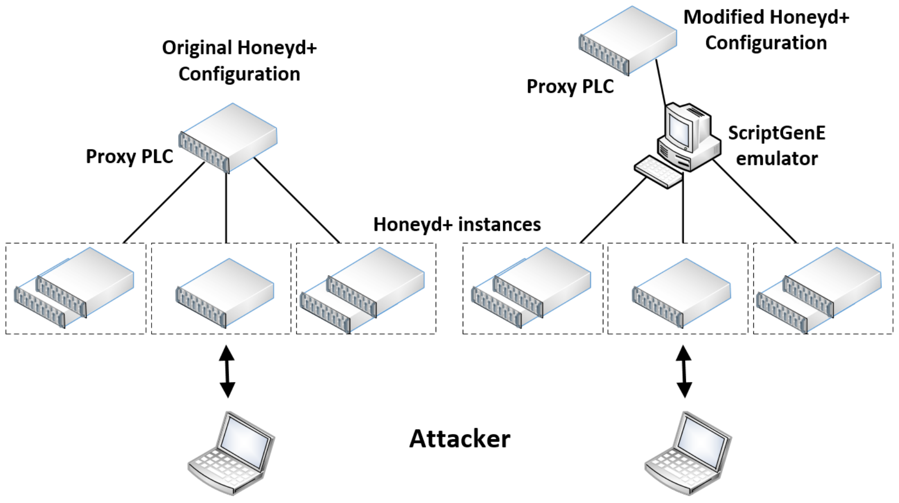


Figure 6. Honeynet with intermediate application layer emulator

3.3 Design Parameters

The extensions to ScriptGenE add new functionality to the existing replay emulator to improve performance and authenticity. Therefore the design goals focus on the improvements rather than overall emulator design. The following are the design parameters under consideration in extending the ScriptGenE framework:

Authenticity:

1. Emulator queries a physical device for responses to unrecognized client requests
2. Emulator completes a task without having been trained on all required task traffic

Performance:

1. Emulator handles multiple client connections simultaneously
2. Emulator updates replay capabilities with new responses to avoid future forwarding
3. Emulator forwards less traffic to the proxy target than it receives from the client
4. Emulator provides several conversation context synchronization algorithms to maximize performance with varying protocols

Flexibility:

1. Emulator is protocol-agnostic (no assumptions about protocol type)
2. Emulator can be added to an existing honeynet
3. Emulator can be automated

3.4 The ScriptGenE Framework

As discussed in Section 2.3.4.1, the ScriptGenE framework is an extension of the Honeyd script generation software ScriptGen [18, 23]. Developed by Warner, ScriptGenE contains many functionality enhancements centered around protocol-agnostic replay [19].

3.4.1 Framework overview.

ScriptGenE consists of several Python files and supporting third party libraries. Functionality centers around protocol trees (p-trees), the finite state machine of ScriptGenE in which edges and nodes represent client and server messages respectively. Generation, manipulation, and replay of p-trees can be automated to create an application layer emulator. A high level view of the framework is shown in Figure 7.

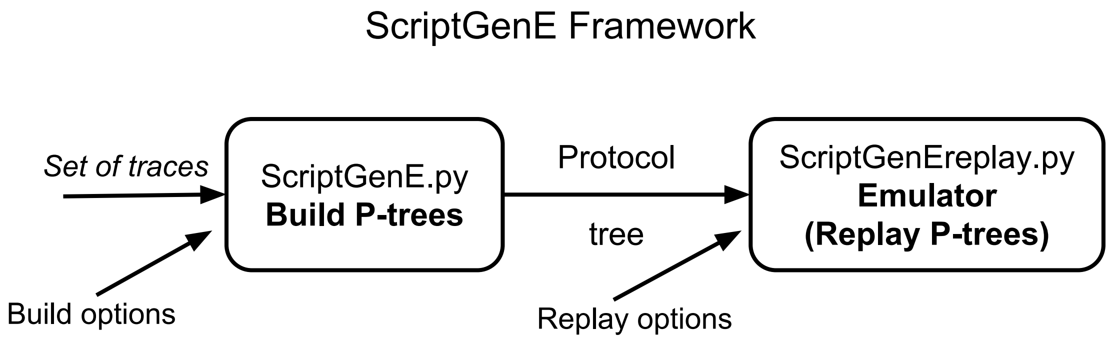


Figure 7. ScriptGenE framework overview [19]

Primary ScriptGenE files are:

- ScriptGenE.py - Builds initial and generalized p-trees
- GeneralizeTree.py - Generalizes initial p-trees built by ScriptGenE.py
- CombineGtrees.py - Combines p-trees at their root
- ScriptGenEreplay.py - Loads p-trees and replays server messages in context during conversations with clients
- clientReplay.py - Provides client-side replay of p-trees for testing
- diffPcaps.py - Computes byte-level differences between two packet captures (pcaps) by consolidating messages

This research assumes protocol trees are already built and exclusively addresses replay of existing p-trees. For this reason, framework details are limited to those relevant to the replay extensions. See Warner's description of the ScriptGenE framework for more information on how p-trees are built and manipulated [19].

3.4.2 ScriptGenE.py.

ScriptGenE.py is the file responsible for building protocol trees. The build process begins with a set of pcaps which are parsed into separate trees for every complete connection found. Nodes in a tree represent server responses while edges represent client messages as illustrated in Figure 8. The initial trees are consolidated over two rounds to create generic p-trees representing abstract protocol behavior.

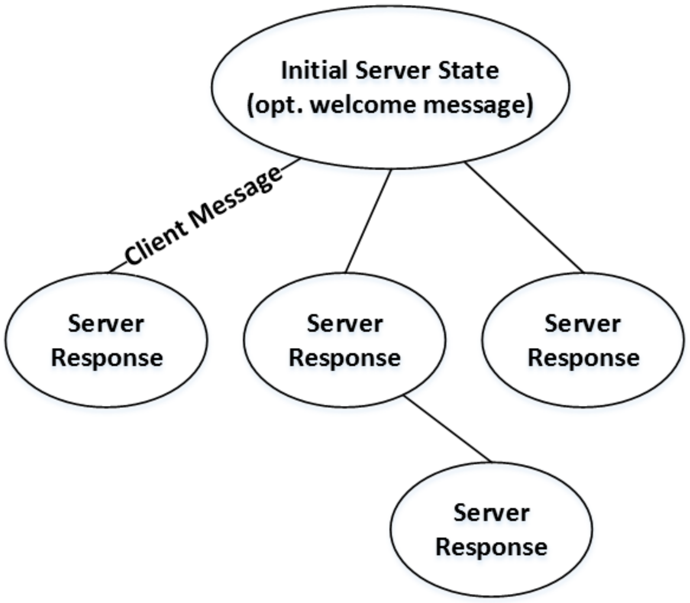


Figure 8. Protocol tree structure

Consolidations produce generic messages with fixed fields, which always contain the same data; variable fields, which may have changing data; and environmental fields, which hold information related to the connection and its participants. The environmental links are replaced with markers allowing the replay emulator to find and insert environmental data. Intra-protocol dependencies, links between corresponding client and server messages are also detected and similarly marked. After a default error message is derived from the finished p-tree, the tree object is written to a Python pickle file which can be loaded for replay later.

3.4.3 ScriptGenEreplay.py.

ScriptGenEreplay.py is the emulator file in the ScriptGenE framework. It loads a p-tree pickle file at runtime and emulates the server side by opening a server socket. Protocol context starts at the p-tree root. ScriptGenEreplay matches incoming client messages to the edges of the current context node. A match shifts the current context to the child node along the matched edge. The server message in the new node is then sent to the client as the server response.

The emulator opens its server socket on a port number specified by the user through runtime options. This is not necessarily the same port that the original p-tree traffic used. The IP address of the emulator is extracted from the user-specified interface provided at runtime. Only one client connection is allowed at a time. When the client connects, all client and server environment information (e.g., IP addresses, ports, hostnames) are loaded into a dictionary to be used for environmental link replacement.

Every byte received from the client is checked for a match in the p-tree. The emulator may determine more data is needed before declaring no match to be found. When sufficient client data arrives and a match cannot be found at the current context, the client data is said to represent an unknown transition (edge) in the p-tree. Unknown transitions are handled in one of two ways: backtracking and default error messages. Backtracking starts by searching the path from the current context to the root to determine if a match can be found in an earlier context. This provides session looping capabilities. If a match is still not found, backtracking extends to search the entire p-tree for any match. If the p-tree contains no matches, the default error message created during the tree building process is sent to the client. The error state is maintained so that future client requests receive the error message repeatedly until the connection closes. The default error message found during build can be overridden at replay runtime through user configuration.

Server messages in the p-tree contain markers for environmental and intra-protocol links as described in Section 3.4.2. When server messages are retrieved from the p-tree to be sent, these markers are replaced with the appropriate information from the client connection. It is possible that the original and new data may have different lengths. These lengths are accounted for during replacement, but undetected length fields elsewhere in the packet may be rendered incorrect after replacement. Because of this known issue, ScriptGenEreplay provides the option of using the original data during replay.

3.5 ScriptGenEreplay Extensions

3.5.1 Overview.

This research modifies the ScriptGenE framework by directly extending the ScriptGenEreplay.py file into ScriptGenEemulate.py. The extensions consist of modifications to existing functions and newly developed functions totaling approximately 440 lines of Python code. All ScriptGenEreplay.py functionality is still available in ScriptGenEemulate.py, but the new file provides additional runtime options and extended emulation capabilities. The following sections detail the changes and extensions within ScriptGenEemulate.py.

3.5.2 Usage.

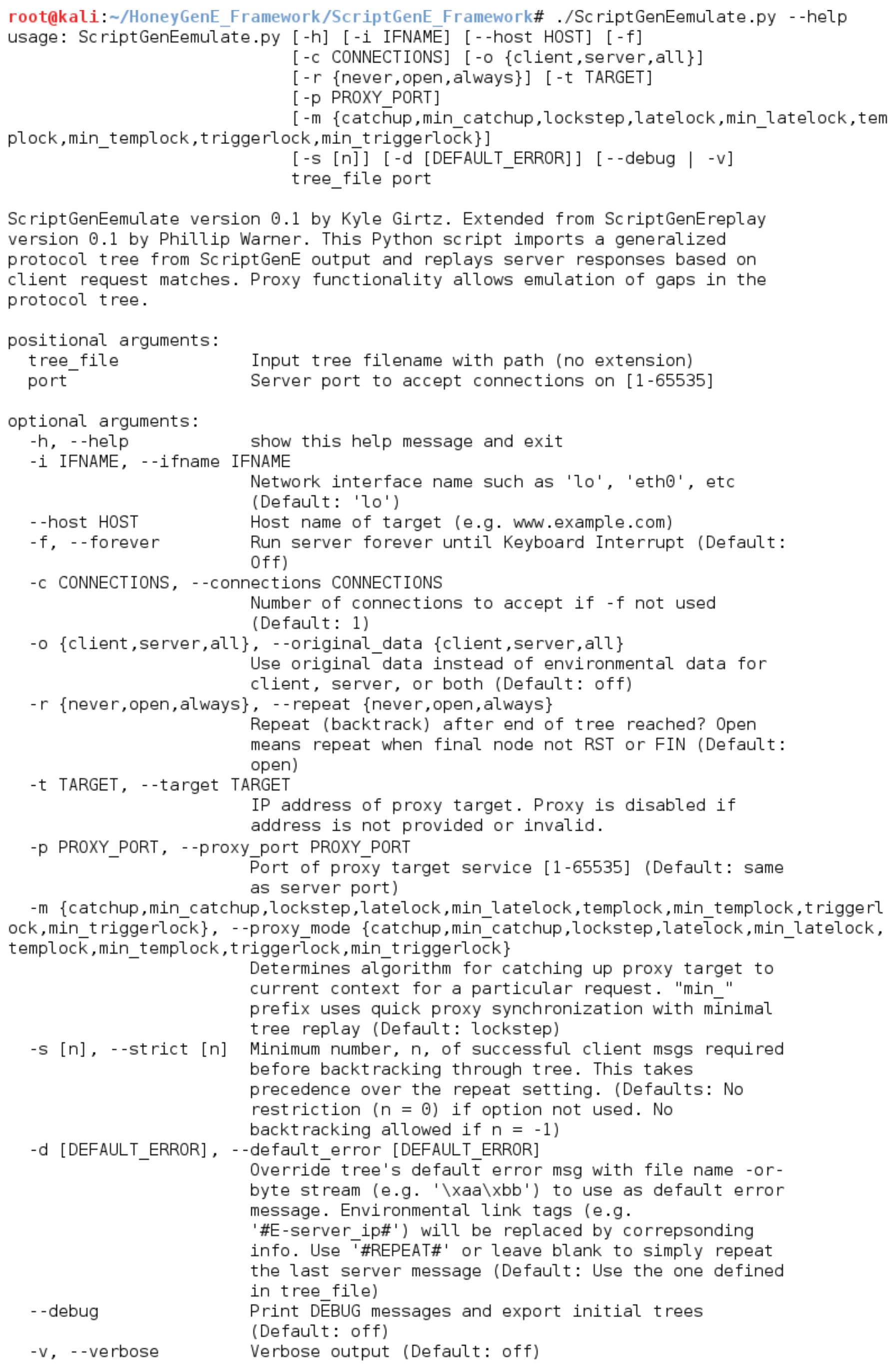
The ScriptGenE framework was designed for a Linux environment. ScriptGenEemulate.py is run from the Linux command line. The command line options are shown in Figure 9.

