

3-26-2015

Automatic Configuration of Programmable Logic Controller Emulators

Phillip C. Warner

Follow this and additional works at: <https://scholar.afit.edu/etd>

Part of the [Computer Engineering Commons](#)

Recommended Citation

Warner, Phillip C., "Automatic Configuration of Programmable Logic Controller Emulators" (2015). *Theses and Dissertations*. 67.
<https://scholar.afit.edu/etd/67>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.



**AUTOMATIC CONFIGURATION
OF PROGRAMMABLE LOGIC
CONTROLLER EMULATORS**

THESIS

Phillip C. Warner, Captain, USAF
AFIT-ENG-MS-15-M-024

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

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

AFIT-ENG-MS-15-M-024

AUTOMATIC CONFIGURATION OF PROGRAMMABLE LOGIC
CONTROLLER EMULATORS

THESIS

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

Phillip C. Warner, B.S.C.E
Captain, USAF

March 2015

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-15-M-024

AUTOMATIC CONFIGURATION OF PROGRAMMABLE LOGIC
CONTROLLER EMULATORS

THESIS

Phillip C. Warner, B.S.C.E
Captain, USAF

Committee Membership:

Barry E. Mullins, PhD (Chairman)

LTC Mason J. Rice, PhD (Member)

Juan Lopez Jr. (Member)

Abstract

Programmable logic controllers (PLCs), which are used to control much of the world's critical infrastructures, are highly vulnerable and exposed to the Internet. Many efforts have been undertaken to develop decoys, or honeypots, of these devices in order to characterize, attribute, or prevent attacks against Industrial Control Systems (ICS) networks. Unfortunately, since ICS devices typically are proprietary and unique, one emulation solution for a particular vendor's model will not likely work on other devices. Many previous efforts have manually developed ICS honeypots, but it is a very time intensive process.

Thus, a scalable solution is needed in order to automatically configure PLC emulators. The ScriptGenE Framework presented in this thesis leverages several techniques used in reverse engineering protocols in order to automatically configure PLC emulators using network traces. The accuracy, flexibility, and efficiency of the ScriptGenE Framework is tested in three fully automated experiments.

Network traffic is generated by interrogating two PLCs with standard clients. An Allen-Bradley ControlLogix L61 PLC is interrogated with `wget` and `RSLink`, and a Siemens S7-300 is interrogated using `STEP7`. Reference traffic is captured and used to build protocol state machines, and then the resultant emulators' experimental traffic is captured. Variability differences between the reference and experimental groups is analyzed to assess the accuracy of the emulators.

Results from the experiments show that ScriptGenE can accurately emulate a PLC's webserver with only one input trace. Additionally, only five input EtherNet/IP traces are required to create an emulator that is identified by `RSLink` as a PLC with modules. A minimum of two input traces are required to create a Siemens PLC

emulator that can be browsed by STEP7. The web and EtherNet/IP experiments resulted in zero incorrect bytes, and the experiments with STEP7 yielded at most one incorrect byte per TCP connection. Additionally, the emulators produce traffic that differs in variability from the reference capture group by less than 0.018% with 95% confidence.

Overall, this research provides numerous contributions including the first successful automatically configured application layer honeypot for EtherNet/IP. ScriptGenE requires less input traces than previous works. Additionally, a novel *backtracking* algorithm is implemented that handles unknown transitions and allows for looping in ICS polling sessions.

To my family.

For those that gave me life, and taught me how to live...

For those that provided encouragement, and continue to give...

For those who showed me wisdom, and how to choose what is best...

and for those who remind me, when it is time to rest.

Thank you, my family, for making every day a joy!

Acknowledgements

I would like to thank my advisor, Dr. Barry Mullins, for providing my first formal introduction to network protocols and all of the wondrous things they can do. Thank you for your passionate teaching, advice, and mentorship.

I would also like to thank the other members in my committee, LTC Mason Rice and Mr. Juan Lopez, for sharing their vision, expertise, and time.

Next, many thanks goes to Mr. Stephen Dunlap for helping me learn to communicate with the PLCs and for providing ideas for designing the experiments.

Finally, I would like to thank Lt Zachary Zeitlin for providing a peer review.

Phillip C. Warner

Table of Contents

	Page
Abstract	iv
Dedication	vi
Acknowledgements	vii
List of Figures	xi
List of Tables	xv
List of Acronyms	xvii
List of Algorithms	xxi
I. Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Research Goals	3
1.4 Approach	4
1.4.1 Protocol state machine generation	5
1.4.2 Protocol replay	5
1.4.3 Experimentation	6
1.5 Assumptions and Limitations	7
1.5.1 Limitations with network trace-based approaches	7
1.5.2 Network protocols involved	7
1.5.3 Single connection server	8
1.5.4 Limited set of tasks and traces	8
1.5.5 Limited configuration setup	8
1.5.6 Timing not considered	9
1.5.7 Ignored or invalid responses	9
1.6 Thesis Overview	9
II. Background and Related Research	10
2.1 Overview	10
2.2 Background	10
2.2.1 Industrial Control Systems	10
2.2.2 Allen-Bradley PLCs	12
2.2.3 Siemens PLCs	13
2.2.4 Industrial Control Systems (in)security	13
2.2.5 Networking and protocols	19
2.2.6 Honeypots	25

	Page
2.2.7 Common algorithms used in reversing protocols	31
2.3 Related Research	43
2.3.1 Manually configured ICS honeypots	43
2.3.2 Dynamic honeynets	45
2.3.3 Automatically configured emulators using protocol knowledge	48
2.3.4 Reverse engineering protocols	50
2.3.5 Putting it all together: Advanced hybrid honeypots using smart replay	56
2.4 Chapter Summary	57
III. Framework Design	59
3.1 Overview	59
3.2 Design Parameters	59
3.3 Technique Suitability Evaluation	60
3.4 Protocol Informatics In-depth	62
3.5 ScriptGen In-depth	63
3.6 The ScriptGenE Framework	67
3.6.1 Framework overview	67
3.6.2 ScriptGenE.py	70
3.6.3 ScriptGenEreplay.py	97
3.7 Design Summary	107
IV. Research Methodology	108
4.1 Goals	108
4.2 Approach	109
4.3 System Boundaries	109
4.4 Services and Outcomes	110
4.5 Parameters and Factors	112
4.5.1 Workload parameters	112
4.5.2 System parameters	118
4.6 Performance Metrics	119
4.7 Experimental Design	121
4.7.1 Overview	121
4.7.2 Introducing variability	121
4.7.3 Determining the number of required captures	122
4.8 Evaluation Technique	123
4.9 Experimental Setup	128
4.9.1 Overview of experimental setup	128
4.9.2 Machine configurations	129
4.9.3 Experimental scripts	130
4.9.4 Graphical User Interface automation	132