Most of the ScriptGenEemulate.py options are the same as the original ScriptGenEreplay.py. The required options are the name of a p-tree file to replay and a port number on which to open the server socket. There are also standard help, debugging, and verbosity options. Environment options like the interface, host name, and original data options define the information used to replace the environmental link markers during replay. Connection options include the '`--forever`' switch and '`--connections`' variable.

The number of connections in ScriptGenEemulate is slightly different than that of ScriptGenEreplay. ScriptGenEreplay.py can only handle a single client connection at a time so this value denotes the number of connections to add to its backup queue. ScriptGenEemulate.py can handle multiple connections, and this value represents the total number of connections to accept before terminating the emulator. It can be overridden by the '`--forever`' option.

The '`--strict`', '`--repeat`', and '`--default_error`' options determine the behavior of the emulator when unknown transitions occur. The first two options indicate when backtracking should occur and how to handle leaves in the tree while the default error option allows the user to specify a custom error message.

Besides the modified meaning of the '`--connections`' option, the only new options are '`--target`', '`--proxy_port`', and '`--proxy_mode`'. These options specify how the emulator should connect to the back-end proxy device. '`--target`' and '`--proxy_port`' specify the IP address of the proxy device and the TCP port number on which the proxy device is running the emulated service. '`--proxy_mode`'



```
root@kali:~/HoneyGenE_Framework/ScriptGenE_Framework# ./ScriptGenEemulate.py --help
usage: ScriptGenEemulate.py [-h] [-i IFNAME] [--host HOST] [-f]
                            [-c CONNECTIONS] [-o {client,server,all}]
                            [-r {never,open,always}] [-t TARGET]
                            [-p PROXY_PORT]
                            [-m {catchup,min_catchup,lockstep,latelock,min_latelock,tem
plock,min_templock,triggerlock,min_triggerlock}]
                            [-s [n]] [-d [DEFAULT_ERROR]] [--debug | -v]
                            tree_file port

ScriptGenEemulate version 0.1 by Kyle Girtz. Extended from ScriptGenEreplay
version 0.1 by Phillip Warner. This Python script imports a generalized
protocol tree from ScriptGenE output and replays server responses based on
client request matches. Proxy functionality allows emulation of gaps in the
protocol tree.

positional arguments:
  tree_file             Input tree filename with path (no extension)
  port                  Server port to accept connections on [1-65535]

optional arguments:
  -h, --help            show this help message and exit
  -i IFNAME, --ifname IFNAME
                        Network interface name such as 'lo', 'eth0', etc
                        (Default: 'lo')
  --host HOST           Host name of target (e.g. www.example.com)
  -f, --forever         Run server forever until Keyboard Interrupt (Default:
                        Off)
  -c CONNECTIONS, --connections CONNECTIONS
                        Number of connections to accept if -f not used
                        (Default: 1)
  -o {client,server,all}, --original_data {client,server,all}
                        Use original data instead of environmental data for
                        client, server, or both (Default: off)
  -r {never,open,always}, --repeat {never,open,always}
                        Repeat (backtrack) after end of tree reached? Open
                        means repeat when final node not RST or FIN (Default:
                        open)
  -t TARGET, --target TARGET
                        IP address of proxy target. Proxy is disabled if
                        address is not provided or invalid.
  -p PROXY_PORT, --proxy_port PROXY_PORT
                        Port of proxy target service [1-65535] (Default: same
                        as server port)
  -m {catchup,min_catchup,lockstep,latelock,min_latelock,templock,min_templock,triggerl
ock,min_triggerlock}, --proxy_mode {catchup,min_catchup,lockstep,latelock,min_latelock,
templock,min_templock,triggerlock,min_triggerlock}
                        Determines algorithm for catching up proxy target to
                        current context for a particular request. "min_"
                        prefix uses quick proxy synchronization with minimal
                        tree replay (Default: lockstep)
  -s [n], --strict [n]  Minimum number, n, of successful client msgs required
                        before backtracking through tree. This takes
                        precedence over the repeat setting. (Defaults: No
                        restriction (n = 0) if option not used. No
                        backtracking allowed if n = -1)
  -d [DEFAULT_ERROR], --default_error [DEFAULT_ERROR]
                        Override tree's default error msg with file name -or-
                        byte stream (e.g. '\xaa\xbb') to use as default error
                        message. Environmental link tags (e.g.
                        '#E-server_ip#') will be replaced by correpsonding
                        info. Use '#REPEAT#' or leave blank to simply repeat
                        the last server message (Default: Use the one defined
                        in tree_file)
  --debug               Print DEBUG messages and export initial trees
                        (Default: off)
  -v, --verbose         Verbose output (Default: off)
```

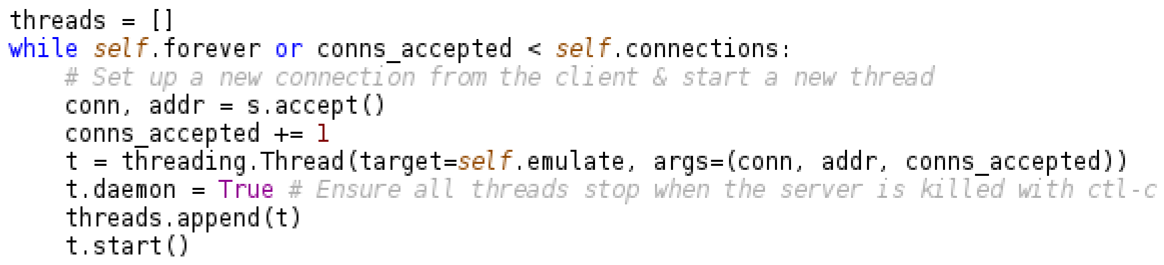
Figure 9. ScriptGenEemulate.py usage

allows the user to choose any of the proxy synchronization algorithms described in Section 3.5.5.3. The currently-available algorithms are `catchup`, `lockstep`, `latelock`, `templock`, and `triggerlock` as well as the minimal synchronization versions of each (see Section 3.5.5.3). Setting the mode to `off`, the default option, will disable the proxy and cause the emulator to ignore the other proxy options.

3.5.3 Initialization.

The main ScriptGenEemulate.py function verifies all input arguments before calling the emulator function. The emulator loads the p-tree and begins listening on a server socket. The socket is configured in code with three custom settings. The *nagling* option is turned off in order to replay small packets exactly as they appear in the tree rather than combining them. *Keep-alive* packets are configured to send every eight seconds during idle connection time as an attempt at keeping the client interested. Finally, the *reuse address* option is turned on in order to avoid having port reuse conflicts when running the emulator multiple times in a short period of time.

Next, the emulator function enters a while loop and tries to accept client connections. Each accepted connection is passed to a new thread along with the client address information and a connection identifier as shown in Figure 10. This behavior is a significant difference from ScriptGenEreplay.py. The replay file structure consists of a single emulator function call. The new structure encapsulates the emulator into an object with a threadable replay method which is called for every new connection. The object model provides two primary benefits. First, the emulator can now service multiple, concurrent client connections. Second, the emulator is able to maintain a unique proxy connection within the thread assigned to each client connection so that proxy conversations do not affect one another.



```
threads = []
while self.forever or conns_accepted < self.connections:
    # Set up a new connection from the client & start a new thread
    conn, addr = s.accept()
    conns_accepted += 1
    t = threading.Thread(target=self.emulate, args=(conn, addr, conns_accepted))
    t.daemon = True # Ensure all threads stop when the server is killed with ctl-c
    threads.append(t)
    t.start()
```

Figure 10. Loop assigning client connections to new threads

Once the thread is started, dictionaries are defined for client and proxy environment information. This information includes IP addresses and port numbers for both sides and the server host name. The proxy dictionary also holds the proxy connection object to make it easy to pass among the various functions requiring access.

It should be noted that the CPython *Global Interpreter Lock* prevents Python programs from truly multi-threading on the processor. However, because the emulator application is I/O bound, this limitation is not expected to significantly restrict emulator performance. Future implementations may remove this restriction for high traffic environments.

3.5.4 Handling unknown transitions.

As in ScriptGenEreplay.py, the new ScriptGenEemulate.py also attempts to backtrack when an unknown transition is encountered. The backtracking algorithm as described in Section 3.4.3 has not been modified. However, when backtracking fails to find a match in the p-tree, the emulator can be configured with the options in Section 3.5.2 to use its proxy before sending a default error message. The proxy allows the emulator to query a live PLC and forward its response to the client. The proxy mechanism is discussed in more detail in Section 3.5.5.

Using the proxy introduces additional latency in the emulator's response to a client request. The added delay depends on the protocol being emulated and the current context of the client conversation. Minimization of these delays is part of the motivation for the algorithms discussed in Section 3.5.5.3.

3.5.5 Proxy connections.

The most significant improvement to the ScriptGenE framework is the addition of a proxy mechanism. The proxy allows the emulator to establish connections to a live PLC, mirror the context of the client connection to the PLC, send an unrecognized message, and collect the response for future replay. When client bytes are designated as an "unknown transition" they can be proxied according to the following steps:

1. Replace environmental information in the unrecognized packet.
2. Synchronize the conversation context with the PLC (depends on the context algorithm).
3. Send the unrecognized packet to the PLC and collect the PLC response.
4. Close/maintain the PLC connection (depends on the context algorithm).
5. Replace environmental information in the PLC response.
6. Update the p-tree with the PLC response.
7. Shift the current p-tree context to the added node and send the server response to the client.

Some of these steps may be omitted or reordered depending on the context synchronization algorithm used. Synchronization and PLC connection handling are discussed in Section 3.5.5.3. Replacing environmental information is another key issue discussed in Section 3.5.5.4. The other key steps are explained below.

3.5.5.1 Proxy send and receive.

The `proxy_new_request` function performs the context synchronization by calling `synchronize_proxy`. It then calls `proxy_send_recv` to exchange the unrecognized request and receive the PLC response. If the proxy connection is broken or errors occur at any time during this process, the proxy connection is closed and reset while the emulator reverts to the default error message technique.

3.5.5.2 Update protocol tree.

After the proxy exchange is complete, the new server message is added to the p-tree with the function `update_tree`. Here, a new child node containing the PLC response is added to the p-tree in memory. A new edge containing the unrecognized client message connects the current context node to its new child. Future requests like this unknown transition will now find a match and be replayed directly rather than being proxied again. The result of an example update is shown in Figure 11.

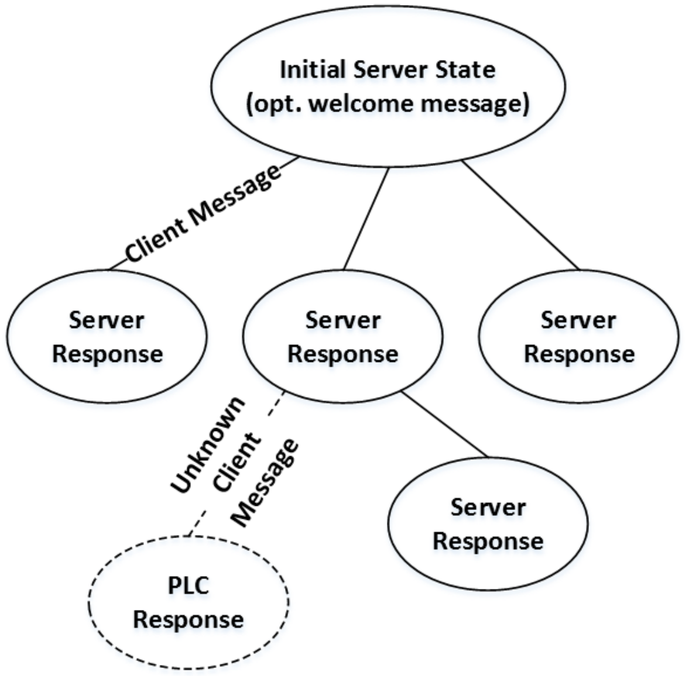


Figure 11. Protocol tree after a dynamic update

Modifications to the p-tree are performed by the `add_node` and `add_edge` methods from the `tree` class. This class is an extension of the `DiGraph` class from the `networkx` package, a toolset for building and manipulating various types of graphs in Python. The new edge and node are based on a single message exchange, so all of the generic attribute information is set to default values.

Updating the p-tree in memory is a temporary solution. The updated tree is not saved when the emulator is terminated. A p-tree is a generic structure constructed from multiple traces containing the same kind of traffic. Updates add a single, real message instance into an abstract protocol tree. Any bytes that vary in the proxied message and its response will go undetected. It is safer to record all proxy traffic from outside the emulator and build a new p-tree for future emulation. This technique will ensure that only the desired new traffic is collected to avoid excessive pcap files. Improving the dynamic update functionality is an important area of future work for an efficient emulator.

3.5.5.3 Context Maintenance Algorithms.

One of the challenges in updating emulation capabilities through a proxy is synchronizing conversation context with the PLC. The problem arises when an unknown client request occurs while the replay state is deep in the protocol tree. In order for the PLC to return the appropriate response to this new request it must understand the current context of the conversation.

Synchronizing the client conversation context with the PLC requires sending each client message in the path from the current p-tree node up to the root in reverse order, effectively replaying the client side of the conversation to the PLC. These messages can be sent all at once, individually as they are received from the client, or in any other manner, but they must *all* be sent to guarantee the correct context. Failure to transfer the full context may not cause a problem in every case, but it is impossible to know when it is or is not necessary.

This research proposes several context-maintenance algorithms. In its non-minimal form, each algorithm performs synchronization as described above. See the **Minimal Synchronization** paragraph below for an explanation of the minimal versions. The core differences between each algorithm described below lie in the conditions for opening and closing proxy connections. In the event of unexpected proxy errors, the proxy connection is broken down immediately regardless of the algorithm chosen.

Catch up. The *catch up* algorithm does not maintain a persistent connection with the back-end PLC. The following process is carried out when a new request requires proxying:

- A proxy connection is established.
- The PLC is caught up on the conversation context.
- The new request is forwarded.
- The new PLC response is recorded and forwarded to the client.
- The proxy connection is disconnected.
- The p-tree is updated with the new response data.

The catch up algorithm is ideal from the perspective of the back-end PLC. No proxy connection is maintained and only the minimum necessary traffic is proxied. However, from the perspective of the client, there may be long delays if the emulator must repeatedly catch up the PLC on consecutive requests while deep in the protocol tree. In the case of very long conversations this delay may realistically be thirty seconds to several minutes. This delay could potentially tip off an attacker to the presence of the honeypot.

Lockstep. The *lockstep* algorithm maintains a persistent connection with the back-end PLC as an exact mirror of the connection with the client. A proxy connection is established when a client connection is established and is broken down only when the client connection is lost. All client traffic is immediately forwarded to the PLC whose responses are ignored. In the case of unknown requests, the traffic is forwarded as normal, but the response is sent to the client rather than ignored.

The lockstep algorithm is ideal from the perspective of the emulator and client. In this mode, a proxy connection is maintained at all times as an exact mirror of the client connection. This means a new request requires no delay for establishing context; it can be sent immediately. However, from the PLC's perspective, this algorithm is the worst-case option. A proxy connection is maintained throughout the entire life of the client connection, and all traffic received is forwarded to the PLC. This may be overwhelming for devices with limited communications resources. In terms of PLC performance, this algorithm is identical to replacing the emulator with the actual device. No traffic is filtered.

Latelock. The *latelock* algorithm is a straightforward combination of the catch up and lockstep algorithms. For this algorithm, no proxy connection is established until a request needs to be proxied at which point the entire conversation context is transferred. This is identical to the catch up process. However, once a proxy connection is established and synchronized, the connection is maintained indefinitely as in the lockstep algorithm rather than being broken down immediately as in the catch up algorithm.

The latelock algorithm improves on the lockstep algorithm by only establishing a proxy connection if it is needed. In this way, PLC resources are conserved in the event that no forwarding is necessary. When the proxy is needed for a new request, a long delay may occur during context synchronization. However, after the proxy is connected this algorithm displays the same benefits and drawbacks as the standard lockstep algorithm.

Templock. The *templock* algorithm is identical to the latelock algorithm until a proxy connection is established and maintained. Rather than maintaining the proxy connection indefinitely, templock terminates it prior to the end of the client connection. The proxy connection is broken down as soon as a client request is handled entirely by the replay p-tree (no proxy is required). As long as new requests require the proxy connection, the connection is maintained. When it is no longer needed, the proxy connection is closed and the algorithm returns to the waiting state until a new request must be proxied. In this way, many proxy connections may be established over the lifespan of a single client connection.

The templock algorithm has all of the same performance characteristics as the latelock algorithm with one improvement. Once a proxy connection is established, there is a chance it may be disconnected before the client conversation terminates. This behavior allows the emulator to establish a proxy connection only when necessary, maintain it over a series of requests, and close it when it is no longer needed. This is ideal behavior from the perspective of the PLC. The primary drawback is the familiar synchronization delay.

All of the algorithms described thus far begin and end proxy connections based on the nature of received client messages. These algorithms are visually represented in Figure 12 where the bars for each algorithm denote the duration of the proxy connection. The left side of each bar represents initial synchronization after which context is maintained until the end of the bar when the proxy connection is closed.

Triggerlock. The *triggerlock* algorithm alternates between maintaining context and waiting in the same manner as the templock algorithm. The difference is in

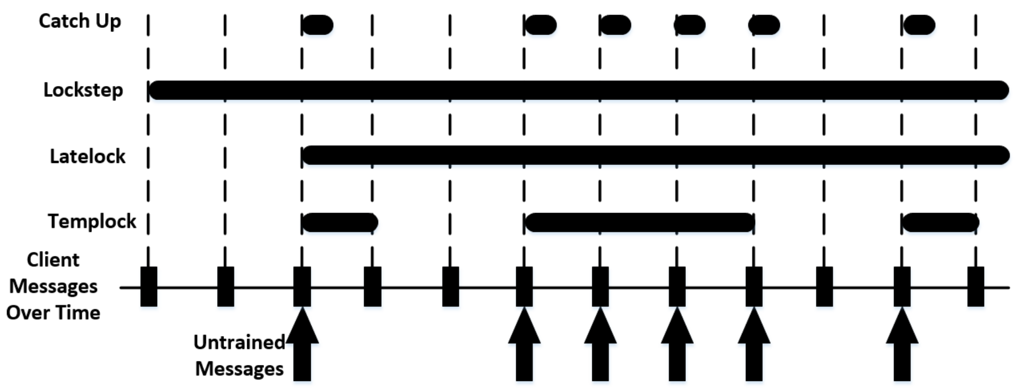


Figure 12. Context synchronization algorithm comparison

the conditions for establishing and breaking down the proxy connection. Templock establishes a connection at the first unknown request and breaks it down when a request is handled independent of the proxy. In contrast, triggerlock builds up and breaks down the proxy connection based on the current context in the p-tree. If the current node is deeper in the tree than a threshold depth, a proxy connection is established and synchronized regardless of the need for it. When the depth of the current node rises above the threshold (due to backtracking), the proxy connection is broken down.

The threshold (`trigger_thresh`) is defined as half of the maximum depth of the tree. It is limited to a minimum value of three in order to prevent superfluous connections on shallow p-trees. The example tree in Figure 13 depicts nodes below the threshold as filled while those above are empty. During replay of this tree, a proxy connection is maintained as long as the emulator context is at a filled node. While the current software version uses hardcoded values, the threshold and minimum trigger value could be made configurable.

The triggerlock algorithm sacrifices a small amount of PLC performance in order to reduce the synchronization delay. This is practically ideal behavior from the client

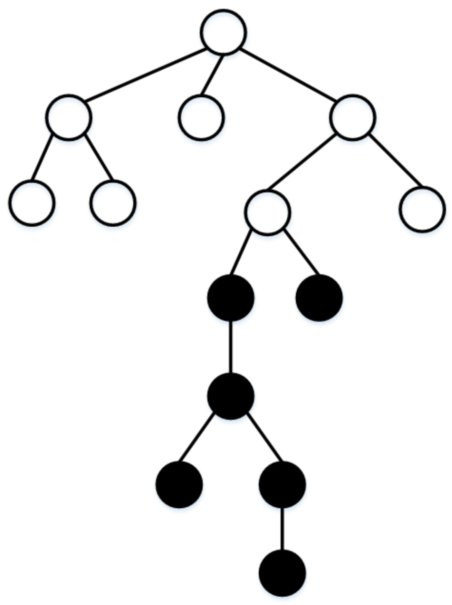


Figure 13. Triggerlock threshold example

perspective. However, it is possible that a proxy connection may be established unnecessarily, an acceptable risk due to the limited lifespan of the extra connection.

Minimal Synchronization. A final consideration is the setup phase of many connections. Once a session is established, any request may be sent in any order. The setup phase is typically linear in the p-tree with branches indicating setup completion. It is possible that a *minimal synchronization* may be possible for some protocols. Only the linear path from the root to the first branch is synchronized. This would allow the proxy connection to synchronize much more quickly regardless of the current conversation context. This optimization can be combined with any other approach to create a *minimal* variation of the algorithm.

Algorithm Summary. The correct choice of algorithm depends heavily on the nature of the protocol being emulated. For example, the catch up algorithm performs poorly on complex protocols with long conversations while performing ideally on a stateless protocol. Theoretically, the best algorithm choice for a complex, state-oriented protocol is either templock or triggerlock depending on the exact shape of the protocol tree. Triggerlock favors complex protocols with very deep trees due to its depth awareness. Protocols with relatively shallow trees fit better with templock as the synchronization delay is limited and unrecognized requests are likely infrequent.

3.5.5.4 Marking up client data.

As discussed in Section 3.4.2 and Section 3.4.3, the p-tree messages are marked with intra-protocol and environmental link indicators. The original replay function is able to replace these markers when replaying messages from the tree. The links are replaced in increasing offset order to maintain consistent offsets later in the package when length differences occur. All intra-protocol links are replaced before environment data is inserted. Link replacement is handled by the ScriptGenEreplay function `insert_link_data`.

The new proxy mechanism adds two complexities to the presence of link markers. First, the emulator must maintain a separate dictionary for proxy environment information as mentioned in Section 3.5.3. During proxy synchronization it is this dictionary that is accessed by `insert_link_data` to replace markers. Because synchronization replays consolidated data created by ScriptGenE.py, these markers already exist.

The second issue is that unknown transition messages will not contain link markers. Because the unknown transition is a single traffic instance, there is also no possibility of consolidation or checking for variable bytes in the client request or the new PLC response. For this reason, new packets require a search to determine if replacements are necessary. The function `replace_elinks` takes two environment dictionaries and a message as arguments. It searches the message for fields in one dictionary and replaces them with the corresponding fields in the other dictionary.

As described in Section 3.5.5 this process occurs before an unrecognized message is sent to the PLC and when the PLC response is received. The second replacement occurs before the p-tree is updated so that it contains data ready to be replayed to the client.

There is one type of protocol link data that cannot be resolved in the preceding fashion. If a protocol uses a *global link*, a field that is consistent across all packets during a connection (e.g., session ID), the current ScriptGenE.py p-tree generation algorithm will fail to recognize this field and insert no markers. This is not a problem for direct replay as intra-protocol links can handle potential problems. However, a proxied message must have a link value matching the new proxy connection. Similarly, responses from the PLC will fail to have the correct client connection value.

This research assumes a pre-built tree so detection must occur during replay. This is challenging as the global link may change values when the connection value is established making it difficult for a protocol-agnostic algorithm to locate a global link reliably. A strict algorithm will often fail to find the true link while more lenient algorithms will produce many false positives.

The current emulator does not have the ability to perform this check accurately during replay. Future software improvements should include global link detection during p-tree build. In its current state, ScriptGenEemulate is able to replace global links on messages passing between the client and proxy connections to ensure authenticity, but the location and length of the link field must be manually provided.

3.5.6 Design limitations.

As a proof of concept design, ScriptGenEemulate has several limitations. Like its ScriptGenE foundation, it can only handle IPv4 addresses and TCP protocols. Encrypted protocols are not supported. While ScriptGenE is intended to be automated and fully protocol-agnostic, the current software iteration requires some manual configuration for global protocol fields such as session IDs.

3.6 Design Summary

In summary, this chapter details the design of the extended ScriptGenE replay function: ScriptGenEemulate. This ICS emulator is designed to be flexible and authentic. Emulated responses originate from two sources: static packet-capture data and dynamic proxy responses.

New features within ScriptGenEemulate.py include:

1. multi-threading the server to enable handling multiple, concurrent client connections
2. a proxy for forwarding client data to a back-end device
3. forwarding PLC responses to unknown client requests in lieu of default error messages
4. incremental updating of the protocol tree in memory in order to improve future response capability

IV. Research Methodology

4.1 Goals

This research focuses on extending a framework for automatically configuring application layer PLC emulators by extending a working implementation as a proof of concept. Testing goals for this extension are derived from the design parameters discussed in Section 3.3. The following questions are addressed:

1. Can the extended emulator service a significant amount of client traffic without using the proxy?

2. Can a supplemental proxy provide more accurate emulation than a default error message in the event of incomplete training data?

3. Which PLC synchronization algorithm performs best with respect to client delays and PLC traffic load?

4.2 Approach

The ScriptGenE framework is extended into a practical, application layer PLC emulator. The base software generates generic protocol trees from observed traffic. The p-trees allow the emulator to replay either server or client traffic in the observed conversation. The extensions of this research further enable the emulator to respond to untrained traffic via a forwarding mechanism in conjunction with a real, back-end PLC. The details of the ScriptGenE framework and its extensions are covered in Chapter III.

Testing the emulator involves building baseline p-trees from traffic generated by ICS-related protocols, modifying these trees by removing nodes, and replaying the modified trees in the extended emulator. Each baseline p-tree reflects the conversation for a particular task performed by standard ICS management software tools. After random modifications to the baseline p-tree, the emulator is interrogated by the same tool and task to determine how the additional proxy functionality affects performance. Each task and p-tree are also performed for several PLC synchronization algorithms. Packet captures are collected for the client and proxy connections during each task. Measurements from each set of captures provide information on the PLC load reduction and synchronization algorithm performance. Measurements on the byte-level variability between captures created by the extended and original emulators reveal the extension's affect on authenticity.

All experimental tasks, regardless of protocol, synchronization algorithm, or p-tree, are randomly ordered and run in a single experiment. Two PLC control protocols and a web protocol are chosen due to their diversity of structure and reflection of typical PLC network activity.

4.3 System Boundaries

As shown in Figure 14, the system under test (SUT) is the extended ScriptGenE Framework. The primary component under test (CUT) is the proxy-enabled p-tree replay software (ScriptGenEemulate.py) which provides metrics for evaluating the system.

The workload into the system is a set of similar *protocol trees* with varying modifications. Each p-tree is generated by a different *task*. Emulation occurs when ScriptGenEemulate.py loads the p-tree for replay according to the specified *replay options*. Proxy connections to the *PLC configuration* are handled according to the specified synchronization algorithm.