	Page
4.9.5 Configuring and running experiments	134
4.10 Methodology Summary	135
V. Results and Analysis	137
5.1 High-level Summary	137
5.2 Experiment #1 - HTTP	138
5.2.1 Build efficiency and qualities	138
5.2.2 Successful task completions	140
5.2.3 Byte-level variability	141
5.3 Experiment #2 - EtherNet/IP	144
5.3.1 Build efficiency and qualities	144
5.3.2 Successful task completions	147
5.3.3 Byte-level variability	149
5.4 Experiment #3 - ISO-TSAP	153
5.4.1 Build efficiency and qualities	153
5.4.2 Successful task completions	155
5.4.3 Byte-level variability	155
VI. Conclusions	159
6.1 Introduction	159
6.2 Research Conclusions	159
6.2.1 Accuracy	159
6.2.2 Flexibility	160
6.2.3 Efficiency	160
6.3 Significance of Research	160
6.3.1 Contributions	160
6.3.2 Applications	161
6.4 Future Work	161
6.4.1 Overview of recommendations	161
6.4.2 Testing	162
6.4.3 Enhancing current ScriptGenE algorithms	162
6.4.4 Hybrid network sensor integration	164
6.4.5 Miscellaneous	165
6.5 Chapter Summary	165
Bibliography	166

List of Figures

Figure	Page
1	ICS block diagram [SFS11] 11
2	General SCADA layout [SFS11] 12
3	NIST recommended ICS network configuration [SFS11] 17
4	Internet Protocol stack [KR13] 19
5	TCP and UDP segment format [KR13] 20
6	Establishing a TCP connection [KR13] 22
7	Closing a TCP connection [KR13] 22
8	HTTP GET request from Wireshark 23
9	HTTP reponse from Wireshark (data not shown) 23
10	CIP object model [Bro01] 25
11	Example tree generated using UPGMA [Edw13] 33
12	Example sequence alignment for two strings [KT06] 36
13	Example Needleman-Wunsch solution matrix [Bed04a] 39
14	Alignment obtained from solution matrix [Bed04a] 39
15	Two-coin Hidden Markov Model [Rab89] 41
16	Example Weighted Interval Schedule [KT06] 42
17	Message cluster output from PI 64
18	Sessions with different message order are not equivalent 65
19	Link consolidation [LDM06] 67
20	ScriptGenE Framework Overview 68
21	ScriptGenE.py Usage 73
22	Example ScriptGenE.py configuration file 74

Figure		Page
23	tcp_connection object life cycle	74
24	Initial output while parsing Pcap file	76
25	Server messages from Pcap file	76
26	Conversation summary	77
27	Output from initial consolidation	79
28	Example RA output - HTML	81
29	Example RA output - EtherNet/IP	83
30	Regular expression for example EtherNet/IP cluster	84
31	Regular expression for example HTTP cluster	84
32	ScriptGenE final summary	87
33	Unfiltered trivial link	90
34	Unfiltered link proposals	90
35	Filtered link proposals	90
36	Cluster link summary	95
37	A simple initial protocol tree	96
38	A simple final protocol tree	97
39	Viewing a GEXF file in Firefox	98
40	ScriptGenEreplay.py Usage	101
41	ScriptGenE Framework	111
42	Calculating the number of different and identical files between a pair of directories	124
43	A successful emulation session in RSLinux	125
44	A successful ISO-TSAP emulation session shows these modules in STEP7	125

Figure		Page
45	Number of different bytes per server message as compared to other reference captures	127
46	Experimental Setup	129
47	Example RobotServer session controlled by netcat	133
48	RSWho SikuliX script snippet	134
49	Number of bytes in HTML directories for each test group	140
50	Number of potentially incorrect bytes per client message found in experimental traces	141
51	Number of potentially incorrect bytes per server message found in experimental traces	142
52	Percent different bytes between Pcap files compared to reference Pcaps - One Pcap per HTTP emulator profile	143
53	Percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per HTTP emulator profile	144
54	RA output showing a problematic fixed field	148
55	EtherNet/IP header [Dun13]	148
56	Percent different bytes between Pcap files compared to reference Pcaps - Three Pcaps per EtherNet/IP emulator profile	150
57	Percent different bytes between Pcap files compared to reference Pcaps - Four Pcaps per EtherNet/IP emulator profile	151
58	Percent different bytes between Pcap files compared to reference Pcaps - Five Pcaps per EtherNet/IP emulator profile	151
59	Percent different bytes between Pcap files compared to reference Pcaps - Six Pcaps per EtherNet/IP emulator profile	152
60	Initial and final trees for STEP7 browse task using two input traces	156

Figure		Page
61	Percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per ISO-TSAP emulator profile	157
62	Percent different bytes between Pcap files compared to reference Pcaps - Three Pcaps per ISO-TSAP emulator profile	158

List of Tables

Table	Page
1	Linux VM configuration 130
2	Windows XP VM configuration 131
3	High-level summary of results 137
4	HTTP protocol tree build time (sec) 139
5	HTTP protocol tree build time confidence intervals 139
6	HTTP protocol tree throughput (nodes/sec) 139
7	HTTP protocol tree throughput confidence intervals 139
8	Statistics for percent different bytes between Pcap files compared to reference Pcaps - One Pcap per HTTP emulator profile 143
9	Statistics for percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per HTTP emulator profile 143
10	EtherNet/IP protocol tree build time (sec) 145
11	EtherNet/IP protocol tree build time confidence intervals 145
12	EtherNet/IP protocol tree throughput (nodes/sec) 146
13	EtherNet/IP protocol tree throughput confidence intervals 146
14	Statistics for percent different bytes between EtherNet/IP emulator Pcap files compared to reference Pcaps 152
15	ISO-TSAP protocol tree build time (sec) 153
16	ISO-TSAP protocol tree build time confidence intervals 153
17	ISO-TSAP protocol tree throughput (nodes/sec) 154
18	ISO-TSAP protocol tree throughput confidence intervals 154

Table		Page
19	Statistics for percent different bytes between ISO-TSAP emulator Pcap files compared to reference Pcaps	155

List of Acronyms

ACK	acknowledgment
AHA	Adaptive Honeypot Alternative
AICS	Automatic Intelligent Cyber-Sensor
APRT	Advanced Protocol Reversing Tool
ASAP	Automatic Semantics-aware Analysis of Network Payloads
BFS	breadth first search
CIP	Common Industrial Protocol
CM	Connection Manager
CRISALIS	Critical Infrastructure Security Analysis
CUT	components under test
DFS	depth first search
DNS	Domain Name System
DoS	denial of service
DT	data type
DTA	dynamic taint analysis
EtherNet/IP	EtherNet Industrial Protocol
FSM	finite state machine
FTP	File Transfer Protocol