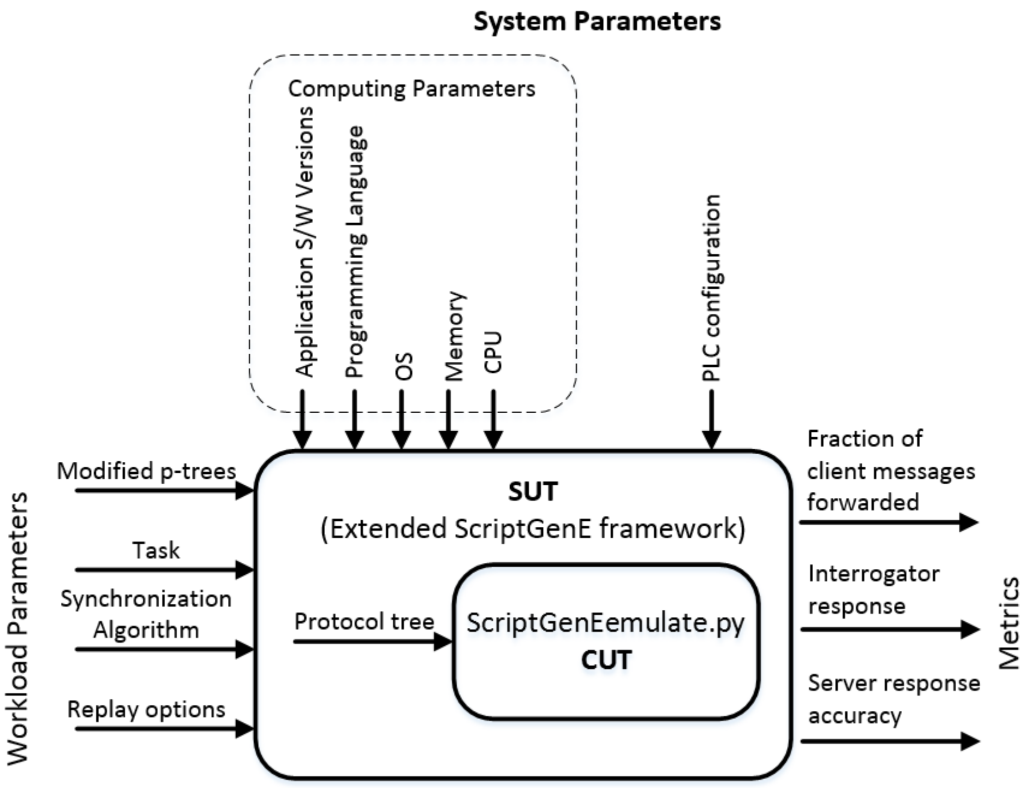


Figure 14. ScriptGenE emulator framework

Evaluation metrics consist of three parts. First, the fraction of client messages forwarded to the PLC is calculated as a percentage. Second, the client interrogator's response for each task is returned as a PASS or FAIL. Finally, server response accuracy is based on statistical analysis of variability between the baseline and modified p-tree captures.

4.4 Parameters and Factors

Any workload or system parameter that varies is called a *factor* with values called *levels*. The remainder of this section describes each parameter and its relevance to the SUT.

4.4.1 Workload parameters.

4.4.1.1 Task.

A single application task is selected for each of the chosen experimental protocols. The HTTP, ENIP, and ISO-TSAP protocols are chosen because they represent a variety of complexities, conversation structures, and purposes in ICS networks. They also provide a reasonable set of reconnaissance tasks that an attacker may leverage during network intrusion. Gathering web pages and browsing hardware modules using RSLinx or STEP7 gives the attacker more information about the configuration of the target system.

The `wget` tool is used to fetch a PLCs web pages over HTTP while RSLinx and STEP7 are used to browse module information for a PLC on the network. These tools operate over EtherNet/IP and ISO-TSAP, respectively. The runtime options for `wget` are:

```
-pr --no-parent --no-verbose --tries 1 $server -o $temp_file
```

These options perform a recursive download of all pages in the root directory of the target `$server`. All items on the page such as images and style sheets are downloaded. The `--tries` option specifies that each download is attempted exactly once. In the event of emulator errors, `wget` will not retry the download. Finally, the lack of verbosity and temporary file output allow the results to be checked for a PASS or FAIL result.

RSLinx and STEP7 are GUI-driven applications rather than command line calls like `wget`. Automation of these tasks requires the use of SikuliX [24]. Section 4.8 provides further detail on the configuration of SikuliX.

4.4.1.2 Modified protocol trees.

For each protocol task a set of baseline pcaps is collected during several successful interrogations of the PLC. The baseline captures are used to build the baseline p-trees for each protocol before the experiment begins. This process only takes a few minutes to create trees for all protocols. A verified initial tree is necessary to mitigate the risk that emulator errors are attributed to the tree-building software rather than the replay software. The experiment is designed to compare replay functions assuming a complete p-tree is already built. The ScriptGenE.py build options used for this process are taken from Warner's experiments [19] in order to produce properly working trees. The building process is performed by the experimental script `build_tree.sh`. Build options for each protocol are:

- HTTP: `-p 80 -xa -M 0`
- ENIP: `-p 44818 -x -M 0.5`
- ISO-TSAP: `-p 102 -x -M 0.5`

Each protocol has a specific port number ('`-p`') and threshold for clustering ('`-M`'). The '`-x`' option limits exported items to only a Python pickle file, the format used by the replay software. The '`-a`' option provides ASCII output.

For each experiment, a set of modified p-trees is created from this baseline in order to evaluate the emulator's ability to compensate for incomplete training data with proxy responses. Modification is done by `modify_tree.sh` which repeatedly calls modTree.py for each modification. Modtree.py removes a random non-root node in the specified p-tree and deletes any descendants of that node. It takes an argument as the number of nodes to remove in this manner. It is possible to delete the entire tree except the root. For this research, a single node is removed from each tree. Pilot testing reveals that a single removed node will cause the modified tree to fail its task in nearly every case. There are unlikely cases where the build process fails to consolidate similar messages and the modification removes the one that is not needed during a task. Removing more than one node from a p-tree with few branches, as in EtherNet/IP, is likely to delete most of the tree, reducing variability.

4.4.1.3 Synchronization algorithm.

The PLC synchronization algorithms in Section 3.5.5.3 are interchangeable in any emulator configuration. For each experiment, the following algorithms are evaluated:

- Latelock
- Templock
- Minimal Templock
- Triggerlock

These algorithms are chosen because they generate the best theoretical performance and are most likely to be deployed in actual implementations. The minimal version of templock is the normal templock algorithm with the minimal catch up heuristic incorporated. This affects the nature of initial synchronizations but does not affect connection openings or closings. A final "mode" is emulation with no proxy capabilities. This is identical to the foundational ScriptGenEreplay.py behavior and is used for comparison.

4.4.1.4 Replay options.

The ScriptGenEemulate.py function is a direct extension of Warner's Script-GenEreplay.py [19] and takes all of the same runtime options in addition to those dictating the proxy behavior. For this reason, the base replay options are again taken from Warner's experiments in order to provide a valid emulation configuration. The options for each protocol are:

- HTTP: `-i $IFACE -f -v --strict`
- ENIP: `-i $IFACE -v -d`
- ISO-TSAP: `-i $IFACE -f -v -d`

In all cases the emulator requires the name of the interface on which to project its service. Also the verbose switch ('`-v`') generates logs and debugging information. The '`-f`' flag keeps the emulator running forever regardless of the number of connections received. EtherNet/IP does not require this flag as a single connection is sufficient for the chosen task; the emulator dies when the connection closes. The '`-d`' option instructs the emulator to continue repeating the last message sent when a default error message is required. This allows the emulator to continue to respond after encountering errors rather than closing the connection early. HTTP does not require this option as each request is a separate connection; no further traffic is expected, even if an error message is sent. Finally, the '`--strict`' option disables backtracking for the same reason regarding HTTP. There is no need to search the tree when each request is a separate connection; session loops are impossible.

In addition to the above options, each protocol requires proxy configuration options as follows:

```
--target $PLC -m $sync_mode
```

The target is the PLC IP address for forwarding traffic. The synchronization mode is one of the PLC synchronization algorithms to be evaluated. The proxy options are excluded for experimental tasks where the proxy is turned off.

4.4.2 System parameters.

Figure 14 shows system parameters consist of PLC configurations and computing parameters. The computing parameters are derived from the host laptop. The laptop and PLC configurations are shown below:

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)

4.5 Performance Metrics

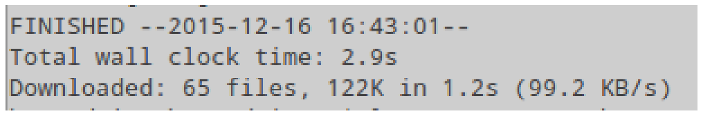
Three performance metrics are used to evaluate the emulator in this research. Because this research exclusively addresses the application layer, network and transport metrics like timing and `nmap` authenticity are considered out of scope. Each performance metric is based on application layer data as observed by standard tools or pcap inspection.

The first metric measures the fraction of client messages forwarded to the PLC by the emulator. This percentage metric is calculated by counting the application layer messages containing data in the proxy and client pcaps. Empty packets (e.g., TCP handshakes and acknowledgments) are not considered traffic for this metric.

It should be noted that counting messages does not guarantee a fraction strictly less than one. It is possible that the emulator could send a client message to the PLC more than once, producing a metric value greater than one. Therefore, this metric should technically be termed a "ratio." However, a practical interpretation of the metric and the average test results fit the fractional description.

The second metric determines the accuracy of the emulator from the perspective of standard industry tools (i.e., interrogators). The interrogator results are either PASS or FAIL. The tools chosen, `wget`, RSLinx, and STEP7, each have a specific success condition which signals the automation scripts to return a PASS result. The default is to FAIL. Detection of the success condition is also performed by the automation scripts. Failure to detect the success condition indicates errors due to incorrect emulator responses and results in a FAIL being returned.

The HTTP success condition is the number and size of the web page files downloaded and reported by `wget`. Initial testing reveals that a full download collected 65 files, each over a separate connection. The aggregate size of these files is 121KB or 122KB. An HTTP PASS is returned when `wget` reports 65 files and 121KB or 122KB file sizes. Any other output produces a FAIL. The successful output is printed in Figure 15.

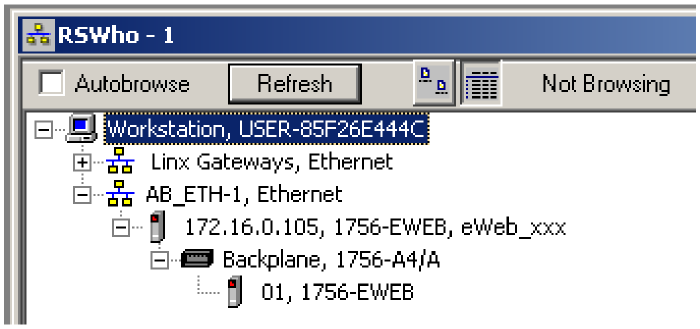


```
FINISHED --2015-12-16 16:43:01--
Total wall clock time: 2.9s
Downloaded: 65 files, 122K in 1.2s (99.2 KB/s)
```

Figure 15. Successful web page download by `wget`

RSLinx and STEP7 both require the use of GUIs to perform the EtherNet/IP and ISO-TSAP tasks respectively. These GUIs produce images which are located by SikuliX. In both cases, a PASS result requires the interrogator to successfully browse hardware modules for each PLC. Error messages, incomplete tasks, and failure to locate the correct modules each result in a FAIL. Successful modules for RSLinx and STEP7 are shown in Figure 16 and Figure 17 respectively.

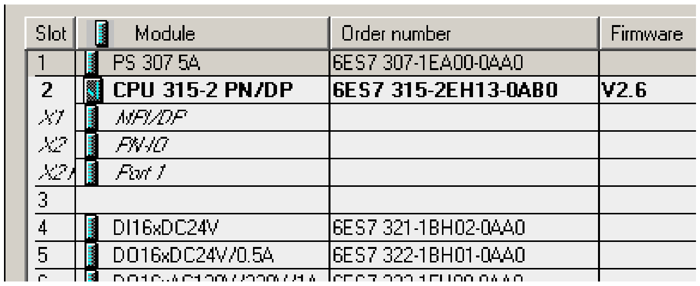
The final metric measures emulator traffic accuracy at the byte level by comparing differences in experimental and reference captures. The reference captures are generated during tasks where an unmodified tree is loaded into the emulator. This makes the proxy superfluous whether it is configured or not. Experimental captures are gen-


RSWho - 1

Autobrowse | Refresh | Not Browsing

```
Workstation, USER-85F26E444C
  Linx Gateways, Ethernet
  AB_ETH-1, Ethernet
    172.16.0.105, 1756-EWEB, eWeb_xxx
      Backplane, 1756-A4/A
        01, 1756-EWEB
```

Figure 16. Successful module browsing in RSLinx



Slot	Module	Order number	Firmware
1	PS 307 5A	6ES7 307-1EA00-0AA0	
2	**CPU 315-2 PN/DP**	**6ES7 315-2EH13-0AB0**	**V2.6**
X1	*MPI/DP*		
X2	*PN-IO*		
X2	*Port 1*		
3			
4	DI16xDC24V	6ES7 321-1BH02-0AA0	
5	DO16xDC24V/0.5A	6ES7 322-1BH01-0AA0	

Figure 17. Successful module browsing in STEP7

erated during tasks involving a modified tree. For each task a percentage difference in bytes is computed between the experimental capture and its corresponding reference capture. Differences in the number of messages are taken into account but differences in the number of connections are not. The difference percentages are divided into two groups: those with the proxy configured and those using no proxy. The latter produces behavior equivalent to the original ScriptGenEreplay.py emulator. A statistical difference between these groups indicates that the proxy has improved overall emulator accuracy when training data is incomplete.