GEXF	Graph Exchange XML Format
GUI	graphical user interface
HI	high-interaction
HMI	human machine interface
HMM	Hidden Markov Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
ICS	Industrial Control Systems
IDE	Integrated Development Environment
IDS	intrusion detection system
IED	intelligent electronic device
IP	Internet Protocol
IPv4	Internet Protocol version 4
ISN	initial sequence number
ISO-TSAP	ISO Transport Service Access Point
IT	information technology
LI	low-interaction
MAC	media access control

MCB	most common byte
MR	mutation rate
MSA	multiple sequence alignment
MSS	maximum segment size
MTU	master terminal unit
NA	neighbor advertisement
NIST	National Institute of Standards and Technology
NMF	non-negative matrix factorization
NS	neighbor solicitation
ODVA	Open DeviceNet Vendors Association
OS	Operating System
PAM	Partition Around Medoids
Pcap	packet capture
P-tree	protocol tree
PI	Protocol Informatics
PLC	programmable logic controller
PRISMA	Protocol Inspection and State Machine Analysis
RA	Region Analysis
RRP	request-response pair

RTU	remote terminal unit
SCADA	supervisory control and data acquisition
SM	state machine
SN	sequence number
SNMP	Simple Network Management Protocol
SUT	system under test
TCP	Transmission Control Protocol
TTL	time to live
UCMM	Unconnected Message Manager
UDP	User Datagram Protocol
UPGMA	Unweighted Pair Group Method with Arithmetic Mean
URL	uniform resource locator
VM	virtual machine
XML	Extensible Markup Language

List of Algorithms

Algorithm	Page
1 Sequence Alignment [KT06]	37
2 Weighted Interval Scheduling [KT06]	43
3 Weighted Interval Scheduling Find Solution [KT06]	44
4 Consolidate Link Proposals	93

AUTOMATIC CONFIGURATION OF PROGRAMMABLE LOGIC CONTROLLER EMULATORS

I. Introduction

1.1 Background

Modern societies such as the United States rely on ICS to provide basic services such as water, electricity, and gas [Whi13]. These systems are made up of a network of sensors and controlling devices [SFS11]. Due to the unique needs that this critical infrastructure provides, past emphasis has been placed on ensuring simplicity, robustness, and continuity of service. However, over time these systems were required to be more interconnected and complex [SFS11, HCA⁺09]. Common ICS devices such as programmable logic controllers (PLCs) have been shown to be increasingly connected to the Internet, and it is apparent that security is not inherent in the design of these exposed devices [San12]. Efforts to shore up vulnerabilities have proven to be difficult because most techniques that are common in the rapidly changing information technology (IT) industry are not applicable in an environment where stability and continuity of service are most important [SFS11, DBDS05]. Instead, resilient control system design must be employed by using network sensors to monitor device states [CCG⁺11, VML14].

This need has been elevated by recent attacks on PLCs, such as Stuxnet, which clearly show that these devices are targeted and can be abused to cause significant physical effects [Fal10, HCA⁺09]. Targeting has become even easier with device identification techniques using Shodan [Wil89]. However, the scope of ICS attacks, as well

as the actors and motivations behind them are unclear. Deploying network sensors such as honeypots has been shown to be an effective technique in characterizing and attributing anomalous network behavior [MB09, STO⁺11, VML14].

Honeypots attempt to entice unauthorized network traffic by acting as a decoy of a target device of interest [PH07]. Honeypots are characterized by the services they provide and the level of interaction of those services. A high-interaction (HI) honeypot provides a real system with real services, while a low-interaction (LI) honeypot provides a simulated environment that only emulates selected services. Both have their advantages and disadvantages, and they can be used together to form a hybrid honeypot. HI honeypots are implemented using a monitored physical device or virtual machine (VM) that resembles the target device, while LI honeypot emulators require only knowledge of their selected services' behavior. Many solutions exist to quickly set up these network sensors for an IT network.

1.2 Motivation

Developing honeypots for PLCs, however, poses some interesting challenges. PLCs can cost thousands of dollars each, so using physical honeypots is not scalable. Also, since PLC hardware and software are typically proprietary and unique, it requires significant time and knowledge to reverse engineer the system in order to develop an emulator. Even then, one emulation solution for a particular vendor's model will not likely work for other devices.

Proxying is one technique that can be used to offset the above challenge, but it is still not scalable since a single real device can only handle so many requests. Another technique is to automate the creation of a LI honeypot. Tools such as Honeyd can emulate at the network level of the protocol stack [Pro13], and it can be automatically configured by capturing network traffic [VM14, HH13, Hie04, KSAMA04]. Addition-

ally, the application layer emulation can be automatically trained by partially reverse engineering protocols from network traces and/or program binaries [LC11].

Together, a HI honeypot proxy and a trained LI honeypot form a hybrid solution where the trained LI honeypot handles all requests that it knows how to handle [PH07]. New (more interesting) traffic is proxied to the real PLC, and the LI honeypot can use the real PLC’s response to refine its training. With this setup, the frontend LI honeypot acts as a traffic *filter*, allowing the system as a whole to handle more requests.

SGNET [LD08], and GQ [CPW06], are two hybrid honeypot systems. Both were tested on IT networks. However, a Critical Infrastructure Security Analysis (CRISALIS) project presentation by Symantec researcher Corrado Leita (author of ScriptGen) seems to indicate that SGNET is also being used on Siemens PLCs in order to assess the security of Europe’s power grid [Lei12]. To date, however, no research or performance metrics have been published that show the suitability of either of these hybrid honeypot systems for usage in ICS networks. Suitability with one brand of PLC such as Siemens also does not guarantee suitability for other manufacturers’ products. Each have their own proprietary hardware, software, and network protocols. Additionally, source code for the projects has not been released, so it is not possible to do a comparative suitability study.

Thus, there is a need to develop an open-source hybrid honeypot system that is optimized for ICS networks. This research focuses on developing the self-learning LI honeypot portion of the hybrid system.

1.3 Research Goals

The goal of this research is to develop and test a flexible framework that can automatically configure the emulation behavior of a networked PLC using network

traces. This configuration must be able to be done without any knowledge of the network protocol used. Additionally, the emulation must be accurate enough to be able to generate traffic that is acceptable to industry standard tools, and is also not statistically different than normal traffic generated by the target device.

It is hypothesized that a target device’s behavior (as seen from the network) for a particular task can be accurately emulated with as little as three training traces.

1.4 Approach

A framework is developed that is primarily based off of ScriptGen’s design [LMD05, LDM06], but with some novel features that allow the resultant emulators to robustly handle unknown state transitions and keep communicating on the long, often repeating sessions that are typical with PLC master/slave polling architectures. Additionally, contextual information is derived from the network traffic similar to Roleplayer’s design [CPWK06] in order to dynamically change Internet Protocol (IP) addresses, host names, and port numbers.

Network traffic is generated by interrogating two PLCs. The first PLC is an Allen-Bradley ControlLogix L61 PLC equipped with EtherNet Industrial Protocol (EtherNet/IP) ENBT and DC Output modules. Interrogation is done by running `wget` on the PLC webserver and also by requesting module information via Rockwell’s RSLink software. The other PLC is a Siemens S7-300 equipped with discrete and analog input/output. This PLC is interrogated using Siemens’ SIMATIC STEP7 software to browse module information. Collected Hypertext Transfer Protocol (HTTP), EtherNet/IP, and ISO Transport Service Access Point (ISO-TSAP) captures are used to generate respective protocol state machines.

1.4.1 Protocol state machine generation.

The primary script parses desired traffic from a packet capture file and recursively builds protocol state machines (SMs). Each SM is represented by a tree with the nodes corresponding to server messages and the edges corresponding to client messages. Messages are initially clustered by the order of appearance in the session, and then they are clustered again using Unweighted Pair Group Method with Arithmetic Mean (UPGMA) with a distance matrix based off of the Smith-Waterman algorithm [SW81]. The resultant cluster (phylogenetic tree) is used to perform progressive multiple alignment using the Needleman-Wunsch algorithm [NW70]. The byte-aligned messages allow for fixed and mutating regions to be identified. Phylogenetic tree creation and alignment is powered by the Protocol Informatics Framework [Bed04a].

Next, message fields, or regions, are identified using ScriptGen’s *Region Analysis* algorithm [LMD05]. A regular expression is created to represent the cluster and contextual links (session IDs, etc) are identified. A default error message is then selected from among the server response. Finally, the protocol tree is exported as a Python pickle file.

1.4.2 Protocol replay.

Two scripts are developed that import the generated protocol tree and allow for traffic replay. The primary script emulates the server, and the other emulates the client. The client emulation script simply randomly walks down the protocol tree, and it is used primarily for testing the tree against itself or fuzzing the PLC. The server replay script includes a novel *backtracking* algorithm developed to robustly handle unknown transitions and allow for message looping.

1.4.3 Experimentation.

Three experiments are developed to test the primary protocols of the L61 and S7-300 PLCs. A HTTP test uses `wget` to download pages from the L61's webserver while the EtherNet/IP and ISO-TSAP experiments browse chassis modules on the L61 and S7-300, respectively. Each experiment is designed to test emulator accuracy and to also determine the number of required traces for successful emulation. All experiments involve a scripted collection of reference captures. Random combinations of these captures are then used to build protocol trees. Next, each tree is used to emulate server responses in order to generate experimental captures. All captures are randomized, allowing several task runs to occur prior to capturing traffic.

Once all random captures are complete, the packet capture files are compared in pairs to measure the amount of variability between each pair. Comparisons count the number of different bytes between each file. Variability differences in the reference captures serves as a baseline. This is statistically compared to the variability found between the experimental and reference capture pairs. Any significant difference in variability indicates that the traffic produced by the emulator is different.

Additionally each test measures accuracy by a second criteria—interrogator output. For the HTTP test, downloaded Hypertext Markup Language (HTML) directories are compared. Comparisons include measuring the number of bytes downloaded and the number of identical and different files. For the EtherNet/IP and ISO-TSAP experiments, the output of the RSLinx and STEP7 graphical interfaces, respectively, are scored as a pass or fail. This is automatically done using OpenCV image recognition software and SikuliX [Hoc15].

1.5 Assumptions and Limitations

This research provides a proof of concept framework for automatically configuring PLC emulators based solely on a limited number of network traces. All trace-based approaches have limitations since it is typically not possible to glean the entire state of a network device solely from bytes sent over the network. Additional limitations apply to this proof of concept's current implementation and can be resolved in future work.

1.5.1 Limitations inherent with network trace-based approaches.

There are two limitations that are true with any approach that attempts to mimic network behavior based solely on network traces. For one, only traffic similar to what was previously seen can be emulated. This is particularly important for identifying fields of messages that should vary. If a particular byte is always seen as one value, there is no way for the emulator to know that in reality it can (and should) vary. Therefore, the quality of emulation is directly related to the quality of the inputted network traces. This research assumes that the traces generated are not missing any packets, but it does not assume by default that two traces are different in all the bytes that do vary.

Secondly, traffic that is encrypted is not able to be emulated correctly unless that traffic has no dynamic fields (all content is fixed and can be simply be replayed as is).

1.5.2 Network protocols involved.

Only HTML, EtherNet/IP, and ISO-TSAP are tested. In addition, emulation is performed only on the application layer of the network stack. The tested framework can be extended to emulate the transport layer or it can be used as a subsystem

in Honeyd to provide lower level emulation [PH07]. Finally, the tools currently only support Transmission Control Protocol (TCP) and Internet Protocol version 4 (IPv4).

1.5.3 Single connection server.

The current implementation of the replay script only supports one connection at a time. None of the experimental tasks chosen require multiple concurrent connections, but a task such as capturing web browser traffic requires an update to the replay script.

1.5.4 Limited set of tasks and traces.

For this proof of concept, only one task is formally tested for each of the three protocols. There are many more complex conversations that can be tested (particularly those that use EtherNet/IP and ISO-TSAP). Additionally, the traces are generated in a laboratory environment and are not representative of the typical traffic seen in a production environment. Production traces have many conversations in them, requiring the protocol tree generation software to work harder at clustering and aligning messages. Processing time could potentially become a factor. This can be mitigated somewhat by using capture filters and limiting the size of the resultant protocol tree.

1.5.5 Limited configuration setup.

Only one hardware and firmware configuration is tested for each PLC. Allen-Bradley has other PLC models such as CompactLogix, various chassis sizes and configuration possibilities (input and output modules) for ControlLogix, and many firmware versions. Similarly, Siemens has a variety of other models and configurations to choose from. All of these factors could potentially affect server responses. Therefore, the generated trees may only be valid for the configurations tested. More

traces would likely be required to accurately emulate other configurations. It should be noted that Leita et al. showed in [LDM06] that ScriptGen required only 50 traces to accurately emulate behavior for over 50 Windows configurations.

1.5.6 Timing not considered.

Real and emulated conversation flows may differ in timing, but it is assumed that this will not adversely affect the content of those flows.