4.6 Experimental Design

4.6.1 Overview.

Each experimental task is designed to determine how well the extended ScriptGenE emulator performs compared to the foundational ScriptGenE replay emulator as directed by the research questions in Section 4.1. The chosen tasks use the HTTP, EtherNet/IP, and ISO-TSAP protocols as these are the protocols used to test the original ScriptGenE framework. The experiments are fully automated and randomness is introduced in order to test the consistency of the emulator's flexibility and accuracy.

4.6.2 Introducing variability.

The starting point for this research, ScriptGenE, was designed and tested as a proof of concept framework [19]. The experiment for this research is designed to test extensions to ScriptGenE rather than retesting the baseline software. Therefore care is taken to prevent unexpected bugs or errors in ScriptGenE during testing. This is achieved by using reference pcaps to generate p-trees known to correctly complete each task. Experimental variability is intentionally added through modification of the baseline p-trees. Every experimental task is performed exactly as the reference captures were created so that emulator behavior can be isolated to the p-tree modifications.

Variability at the p-tree level is appropriate for SUT assessment. The extended emulator is designed to supplement p-tree deficiencies with proxied PLC responses. A p-tree is modified by removing a random, non-root node from the baseline p-tree and deleting the subtree rooted at the removed node. Node removal simulates the loss of the client message on the edge between the removed node and its parent. When the emulator reaches the parent context and searches for a client message, modified trees will lack that client message and provide no match. This allows the proxy function a chance to update the tree with new PLC responses.

Because the triggerlock synchronization algorithm is triggered by p-tree depth and the other algorithms are not, it is critical that some modifications occur below the trigger threshold. If all modifications occur above the threshold, the algorithms cannot be fairly compared. While random node selection does not guarantee this condition, manual inspection of each set of modifications reveals the random choices are acceptable.

4.6.3 Determining the number of modified protocol trees.

Removal of a single node per modification reflects realistic training data in a deployed honeypot. The assumption is that training data in the real world is thorough but not complete. Pilot testing reveals that the loss of a single node is sufficient to cause the p-tree to fail the experimental task for each protocol. In addition, EtherNet/IP p-trees are relatively small (fewer than 30 nodes) and linear. Removal of more than one random node yields a very reduced tree with high probability. Removing more than one node actually reduces the added variability as the number of removed nodes increases.

Removing only one node limits the number of modified trees to the size of the baseline tree for each protocol. EtherNet/IP has the smallest baseline tree with 22 nodes. HTTP has 75 and ISO-TSAP has 83. The Birthday Paradox dictates that the probability of removing the same node in more than one modified tree rises rapidly as the number of modifications increases. Testing the same modified tree more than once provides no new information and requires additional testing time. Therefore the number of modifications is maximized until the probability of collisions reaches 50%. For the 22 nodes in the EtherNet/IP tree, this number is six at a probability of 52%. Therefore six is chosen as the number of modifications to each baseline p-tree for each experiment. To increase the number of potential modifications and show consistency, the entire experiment is performed three times and all traffic data is aggregated.

4.7 Evaluation Techniques

All data analysis and evaluation is performed in the statistical package R [58]. Scripts for each protocol read the data log files and perform the calculations listed below. Standard statistics (e.g., mean, standard deviation, 95% confidence interval) are calculated for all data sets in addition to the tests described below.

Proxy forwarding rates are the ratio of data-bearing forwarded messages to client messages received. A one-sided Mann-Whitney-Wilcoxon test is used to determine if the rates are significantly less than 100%. This test is chosen over a t-test because a normal distribution is not assumed.

Pass rates for each protocol task are computed as the percentage ratio of PASS results to total task runs. This metric is computed for all tasks with and without the proxy. The mean and standard deviation are computed for each value across multiple experimental runs. A Mann-Whitney-Wilcoxon test is used to determine if there is a significant difference when the proxy is turned on.

The final metric measures byte-level accuracy of the emulator traffic. As with the metrics above, the Mann-Whitney-Wilcoxon test is used to determine if there is a significant difference between the variability in traffic when the proxy is used versus the original replay functionality.

Synchronization Algorithm Comparison. PLC synchronization algorithm performance is heavily dependent on the protocol used. The forwarding rate metric used to determine PLC load reduction is examined at the algorithm level to determine which algorithm reduced the load by the greatest amount. A Kruskal-Wallis test reveals if the performance of any algorithm deviates significantly from the others. Because the Kruskal-Wallis test does not identify which algorithm is different, further

Mann-Whitney-Wilcoxon tests are used to isolate it through pair-wise comparisons. As before, the data is not assumed to be normal.

4.8 Experimental Setup

4.8.1 Overview.

As shown in Figure 18, the experimental network setup includes two physical PLCs, an Allen-Bradley L61 and a Siemens S7-300, and a laptop hosting four VMs. These three machines are connected by a Cisco SG 100D-08 switch. The laptop VMs each have a bridged network interface to the physical interface of the laptop. Three of the VMs run client interrogators while the fourth, a Kali Linux VM, hosts the SUT, runs experiment coordination scripts, and captures all traffic data with `tshark`. The RSLinx and STEP7 interrogators each run on Windows XP VMs and `wget` runs on a Honeydrive Linux VM. SikuliX scripts are used to automate the RSLinx and STEP7 GUIs. These scripts and a `wget` task script are initiated remotely by a simple Python server running on the Kali Linux coordinator. The physical machine configurations are listed in Section 4.4.2.

4.8.2 Machine configurations.

All physical configuration information for the laptop and PLCs is discussed in Section 4.4.2. The primary VM is the Kali Linux 1.0 which hosts the SUT. Kali is derived from Debian Linux and is commonly used for penetration testing. All VM configuration and software information are listed in Table 1.

Both Windows XP VMs have nearly the same virtual hardware and performance capabilities. They differ primarily in the installed software packages. One carries the RSLogix suite, which includes RSLinx, while the other substitutes the STEP7 suite

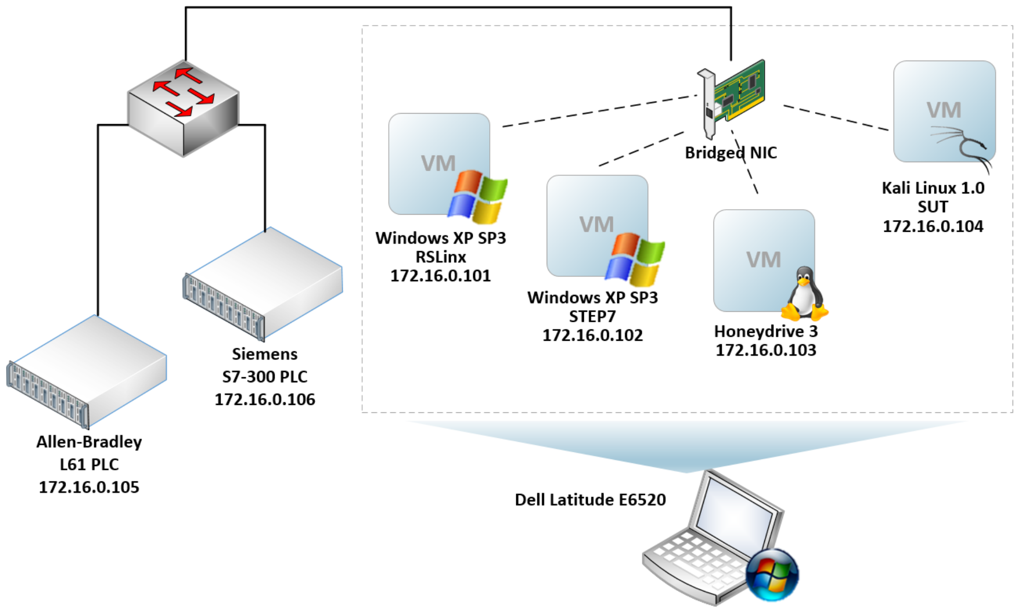


Figure 18. Experiment setup

Table 1. Kali Linux VM configuration

Kali 1.0		
1 processor core	Linux kernel 3.14.5	bash 4.2.37
2GB RAM	Python 2.7.3	tshark 1.10.2
30GB HD space	gcc 4.7.2	

for RSLogix. The Windows XP configurations for the RSLogix VM and STEP7 VM are listed in Table 2 and Table 3.

Table 2. RSLogix Windows XP VM configuration

Windows XP Service Pack 3		
2 processor cores	RSLogix V19.01.00	Python 2.7.2
2GB RAM	ControlFLASH	Java 1.8.0u51
60GB HD space	RSLinx Classic 2.59.02	SikuliX 1.1.0

Table 3. STEP7 Windows XP VM configuration

Windows XP Service Pack 3		
2 processor cores	STEP7 5.5	Python 2.7.2
4GB RAM	ControlFLASH	Java 1.7.0u25
60GB HD space	SikuliX 1.1.0	

The last VM is a third-party Linux distribution called Honeydrive 3.0. This Xubuntu variant is designed for honeypot deployment by housing a wide variety of pre-installed data collection and analysis tools. The VM acts as a `wget` interrogator for this experiment. All VM configuration and software information are listed in Table 4.

Table 4. Honeydrive Linux VM configuration

Honeydrive 3 (Royal Jelly)		
1 processor core	Linux kernel 3.2.0	bash 4.2.25
1GB RAM	Python 2.7.3	wget 1.13.4
80GB HD space		

4.8.3 Experimental scripts.

Experiment automation and data collection is controlled by a variety of custom scripts. These scripts perform the actual experiment and generate files to be analyzed in R [58]. The scripts include:

modify_tree.sh

- Randomly removes non-root nodes from a protocol tree and cleans up dangling fragments

master-exp.sh

- Coordinates all experimental tasks
- Manages network connectivity status

wget-exp.sh

- Runs the HTTP experimental task
- Uses RobotServer.py to control `wget` remotely

wget-test.sh

- Runs the experimental `wget` task
- Called by RobotClient.py on the remote host

enip-exp.sh

- Runs the EtherNet/IP experimental task
- Uses RobotServer.py to control RSLinx remotely

step7-exp.sh

- Runs the ISO-TSAP experimental task
- Uses RobotServer.py to control STEP7 remotely

RobotClient.py

- Controls RobotServer.py to run GUI tasks powered by SikuliX
- Sends one command and gets PASS or FAIL result

RobotServer.py

- Controlled by RobotClient.py to run GUI tasks powered by SikuliX
- Receives one command, executes the remote task, and returns PASS or FAIL result

diffPcapDir.py

- Utilizes Warner's diffPcaps.py [19] to collect difference data on combinations of baseline and experimental pcaps in a directory
- Difference data is exported to text files suitable for import into the R statistical package

4.8.4 Task automation.

While the scripts in Section 4.8.3 automate the coordination of the experiment and its command line components, several SikuliX scripts handle GUI automation of the RSLinx and STEP7 software. Using OpenCV and Tesseract, SikuliX is able to automate anything on a computer screen using image and text recognition along with user input emulation (e.g., mouse, keyboard) [24, 59]. SikuliX scripts can be integrated with other software due to its Java foundation. The scripts for this research use Jython, a Python equivalent built on Java.

The scripts used in this experiment are modified from those developed by Warner [19]. SikuliX scripts control the RSLinx drivers, STEP7 object status, and task conversations. Batch files restart the RSLinx and STEP7 servers and a bash script controls the `wget` task. All of these scripts are remotely called by RobotClient.py which receives instructions from RobotServer.py with the keyword 'RUN'. By category, the RobotServer tasks used in this experiment are:

Driver 'STOP_DRIVER', 'DEL_DRIVER', 'CONFIG_DRIVER'

RSLinx 'RESTART'

STEP7 'DEL_OBJECTS', 'CONFIG_NODE', 'RESTART_STEP7'

wget 'WGET'

Conversations 'RSWHO', 'STEP7BROWSE'

Driver tasks handle tasks related to device drivers and browsing in EtherNet/IP. *Conversation* tasks bundle several simple tasks together to perform a sequence of realistic interactions between the interrogator and a device. The remaining categories contain the simple tasks performed by each interrogator.

In addition to running the 'RSWHO' task, the EtherNet/IP experiment stops the RSLinx driver between experimental runs. This is to prevent extraneous traffic created by the auto browse feature. The 'RSWHO' task deletes and restarts this driver during the next run.

4.8.5 Configuring and running the experiment.

The full experiment consists of all tasks for all three protocols run in random order to mitigate unexpected sources of variability. Randomization is accomplished by creating a list of all experiment tasks and randomizing the list order with the bash tool `shuf`. As tasks are completed the corresponding line is removed from the list and added to a list of completed tasks. To eliminate bias from previous p-tree modifications, each experiment task is run on a new emulator process. This guarantees that previous client interactions will not affect the emulator's responses to future requests.

Coordination of the task list and experiment completion is controlled by master-exp.sh. This script calls the individual scripts for the protocol tasks: wget-exp.sh, enip-exp.sh, and step7-exp.sh. All four scripts have unique configuration files ('*-exp.conf') which hold relevant experimental and environmental details. The experiment scripts can be called from the command line with custom configurations. The following example runs the master experiment (ID = 1) with seven modified trees rather than six:

```
NUM_MODS=7 ./master-exp.sh 1
```

4.9 Methodology Summary

This chapter describes the experiment methodology for testing the accuracy and performance of the extended ScriptGenE emulator. The experiment performs tasks over three different protocols with an emulator using varying sets of training data. Each task consists of an emulator configuration with a modified p-tree, a PLC synchronization algorithm, and an automated protocol task. The emulator is able to proxy requests to a physical PLC it is trying to emulate, an Allen-Bradley L61 or Siemens S7-300. The tasks are performed by the interrogator programs `wget`, RSLinx, and STEP7.

The experiment is evaluated by the fraction of client requests forwarded to the PLC, the task success rate as determined by the PASS/FAIL results, and the byte-level differences between traffic captures from the experimental and reference emulators.