1.5.7 Interrogators may ignore or accept invalid responses.

Scripted interrogators in particular may ignore unexpected emulator responses and provide no indication that anything is out of the ordinary. Certain responses may be critical for the script to continue without error. However, any deviation from the expected messages from either the client or server are detected when comparing packet capture byte variability as described in the experimental approach.

1.6 Thesis Overview

Chapter II provides background and related research on honeypot technology, reverse engineering protocol techniques, and usage of those techniques in developing advanced honeypots. Chapter III covers a detailed description of the developed framework. Chapters IV and V cover experimental design and results. Finally, Chapter VI presents the research conclusions and future work.

II. Background and Related Research

2.1 Overview

This chapter provides a background of why Industrial Control Systems are important, how they are vulnerable, and how honeypots can be used to secure them. It is shown that traditional security paradigms are not sufficient because control system devices are insecure by design. Instead, the state of these devices needs to be monitored using network sensors such as honeypots.

A brief background on networking is provided as a prerequisite to understanding honeypot technology. Additionally, several algorithms are discussed in Section 2.2.7 that are important to techniques used in developing protocol-agnostic emulators. Finally, a thorough survey of honeypot technology and protocol reverse engineering techniques is provided.

2.2 Background

2.2.1 Industrial Control Systems.

Presidential Policy Directive 21 defines 16 sectors of the United States as *critical infrastructure* [Whi13]. These include areas such as food, transportation, chemicals, energy, water, and nuclear reactors. ICS are used in all of the above in order to provide reliable, safe, efficient, and *automated* service [SFS11].

Industrial Control Systems, which include supervisory control and data acquisition (SCADA) systems, can be broken up into three main components. Figure 1 shows that these components are a human machine interface (HMI), remote diagnostics and maintenance, and a control loop [SFS11]. Within the control loop are a set of sensors that provide data to the controller. The controller uses the sensor readings, which could be as simple as the level of water, in order to autonomously make decisions about

how to affect the controlled process. For instance, when the water level becomes too high the controller can command an actuator to open a valve. This autonomous feedback loop provides an abstraction to the controlled process, making it easier for human operators to manage it (typically via the HMI).

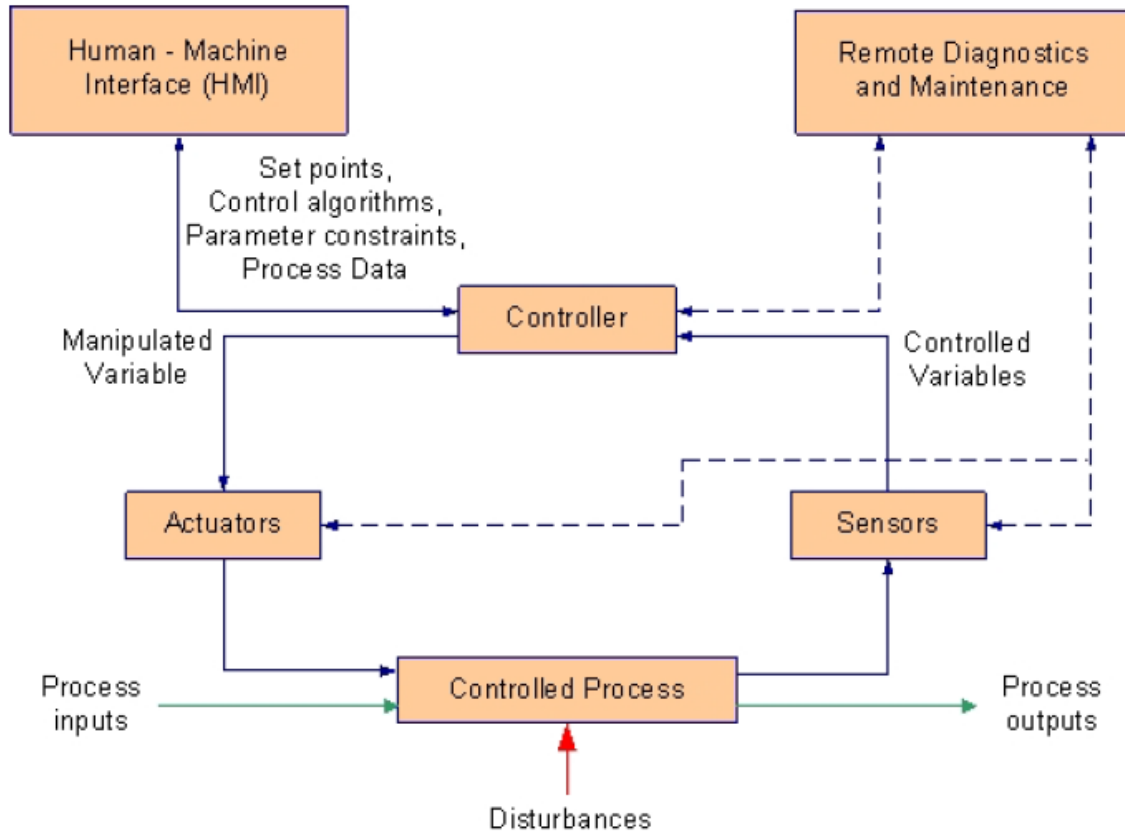


Figure 1. ICS block diagram [SFS11]

Breaking down the components further, Figure 2 shows a general SCADA layout [SFS11]. This figure helps to highlight the fact the ICS equipment can be separated by great geographic distance. Several data links connect various parts of the ICS. The primary control center contains components such as the HMI, a data historian (control process log server), and master terminal unit (MTU). The MTU polls the remote terminal units (RTUs) in a master/slave communication style in order to collect specific data. The RTUs process and summarize the data collected from the

sensors (not shown) over time. Next, the intelligent electronic device (IED) is a smart sensor that performs some local data processing. Finally, it should be noted that a PLC can be used as a RTU (as shown) or a controller.