V. Results and Analysis

5.1 Overview

The experiment described in Chapter IV was conducted three times as specified. The HTTP and EtherNet/IP tasks completed successfully without errors. The ISO-TSAP tasks produced errors and failure modes which kept valid data from being collected. Further details on the STEP7 errors are provided in Section 5.1.1.

During analysis in R, all collected data is aggregated for computation of the forwarding rate and byte-level variability metrics (Section 5.2 and Section 5.4). Each experiment is treated as a separate data point for the success rate metric in Section 5.3. Finally, a comparison of the performances of each context algorithm is provided in Section 5.5.

As discussed in Section 4.7, non-parametric tests were chosen for all data due to lack of normality. Shapiro-Wilk tests verify this assumption. The largest p-value for any set of data collected is 10^{-11} indicating that each data set is significantly non-normal in distribution.

The forwarding rate, or fraction of client messages proxied, is less than 100% in all cases indicating load relief on the PLC. Though the proxy does not pass every task, it produces higher task success rates than default error messages. The byte level differences show that increased success rates do not imply a higher average level of accuracy. Finally, the context algorithm comparison confirms the proposition that algorithm performance is highly dependent on protocol. The triggerlock algorithm provides the worst performance while the minimal templock algorithm provides the best.

5.1.1 STEP7 Tasks.

Warner found ScriptGenEreplay is able to accurately replay the ISO-TSAP protocol and observed only PASS results from STEP7 tasks during his experiment [19]. The experiment for this research corrupted p-trees before emulation causing expected and unexpected task failures. The failures are a direct result of the emulator providing an inadequate response either by not responding at all or sending incorrect data.

Experiment attempts resulted in task failures producing critical STEP7 errors that disrupted the SikuliX automation scripts. Failure modes were diverse and unpredictable. Occasionally a failure caused no activity and the task ended abruptly. In other cases, the browse task stalled and STEP7 would stop responding for an arbitrary amount of time. In many cases a failure left the STEP7 GUI in an unexpected state that affected the results of the next task. Complete understanding of these errors requires in-depth knowledge of the STEP7 browsing process.

Manual handling of the errors was not straightforward in some cases. Closing the unresponsive program during hang ups either took a large amount of time or simply failed to close the GUI. Although the automation scripts are designed to be robust and handle many errors, they were not reliable enough to produce valid data. Full experiment automation in the presence of such unpredictable and troublesome errors requires a more extensive error handling technique and much longer experimental tasks to recover from hang ups.

5.2 Metric 1 - Message Forwarding Rate

The fraction of client messages proxied to the PLC, or the forwarding rate, is used to evaluate the PLC load reduction when ScriptGenEemulate is an intermediary emulator in a honeynet like Honeyd+. Any rate less than 100% shows a lightened load on the PLC. When no proxy is used (‘Off’ mode), there is no forwarded traffic and all forwarding rates are 0%. With the proxy turned on, the rates are all greater than 0% and less than 100%. This indicates that some, but not all, traffic was forwarded. The following sections address protocol-specific results.

5.2.1 HTTP forwarding rates.

Summary statistics for aggregated HTTP forwarding rates are shown in Table 5. The forwarding rate for each task takes one of three distinct values: 0%, 1.35%, or 2.7%. Each `wget` request establishes approximately 65 connections with the emulator. Each of these connections consists of one or two request packets. Most connections send a single request; however, incorrect server responses occasionally trigger a second client packet in the same connection. P-tree modification removes one node from the stateless tree which causes one of the request connections to fail. The 1.35% forwarding rate reflects one forwarded packet. 2.7% is double this value. A second packet is forwarded on failing runs because an extra request is received. The only other forwarding rate is 0% which occurs when unmodified p-trees are replayed. With an average rate of less then 2% as shown in Figure 19, the proxy conclusively reduces PLC loads ($p < 2.2 \cdot 10^{-16}$).

Table 5. HTTP proxy forwarding rates (%)

Mean	Std Dev	Min	Median	Max	95% CI Lower	95% CI Upper	*V*	*p*
1.53	0.86	0.00	1.35	2.70	0.00	3.23	0	$< 2.2 \cdot 10^{-16}$

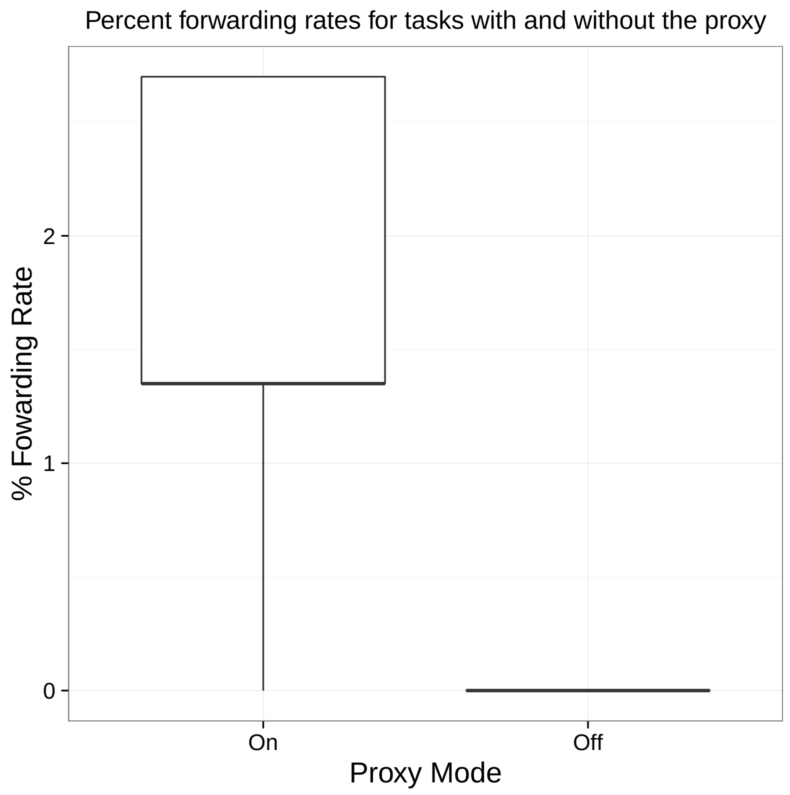


Figure 19. HTTP forwarding rates

5.2.2 EtherNet/IP forwarding rates.

Summary statistics for aggregated EtherNet/IP forwarding rates are shown in Table 6. Unlike HTTP, forwarding rates for EtherNet/IP vary widely as depicted in Figure 20. There are still 0% values reflecting the unmodified p-trees, but the average forwarding rate, approximately 54%, is much higher. The difference in the p-tree structure causes this discrepancy. While HTTP is a stateless protocol, EtherNet/IP produces deep, linear trees. PLC synchronization requires forwarding all of the received traffic in many cases, regardless of where modifications occur in the p-tree. The high standard deviation can also be attributed to tree structure and the diverse behavior required for different modifications. Even with unavoidable traffic, a successful PLC load reduction still occurs with an average rate less than 100% ($p < 2.2 \cdot 10^{-16}$).

Table 6. ENIP proxy forwarding rates (%)

Mean	Std Dev	Min	Median	Max	95% CI Lower	95% CI Upper	V	p
54.23	36.77	0.00	77.52	89.47	0.00	100.00	0	$< 2.2 \cdot 10^{-16}$

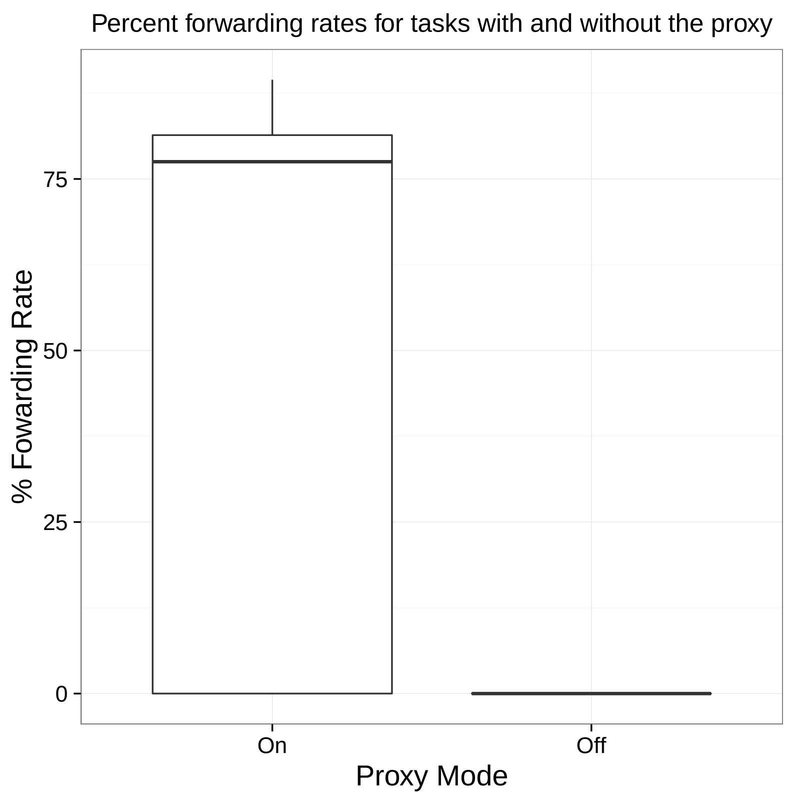


Figure 20. EtherNet/IP forwarding rates

5.3 Metric 2 - Task Success Rate

The task success rates for all experiments and protocols are shown in Table 7. The proxy provides higher success rates than default error messages for both protocols. The improvement is large for HTTP which swings from consistent failure to consistent success. EtherNet/IP shows a more modest improvement of 14%. Protocol-specific results are discussed below.

Table 7. Pass Rate Results (%)

Protocol	Proxy	Exp 1	Exp 2	Exp 3	Mean
HTTP	Off	0.00	0.00	0.00	0.00
	On	79.17	100.00	91.67	90.28
ENIP	Off	33.33	33.33	33.33	33.33
	On	48.00	45.83	48.94	47.59

5.3.1 HTTP success rates.

Summary statistics for HTTP task success rates are shown in Table 8. The `wget` task requires all of the web page files to be downloaded, each over a separate connection. When the emulator fails to provide the correct response for one of these connections, the task is not successful. This is why all tasks performed with the default error messages return a FAIL.

Table 8. HTTP task success rates (%)

Proxy	Mean	Std Dev	Min	Median	Max	95% CI		V	p
						Lower	Upper		
Off	0.00	0.00	0.00	0.00	0.00	0.00	0.00	6	0.125
On	90.28	10.49	79.17	91.67	100.00	45.16	100.00		

The proxy is able to fill the gap in the p-tree and pass tasks at higher rates. With a p-value of 0.125 the difference is not statistically significant, but this is attributed to the small number of data points. Inspection of the few failed tasks reveals failure always occurs on the same modified p-tree regardless of the proxy algorithm used. This suggests the modification to the tree was particularly troublesome for the emulator to handle, even with PLC responses.

5.3.2 EtherNet/IP success rates.

Summary statistics for EtherNet/IP task success rates are shown in Table 9. One third of all RSLinx tasks pass when the emulator replays a modified tree and does not use the proxy. This indicates there are some modifications to the p-tree which, while limiting its replay accuracy, do not prevent the success condition from being achieved. The correct modules are browsed without a full, accurate conversation.

Table 9. ENIP task success rates (%)

Proxy	**Mean**	**Std Dev**	**Min**	**Median**	**Max**	**95% CI**		***V***	***p***
						Lower	**Upper**		
Off	33.33	0.00	33.33	33.33	33.33	33.33	33.33	6	0.125
On	47.59	1.59	45.83	48.00	48.94	40.74	54.44		

Just as with HTTP, the tasks that failed while the proxy was in use were all confined to the same set of p-trees. The context algorithm is once again irrelevant. Given the baseline pass rate of 33%, the proxy improves the average success rate of RSLinx tasks by 14%. Again, more data is required for a statistically significant difference ($p = 0.125$).

5.4 Metric 3 - Byte-level Variability

Byte-level variability represents relative conversation variability where each data point is the percent difference between the successful baseline task pcap and a pcap collected while replaying a modified p-tree. For all protocols, the average variability is consistent whether the proxy is used or not. The default error messages derived from the most common response in the p-tree look similar to the correct response. This intentional design choice for ScriptGenEreplay is showcased by the results of this experiment. The following sections examine protocol-specific results.

5.4.1 HTTP variability.

Summary statistics for HTTP traffic differences are shown in Table 10. The variability when using the proxy is statistically equivalent to the non-proxy variability ($p = 0.0954$). While it is not conclusive from Figure 21 that the proxy improved overall accuracy of the emulator, there was very little room for improvement. A default error message sent during one of 65 connections results in very little variability for the conversation as a whole.

Table 10. HTTP traffic differences (%)

Proxy	Mean	Std Dev	Min	Median	Max	95% CI Lower	95% CI Upper	V	p
Off	11.80	22.08	0.14	1.59	75.66	0.00	58.86	729	0.0954
On	14.62	26.09	0.00	0.003	77.69	0.00	66.64		

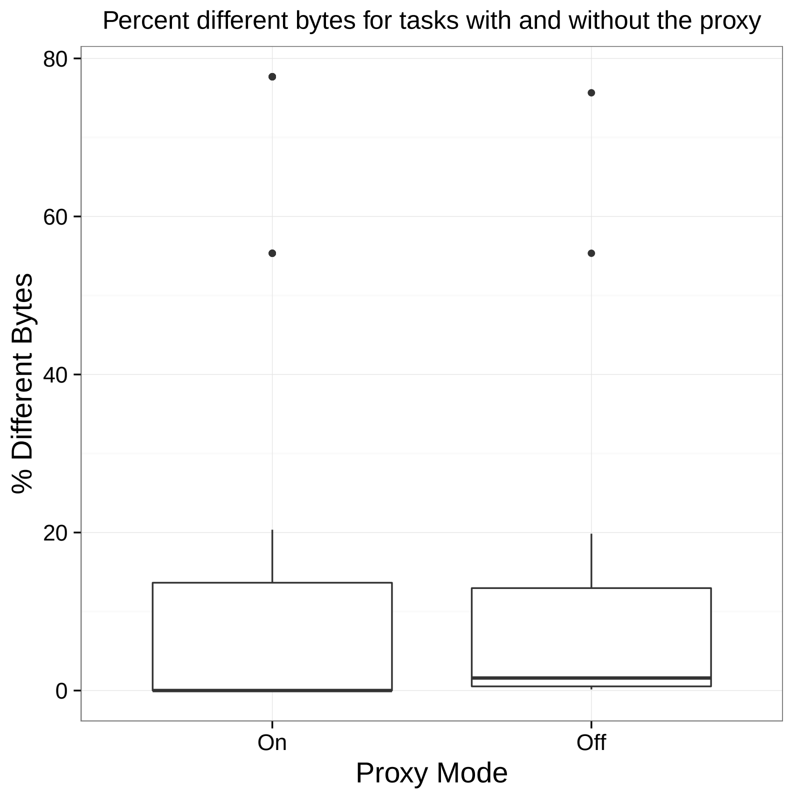


Figure 21. HTTP percentage byte-level differences

5.4.2 EtherNet/IP variability.

Summary statistics for ENIP traffic differences are shown in Table 11. Figure 22 shows the variability difference results for the EtherNet/IP tasks. Even though the task success rate discussed in Section 5.3.2 is higher when using the proxy, the average variability of the traffic is actually higher than when no proxy is used.

This may appear to be a contradiction with success rates showing improved accuracy while variability rates show decreased accuracy. Manual inspection of the data reveals that successful tasks had very low variability rates. The failed tasks produced inflated variability rates due to extra session loops. The session loops occur as RSLinx attempts to correct an error after receiving incorrect traffic from the emulator. These loops do not occur in a successful task conversation and contribute a large amount of variability to failed tasks. This additional variability is not a reflection of poor emulation accuracy. Statistically, the variability rates with and without the proxy are equivalent with a p-value of 0.7687.

Table 11. ENIP traffic differences (%)

Proxy	Mean	Std Dev	Min	Median	Max	95% CI Lower	95% CI Upper	V	p
Off	9.70	15.82	1.58	3.72	69.24	0.00	43.08	610	0.7687
On	12.03	12.83	0.00	8.61	44.76	0.00	37.61		

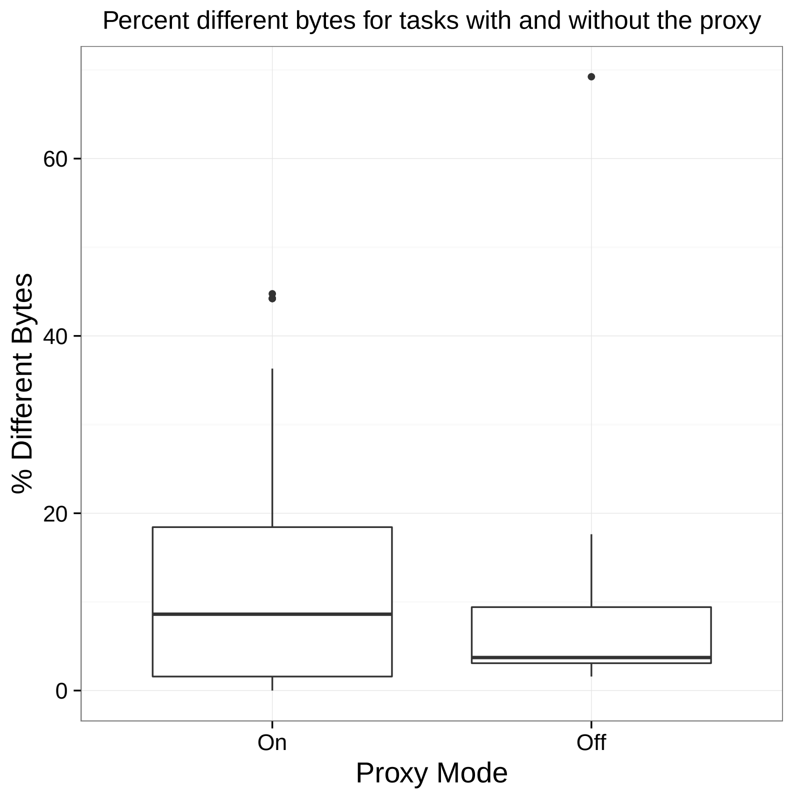


Figure 22. EtherNet/IP percentage byte-level differences

5.5 Context Algorithm Comparison

The following sections provide analysis of each context algorithm for distinct protocol tasks.

5.5.1 HTTP results.

Summary statistics for HTTP forwarding rates by algorithm are shown in Table 12. The forwarding rates for each algorithm during the HTTP tasks reveal very little about the strengths and weaknesses of each algorithm. Figure 23 illustrates each algorithm performing nearly identically. The group comparison test gives a p-value of 0.9947 ($\chi^2 = 0.0750$) indicating none of the algorithms deviates from the others. This is expected; the algorithms are intended to improve performance for long conversations with extensive synchronization processes. As a stateless protocol, HTTP does not fit these criteria. The chosen algorithm is irrelevant.

Table 12. HTTP proxy algorithm forwarding rates (%)

Algorithm	**Mean**	**Std Dev**	**Min**	**Median**	**Max**	**95% CI**	
						Lower	**Upper**
latelock	1.54	0.88	0.00	1.35	2.70	0.00	3.35
templock	1.54	0.88	0.00	1.35	2.70	0.00	3.35
min_templock	1.49	0.85	0.00	1.35	2.70	0.00	3.24
triggerlock	1.54	0.88	0.00	1.35	2.70	0.00	3.35

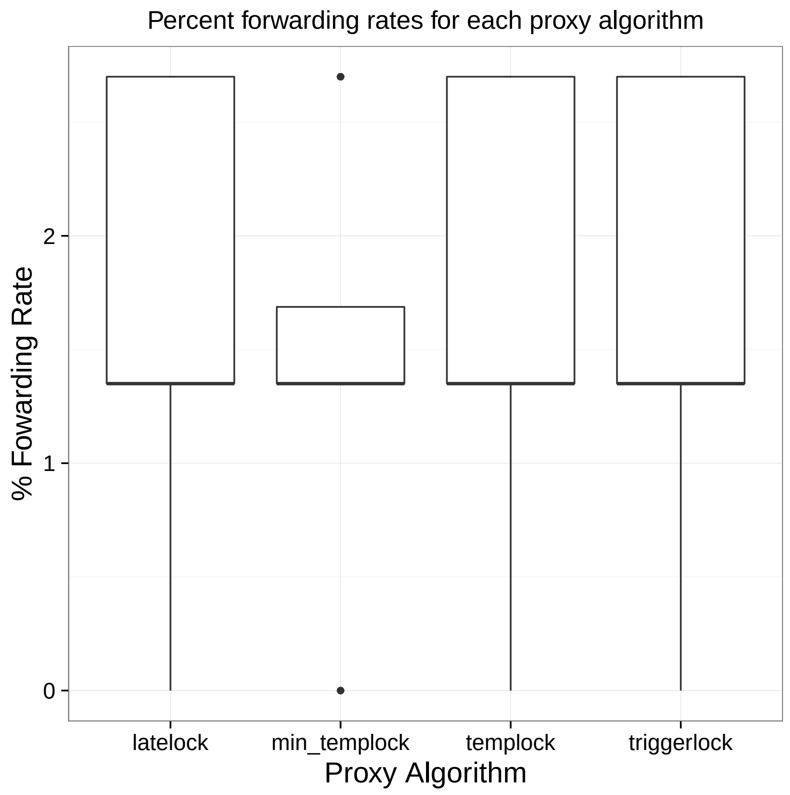


Figure 23. HTTP forwarding rates for each context algorithm

5.5.2 EtherNet/IP results.

Summary statistics for ENIP forwarding rates by algorithm are shown in Table 13. EtherNet/IP, a state-based protocol with a deep p-tree and frequent session looping, distinguishes the algorithms more than HTTP, as seen in Figure 24.

Table 13. ENIP proxy algorithm forwarding rates (%)

Algorithm	Mean	Std Dev	Min	Median	Max	95% CI	
						Lower	Upper
latelock	49.60	38.55	0.00	71.97	89.47	0.00	100.00
templock	43.36	38.83	0.00	66.66	84.21	0.00	100.00
min_templock	42.85	38.48	0.00	61.9	83.33	0.00	100.00
triggerlock	81.1	7.46	57.14	83.33	85.71	64.80	96.40

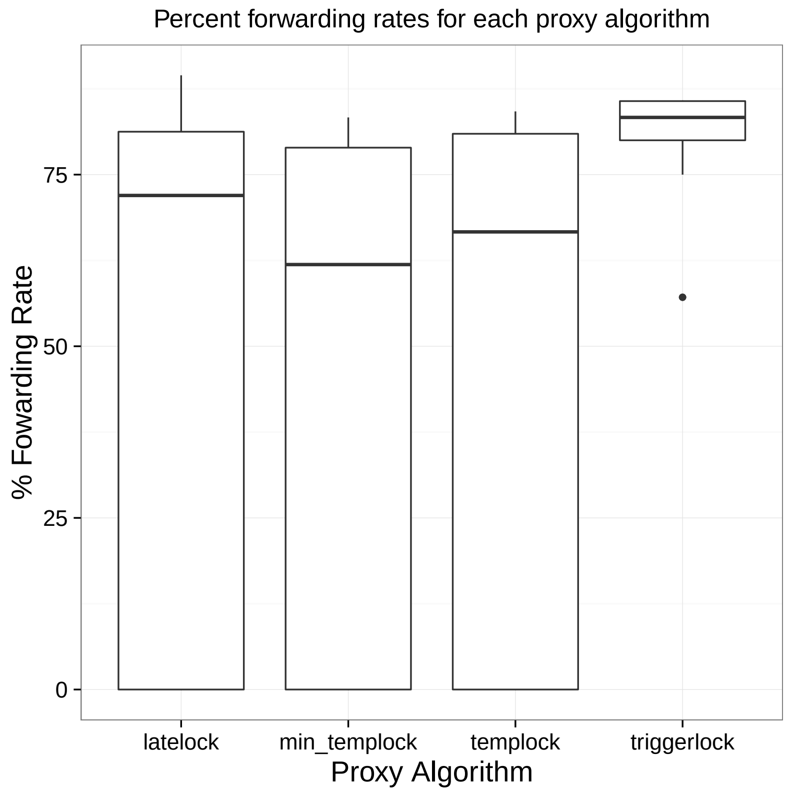


Figure 24. EtherNet/IP forwarding rates for each context algorithm

Triggerlock performs decidedly worse with a consistently higher forwarding rate. Kruskal-Wallis tests verify this with a negligible p-value ($\chi^2 = 30.011, p = 1.373 \cdot 10^{-6}$)

when comparing all algorithms and a p-value of 0.5634 ($\chi^2 = 1.1474$) when triggerlock is left out. Further evidence is provided by a Mann-Whitney-Wilcoxon test which reveals that triggerlock rates are significantly higher than the next highest algorithm, latelock ($W = 608.5, p = 0.0001751$). The poor performance of triggerlock can be explained by the session loops which cycle through the p-tree at heights above and below the trigger threshold. This causes the proxy to synchronize multiple times unnecessarily, resending a single client message more than once.

Although the performances of the latelock, minimal templock, and pure templock algorithms are similar, the differences in their averages can be explained. Templock, an optimization of latelock, barely outperforms latelock because it can close a proxy connection once it is no longer needed. The minimal version of templock, a further optimization, performs slightly better yet by synchronizing only part of the conversation context.

VI. Conclusions

6.1 Introduction

This chapter presents a summary of the research conclusions, impact, and future work. Section 6.2 gives conclusions from the experiment results. Section 6.3 discusses the impact of the research. Section 6.4 presents several ideas and recommendations for future work and related research.

6.2 Research Conclusions

This research successfully provides an enhanced PLC emulator based on the application layer replay framework found in ScriptGenE. The outcome of each research goal from Section 4.1 is discussed below. Each goal is satisfied successfully as shown through the conducted experiments.

6.2.1 Performance.

In order to reduce bottlenecks in distributed honeynets using a single back-end PLC, the emulator is required to handle a significant portion of incoming client traffic without using its proxy. All experiments verify the ability of ScriptGenEemulate to reduce the PLC load. For the HTTP protocol, the average load reduction is a dramatic 98%. The EtherNet/IP protocol shows an average load reduction of 46%.

6.2.2 Authenticity.

Industry standard tools are used to interrogate the emulator with specified tasks. Emulator authenticity is evaluated in two ways. First, the results of the tasks are returned PASS/FAIL. Experiment results for modified p-trees show that the enhanced emulator is able to produce higher PASS rates than the original ScriptGenEreplay which relies on default error messages. These results are consistent but lack enough data to show statistical significance.

The second accuracy evaluation technique consists of byte-level variability comparisons between the responses of ScriptGenEemulate and ScriptGenEreplay when replaying modified p-trees. The data shows that average variability is statistically equivalent in all cases. The EtherNet/IP task initiated by RSLinx elicits much higher levels of variability when the proxy is on and when the emulator fails to respond correctly. The high variability is attributed to the extra traffic generated by the interrogator not the degree of inaccuracy in the emulator.