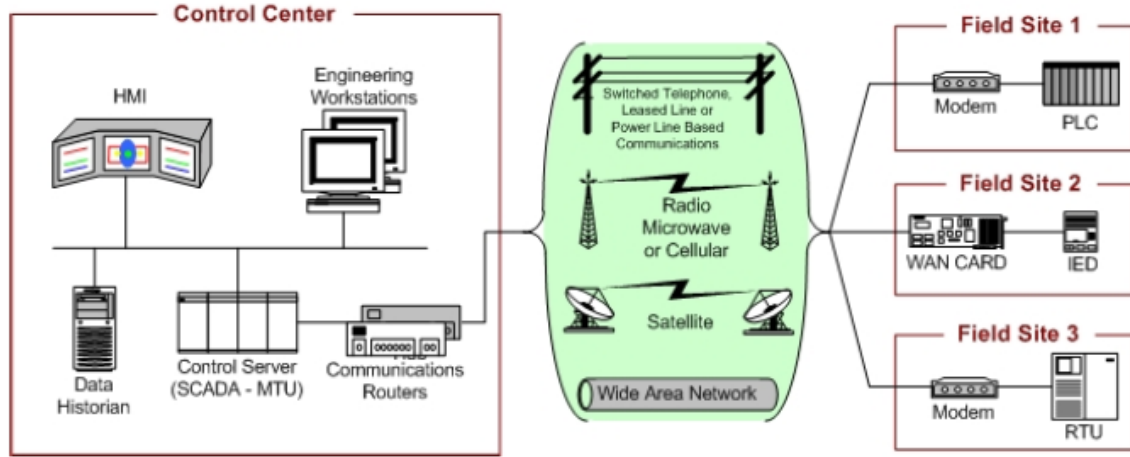


Figure 2. General SCADA layout [SFS11]

In fact, PLCs are one of the most flexible components in ICS. They can control, monitor, and manage sensors and actuators in a variety of settings and configurations by using modules. Each vertical slot shown on the PLC in Figure 2 could be a module for digital or analog input/output, a communications module, a controller, etc. A PLC is typically made with vendor-specific hardware and firmware, and they can be programmed remotely for specific applications using software such as Ladder Logic [SFS11].

2.2.2 Allen-Bradley PLCs.

Allen-Bradley is a common PLC vendor in the United States. They provide a wide range of controllers such as the ControlLogix and MicroLogix families. This research focuses on the former. Common controllers from the ControlLogix family include the L6x and L7x controllers. The controllers manage communications between other modules (such as input/output) across the chassis backplane. At least one of the

other modules is for external communications such as a Web+ or EtherNetIP ENBT module. They provide a network interface for the PLC and contain a web server for viewing device statistics and configuration.

Further device configuration can be done remotely via Rockwell’s RSLogix software suite. RSLogix allows an operator to remotely monitor, control, troubleshoot, and program PLCs. Programming includes updating firmware and application software. For the latter, RSLogix can be used to develop and perform runtime analysis of Ladder Logic. All of these remote capabilities depend on communication drivers which are managed by a program called RSLinx. Before any communication to the PLC can occur, one must first set up a driver and browse for the device using RSLinx’s *RSWho* interface. RSLogix has a similar interface called *WhoActive* that can be used to browse for devices once an appropriate driver is set up.

2.2.3 Siemens PLCs.

Siemens is also a common PLC vendor, and it is particularly popular in Europe. Siemens provides many models of controllers including the S7-200, S7-300, and S7-1200. This research uses the S7-300. The software that controls these PLCs is called SIMATIC STEP7. Rather than setting up a driver as with the Allen-Bradley models, in STEP7 a network node must be configured to upload station objects (e.g., PLC module information).

2.2.4 Industrial Control Systems (in)security.

Unfortunately, the great flexibility and reliability of ICS devices do not provide adequate security for today’s interconnected networks. An example of the inherently insecure designs of PLCs, for instance, can be seen in Digital Bond’s Project Basecamp [San12]. As a part of Project Basecamp, Santamarta used *legitimate* commands to

remotely exploit an Allen-Bradley ControlLogix L61 PLC with an attached ENBT/A module. He was able to remotely change the device’s IP address, force a CPU halt or crash, crash the ENBT, change or capture firmware, and more. The vulnerabilities that Santamarta exploited were not software bugs; they were features of the design for ease of use. PLCs have also been exploited out in the wild such as in the much publicized Stuxnet attacks that damaged nuclear centrifuges [Fal10].

Exploitation of ICS devices is expected to rise as control systems become more interconnected and more complex [SFS11, HCA⁺09]. With the advent of Shodan, an Internet search engine for embedded devices, finding vulnerable PLCs has become even easier. Researchers have used crafted Shodan queries in order to enumerate, catalog, and create visual mappings for vulnerable ICS devices [Lev11, Hig13]. Williams extended this work by showing how to use downloaded device tags to make PLC identification specific enough to enumerate by industry and function [Wil89].

However, while it is clear that Shodan makes it easier to identify vulnerable devices, it is unclear whether Shodan affects how often ICS devices are attacked. In two 2013 TrendMicro reports, Wilhoit seeded Google and Shodan with entries for several honeypots that he placed on public IP addresses around the world. The first experiment lasted 28 days and resulted in 12 targeted attacks [Wil13b], and the second experiment lasted four months and resulted in 74 targeted attacks [Wil13a]. It is unclear, however, how many attacks were targeted against HMIs and how many were targeted against PLCs [Wil89].

In 2014, Bodenheimer performed an experiment to test the impact Shodan has on PLC network activity [Bod19]. Over 55 days, four Allen-Bradley PLCs were deployed to an ICS network address space with Internet-facing IP addresses. Two PLCs used default configurations, and two PLCs used modified service banners so they could be targeted easier via Shodan. Contrary to Wilhoit’s research, no targeted attacks

were found over the testing period. It was concluded that Shodan had no significant experimental impact on device targeting [Bod19].

Regardless of Shodan, ICS exploitation is on the rise [SFS11, HCA⁺09]. The implications of this can be very serious as Huang et al. showed in [HCA⁺09]. They designed threat models in order to assess the impacts of denial of service (DoS) and integrity attacks against PLCs used in chemical processes. Their findings showed that targeting the reactor pressure sensor could result in an unsafe state (i.e., cause reactor shutdown or *explosion*).

The real question, then, is what should be done to prevent critical infrastructure devices from being exploited? National Institute of Standards and Technology (NIST) published a guide for ICS security in 2011 that helps to address this question [SFS11]. The guide clearly identifies the root of the problem: ICS devices are different from IT devices. Control systems were originally isolated special-purpose networks that were optimized using proprietary hardware, software, and protocols. Over time, these special-purpose networks were interconnected with business networks in order to drive down costs and make timely business decisions based upon real-time control system data. This caused a classical example of requirements creep because the control system devices were not designed for networks with potentially untrusted parties. As was discussed previously with Project Basecamp [San12], many control system devices will do anything a remotely connected user requests.

Unfortunately, ICS networks cannot be patched or updated in the same manner as IT networks [SFS11]. While IT networks are designed to prioritize data integrity and confidentiality, ICS networks prioritize availability and safety above all else. This should not be surprising as no company, for example, wants to be known as the one that can only provide power or clean water 20 hours a day. Control systems provide *time-sensitive* services, so providing down time for patching and rebooting machines

in order to improve security is difficult [SFS11, DBDS05]. Approximately 90% of the companies providing services using control systems are *privately* owned [SFS11]. From the business perspective, the guaranteed consequences (costs) simply outweigh the risks associated with potential attacks.