6.2.3 Context Maintenance.

The results of experimental HTTP tasks provide no conclusive differences in the performance of the various context algorithms. The EtherNet/IP tasks reveal that triggerlock sends significantly more traffic to the PLC than other algorithms as a result of session looping and the trigger threshold placement. Performance of the other three algorithms (latelock, templock, and minimal templock) is consistent. The minimal templock algorithm average forwarding rate reduces the PLC load by the greatest amount. However, if minimal synchronization is insufficient and the full context is required, templock is the recommended algorithm.

6.3 Significance of Research

6.3.1 Contributions.

The primary contributions of this research consist of ScriptGenE framework enhancements. ScriptGenEreplay.py is extended into ScriptGenEemulate.py, an enhanced application layer emulator. Enhancements include a restructured, object-oriented design, for increased maintainability and extensibility, and multi-threading, to allow the emulator to service multiple client connections concurrently.

The most significant enhancement is the addition of a proxy mechanism which replaces default error messages. The proxy is able to forward unrecognized messages from a client to a live PLC and respond to the client with the PLC response. This improves the accuracy of the emulator and allows it to handle situations where protocol training data is insufficient.

The final contribution is the dynamic update function which incorporates PLC responses into the protocol tree. Updating the p-tree allows the emulator to replay the new response directly and avoid further proxy usage. Theoretically the proxy should be used less over time leading to gradual PLC load reductions.

6.3.2 Applications.

The general applications of an application layer ICS emulator are very diverse and include network fuzzing, security training, research projects, and standalone honeypots. More specific applications involve incorporating the emulator into a hybrid network sensor or honeynet. This application allows the emulator to help detect intrusions or collect malware and other attack information. As discussed in Section 3.2, the design of ScriptGenEemulate is specifically intended for use in Honeyd+, a production-level ICS honeypot framework [20].

6.4 Future Work

6.4.1 Overview of recommendations.

Warner lists a variety of recommendations for future work on the ScriptGenE framework [19]. This research addresses some of those recommendations through extensions to the ScriptGenEreplay functionality. In the process, new areas of future work are created. This section recommends areas of future work on the ScriptGenEemulate application layer emulator. The recommendations fall into three categories: testing, emulator improvement, and honeynet integration.

6.4.2 Testing.

Further testing of ScriptGenEemulate.py is required to show it can be deployed in a production environment. Testing with additional protocols, tasks, and PLC configurations can show the emulator is able to handle diverse network conditions.

Further testing is also recommended for profiling the strengths and weaknesses of each proxy context maintenance algorithm. Each algorithm is theoretically ideal for appropriate protocols. The algorithms can be tested with a wide range of p-tree types to verify the theoretical predictions and further refine the process of choosing the best algorithm for real-world deployments.

The enhanced emulator also contains features that have not been tested. Dynamic updating can be tested to determine if PLC load is reduced over the course of several consecutive tasks as the p-tree is gradually expanded. In addition, it is not known whether the subsequent tasks in such an experiment will be consistently accurate or if single message updates to a generic protocol tree will reduce its accuracy.

Another untested feature of ScriptGenEemulate is its ability to handle multiple simultaneous connections. Although pilot testing proves that the emulator can handle multiple connections, the performance limitations are not known. Further testing can determine how many simultaneous connections ScriptGenEemulate can handle on various computing platforms and if there are any adverse affects on emulation quality as the number of connections increases.

6.4.3 Enhancing ScriptGenEemulate algorithms.

The current ScriptGenEemulate implementation can be enhanced in a variety of ways described in the following sections.

6.4.3.1 Global link recognition.

As discussed in Section 3.5.5.4, the emulator is currently unable to automatically recognize globally-linked fields across a whole conversation. Manual configuration is necessary for ScriptGenEemulate to locate global link fields. Future software iterations can add global link recognition during emulation or during the p-tree building process. In either case, an algorithm similar to the intra-protocol link detection is required to find the global link and determine if it changes value through session establishment.

6.4.3.2 Robust dynamic updates.

Dynamic updating as described in Section 3.5.5.2 can be improved through iterative updating. To make dynamically-added nodes generic like the rest of the tree, unknown transitions can be proxied multiple times. The results could be consolidated as in ScriptGenE.py. Proxy repetition can occur all at once in rapid succession or the new node could be assigned probationary status as it waits on future proxy responses of that type.

6.4.3.3 New context maintenance algorithms.

Although several algorithms are tested in this research, there are a many other ways client context can be transferred to the PLC. One potential option is a *lagstep* algorithm which performs similarly to triggerlock by establishing a proxy connection based on the depth of the current context in the p-tree. However, rather than synchronizing the whole conversation at once and creating delays visible to the client, lagstep could begin synchronizing incrementally between client messages. This would reduce the synchronization delay and maintain a low forwarding rate.

6.4.4 Honeynet integration.

With the ability to accept multiple connections, ScriptGenEemulate is ready to be integrated into existing honeynet configurations. The ScriptGenE framework operates exclusively at the application layer, so adding to a honeypot framework capable of handling the network and transport layers is recommended. As discussed in Section 6.3.2, Honeyd+ is the framework for which ScriptGenEemulate was designed. Honeyd or any other low-level honeypot framework may benefit from the abilities of ScriptGenEemulate.

6.5 Chapter Summary

This chapter presents the conclusions and impact of the ScriptGenEemulate research. To conclude, several recommendations for future work are provided which build upon and extend the application layer emulation techniques developed in this research.

Bibliography

1. 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*.

2. 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.

3. Ruben Santamarta. Project Basecamp – attacking ControlLogix. In *5th SCADA Security Scientific Symposium*, Miami Beach, FL, USA, January 2012. Digital Bond. Retrieved 23 June, 2015 from *http://reversemode.com/downloads/logix_report_basecamp.pdf*.

4. 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*.

5. 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*.

6. Todd Vollmer, Milos Manic, and Ondrej Linda. Autonomic intelligent cyber-sensor to support industrial control network awareness. *Industrial Informatics, IEEE Transactions on*, 10(2):1647–1658, May 2014.

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

8. Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection.* Pearson Education, Boston, MA, USA, first edition, 2007.

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

10. 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.

11. 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.

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

13. Todd Vollmer and Milos Manic. Cyber-physical system security with deceptive virtual hosts for industrial control networks. *Industrial Informatics, IEEE Transactions on*, 10(2):1337–1347, May 2014.

14. Christopher Hecker and Brian Hay. Automated honeynet deployment for dynamic network environment. In *System Sciences, 46th Hawaii International Conference on*, HICSS'13, pages 4880–4889. IEEE, 2013.

15. Jeff Hieb. Anomaly based intrusion detection for network monitoring using a dynamic honeypot. Master's thesis, University of Louisville, Louisville, KY, USA, December 2004. Retrieved 26 October, 2014 from *http://digital.library.louisville.edu/utils/getfile/collection/etd/id/516/.../517.pdf*.

16. Iyad Kuwatly, Malek Sraj, Zaid Al Masri, and Hassan Artail. A dynamic honeypot design for intrusion detection. In *Pervasive Services, IEEE/ACS International Conference on*, pages 95–104. IEEE, July 2004.

17. Xiangdong Li and Li Chen. A survey on methods of automatic protocol reverse engineering. In *Computational Intelligence and Security, Seventh International Conference on*, CIS'11, pages 685–689. IEEE, 2011.

18. Corrado Leita, Ken Mermoud, and Marc Dacier. ScriptGen: an automated script generation tool for Honeyd. In *Computer Security Applications Conference, 21st Annual*, pages 12 pp.–214. IEEE, December 2005.

19. 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*.

20. 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*.

21. 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.

22. 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*.

23. 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.

24. Raimund Hocke. SikuliX powered by RaiMan, 2015. Retrieved 9 March, 2015 from *http://www.sikulix.com*.

25. 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*.

26. Éireann P. Leverett. Quantitatively assessing and visualising industrial system attack surfaces. Master's thesis, University of Cambridge, Darwin College, Cambridge, UK, June 2011. Retrieved 8 February, 2015 from *http://www.cl.cam.ac.uk/~fms27/papers/2011-Leverett-industrial.pdf*.

27. Kelly J. Higgins. 'Project SHINE illuminates sad state of SCADA/ICS security on the net. Dark Reading blog entry, 2013. Retrieved 23 June, 2015 from *http://www.darkreading.com/vulnerabilities---threats/project-shine-illuminates-sad-state-of-scada-ics-security-on-the-net/d/d-id/1140691*.

28. Paul M. Williams. Distinguishing Internet-facing ICS devices using PLC programming information. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, June 2014 (ADA602989).

29. Kyle Wilhoit. Who's really attacking your ICS equipment? Technical report, Trend Micro, Inc., 2013. Retrieved 23 June, 2015 from *http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-whos-really-attacking-your-ics-equipment.pdf*.

30. Kyle Wilhoit. The SCADA that didn't cry wolf. Technical report, Trend Micro, Inc., 2013. Retrieved 23 June, 2015 from *http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-scada-that-didnt-cry-wolf.pdf*.

31. Roland C. Bodenheim. Impact of the Shodan computer search engine on Internet-facing Industrial Control System devices. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2014 (ADA601219).

32. Craig G. Rieger, David I. Gertman, and Miles A. McQueen. Resilient control systems: Next generation design research. In *Human System Interactions, 2nd Conference on*, HSI '09, pages 632–636, May 2009.

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

34. 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*.

35. 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.

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

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

38. Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The nepenthes platform: An efficient approach to collect malware. In *Recent Advances in Intrusion Detection*, pages 165–184. Springer, 2006.

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

40. 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.

41. 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.

42. 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*.

43. Robin G. Berthier. Honeybrid: Hybrid honeypot framework, 2013. Retrieved 24 August, 2015 from *http://honeybrid.sourceforge.net*.

44. Lukas Rist, Johnny Vestergaard, Daniel Haslinger, and Andrea Pasquale. Conpot, 2013. Retrieved 27 August, 2015 from *http://conpot.org/*.

45. Digital Bond. SCADA honeynet, 2006. Retrieved 27 August, 2015 from *http://www.digitalbond.com/tools/scada-honeynet/*.

46. Daniel Buza, Ferenc Juhasz, Gyorgy Miru, Mark Felegyhazi, and Tamas Holczer. CryPLH: Protecting smart energy systems from targeted attacks with a PLC honeypot. In *Proceedings of Smart Grid Security*, pages 181–192, February 2014. Retrieved 23 June, 2015 from *http://crysys.hu/publications/files/BuzaJMFH2014smartgridsec.pdf*.

47. Robert M. Jaromin. Emulation of industrial control field device protocols. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2013.

48. Kazuya Kishimoto, Kenji Ohira, Yukiko Yamaguchi, Hirofumi Yamaki, and Hiroki Takakura. An adaptive honeypot system to capture IPv6 address scans. In *Cyber Security, International Conference on*, CyberSecurity'12, pages 165–172. IEEE, December 2012.

49. Christopher Hecker, Kara L. Nance, and Brian Hay. Dynamic honeypot construction. In *Information Systems Security Education, Proceedings of the 10th Colloquium for*, Adelphi, MD, USA, June 2006. University of Maryland.

50. 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*.

51. 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*.

52. Deanna R. Fink. Toward automating web protocol configuration for a programmable logic controller emulator. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, June 2014.

53. 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.

54. 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*.

55. 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.

56. Mariano Graziano, Corrado Leita, and Davide Balzarotti. Towards network containment in malware analysis systems. In *Computer Security Applications Conference, Proceedings of the 28th Annual*, pages 339–348. ACM, 2012.

57. Christian Martin Fuchs and Martin Brunner. Towards next generation malware collection and analysis. *Advances in Security, International Journal on*, 6(1 and 2):32–48, 2013. Retrieved 31 August, 2015 from *http://www.thinkmind.org/download.php?articleid=sec_v6_n12_2013_3*.

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

59. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: Using GUI screenshots for search and automation. In *User Interface Software and Technology, Proceedings of the 22nd Annual ACM Symposium on*, UIST '09, pages 183–192, New York, NY, USA, 2009. ACM.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 24–03–2016	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2014 — Mar 2016

4. TITLE AND SUBTITLE Dynamic Honeypot Configuration for Programmable Logic Controller Emulation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Girtz, Kyle A., Mr., Civ	**5d. PROJECT NUMBER** 16G264
	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-16-M-253
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security ICS-CERT POC: Neil Hershfield, DHS ICS-CERT Technical Lead ATTN: NPPD/CS&C/NCSD/US-CERT Mailstop: 0635, 245 Murray Lane, SW, Bldg 410, Washington, DC 20528 Email: ics-cert@dhs.gov phone: 1-877-776-7585	**10. SPONSOR/MONITOR'S ACRONYM(S)** DHS ICS CERT
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

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

14. ABSTRACT

This research develops an enhanced, application layer emulator capable of alleviating honeynet scalability and honeypot inauthenticity limitations. The proposed emulator combines protocol-agnostic replay with dynamic updating via a proxy. The result is a software tool which can be readily integrated into existing honeypot frameworks for improved performance. The proposed emulator is evaluated on traffic reduction on the back-end proxy device, application layer task accuracy, and byte-level traffic accuracy. Experiments show the emulator is able to successfully reduce the load on the proxy device by up to 98% for some protocols. The emulator also provides equal or greater accuracy over a design which does not use a proxy. At the byte level, traffic variation is statistically equivalent while task success rates increase by 14% to 90% depending on the protocol. Finally, of the proposed proxy synchronization algorithms, templock and its minimal variant are found to provide the best overall performance.

15. SUBJECT TERMS

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

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	123	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636 x7979 Barry.Mullins@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18