Additionally, the risks associated with attempting to secure these devices is compounded by the fact that they are old. Compared to IT equipment with an average operational lifespan of three to five years, ICS devices remain in production for 15-20 years [SFS11]. This means that recent security methods often do not work, and they sometimes can be downright hazardous. For instance Duggan, Berg, Dillinger, and Stamp cited many issues with typical penetration techniques [DBDS05]. Simple ping sweeps were identified as being responsible for unexpected robotic arm movement and, in an fabrication plant, led to the destruction of \$50K worth of integrated circuits. Another incident was cited where a penetration test caused customers to lose gas service for four hours. Since ICS devices are often several generations behind their IT counterparts, their resources (e.g., memory and processing) can be easily exhausted.

One of the common methods to help alleviate risks associated with corporate and ICS interconnectivity is to ensure that the two networks are separated [DBDS05, SFS11]. Figure 3 shows NIST’s recommended network configuration for splitting the corporate and control networks using a firewalled neutral network (DMZ) [SFS11]. The idea is to minimize untrusted traffic from reaching the control network.

However, separating the networks does not deal with the crux of the matter: ICS devices are fragile and too trustful. Instead, there must be a paradigm shift to “resilient control systems” [RGM09]. These are defined by Rieger, Gertman, and McQueen as systems that:

...maintain state awareness and an accepted level of operational normalcy in response to disturbances, including threats of an unexpected and malicious nature [RGM09].

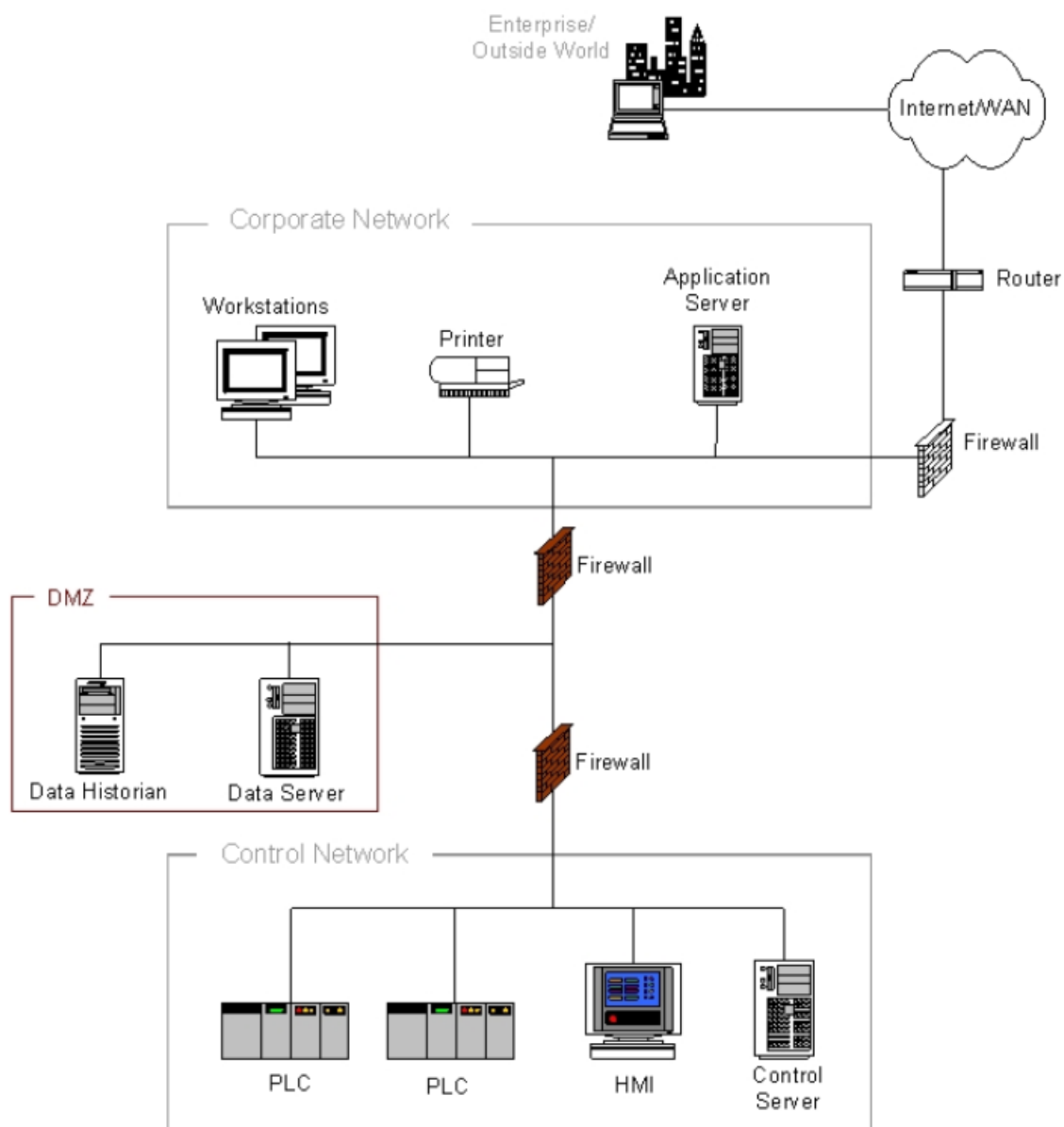


Figure 3. ICS network configuration recommended by NIST [SFS11]

This concept of “state awareness” was addressed by Carcano et al.’s *Critical State Analysis* technique [CCG⁺11]. They recognized that most control systems have well documented critical states and that the state of the system can be monitored to detect when it is converging on a critical state. In order to do this effectively, several metrics were proposed that yield a distance of the current state to a critical state. In contrast to signature-based methods, Carcano et al. cited their method as effective for detecting zero day attacks.

Finally, other researchers have called on new anomaly detection paradigms for ICS [MB09, VML14, STO⁺11]. McQueen and Boyer’s research cited deception as a needed technique for control system defense [MB09]. Their research mapped different methods of data hiding and information falsification to seven abstract dimensions of security. Further research showed that hybrid sensors consisting of intrusion detection systems and honeypots proved to be much more effective than traditional techniques [VML14, STO⁺11].

In summary, Industrial Control Systems have been shown to be critical for providing every day services, yet these systems are exposed and vulnerable to cyber threats. Securing control systems with traditional IT methods is not effective, because vital components such as PLCs are aging and were designed to openly trust other devices. Thus, a new paradigm is needed based on autonomously maintaining awareness of the system’s current state. Researchers have proposed new network sensors to accomplish this, and honeypots are an integral part of that solution. To that end, Section 2.2.6 provides an overview of current honeypot technology after a brief prerequisite networking review.

2.2.5 Networking and protocols.

This section includes a very brief overview of some networking principles and network protocols of interest for this research. They are an important prerequisite for understanding how honeypots work including the one designed in this research. For more networking basics the reader is referred to [KR13].

2.2.5.1 Networking review [KR13].

Network protocols define how two devices talk to one another. Internet protocols are layered so that they are more modular and easier to maintain. Figure 4 shows that the basic protocol layers are divided into five parts. Each layer includes a header and payload, where the payload consists of the layers above it. For this research, the discussions will primarily concern the transport and application layers.

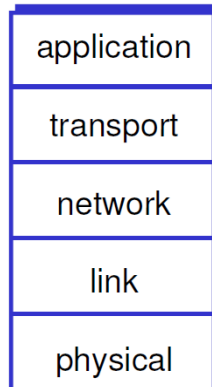


Figure 4. Internet Protocol stack [KR13]

The transport layer facilitates communication between two processes. If the processes are on the same host, then interprocess communication can be used. Otherwise, messages (data segments) need to be passed between the hosts using a transport protocol such as TCP or User Datagram Protocol (UDP). Both protocols use source and destination ports as can be seen in Figure 5. The ports, along with the source and destination IP addresses are collectively called a *four-tuple*. The four-tuple is used to

identify which *socket* the data segment should go to, and the socket corresponds to a specific application process.

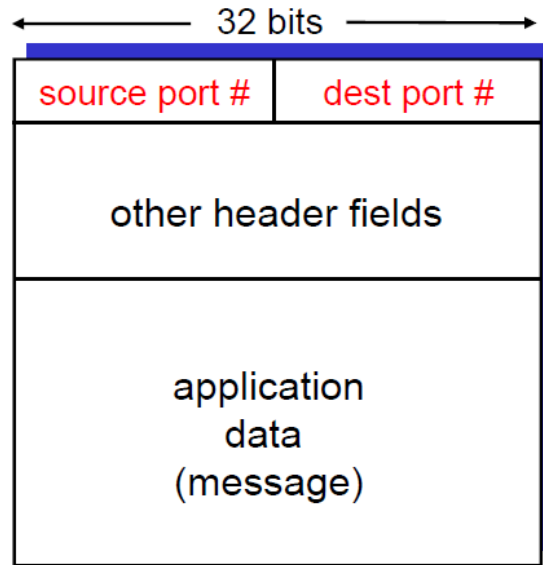


Figure 5. TCP and UDP segment format [KR13]

Once the application payload is delivered to the right process, the application layer processes it according to the protocol in that layer. For instance, when a web page is retrieved from a server, HTTP is the application layer protocol used to deliver the HTML code to the browser application. The browser then interprets the markup language and displays the results.

Overall, the process described can be thought of in terms of receiving mail. The sender marks the mail with their address (source IP) and the destination address (destination IP). The mail is transferred to the destination house via a network layer protocol (IP). Then, a transport layer protocol (TCP) delivers the mail to the destination door (a socket). Once inside the house the mail envelope is opened by an application layer protocol (HTTP), and the letter is read (HTML).

It is important to note here that the application layer itself can have other layers within it. For instance Ethernet/IP, which will be described later, feeds into a higher protocol called Common Industrial Protocol (CIP).

It should also be noted that TCP is a connection-oriented protocol that provides reliable transfers. This is accomplished in many ways, but of important interest for this research is how TCP establishes a connection and how it tears one down. Referring back to Figure 5, TCP includes some important fields in the “other header fields” section of the segment. These include a 4-byte sequence number, a 4-byte acknowledgment number, and several single bit flags. Figure 6 shows how TCP establishes a connection using these additional fields. The client first attempts to establish a connection by sending a SYN message (SYN flag set). At this time the initial sequence number (ISN) is established. If the server agrees to accept the request, it replies back with a SYN-ACK (SYN and ACK flags set) and its respective ISN. The ACK flag indicates that the acknowledgment field is valid. That field indicates what byte is expected *next* from the other party. After the client receives the SYN-ACK from the server, it replies back with another ACK. From this point on, the connection is established until torn down. Sequence and acknowledgment numbers continue to increase with the size of the data sent and received. Because of this, even if the messages are received out of order, the application layer can reorder them by the sequence numbers.

After the client decides to end the connection, the proper way to initiate it is to send a FIN flag (see Figure 7). The server then acknowledges this and sends any final data. After the server is done it tells the client it is finished. After this the client sends the final ACK.

2.2.5.2 Application layer protocols of interest.

The protocols of interest for this research include HTTP, EtherNet/IP, CIP, and ISO-TSAP. HTTP is a stateless, text-based protocol defined in RFC 2616 that facilitates the transfer of hypertext [FGM⁺99]. Fields are separated by spaces and

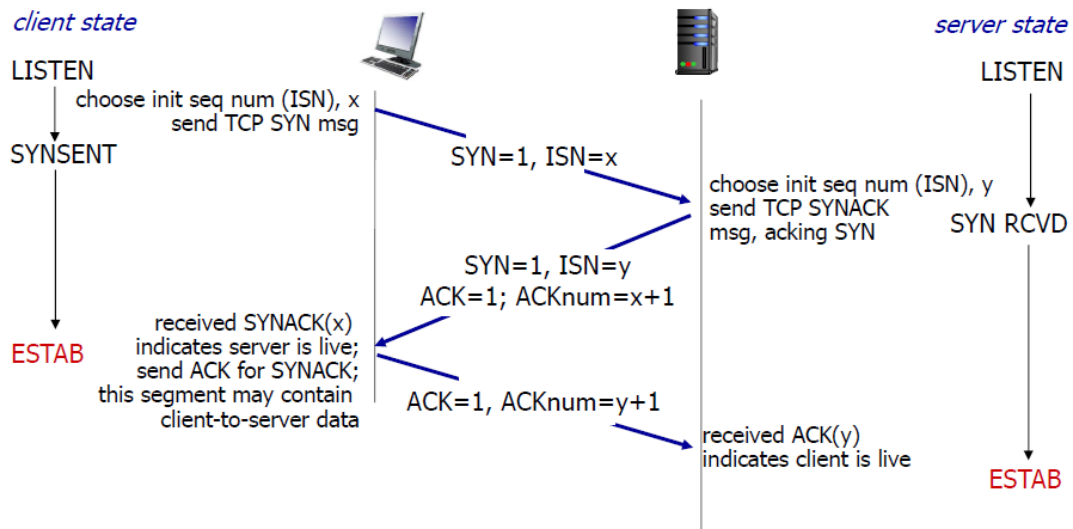


Figure 6. Establishing a TCP connection [KR13]

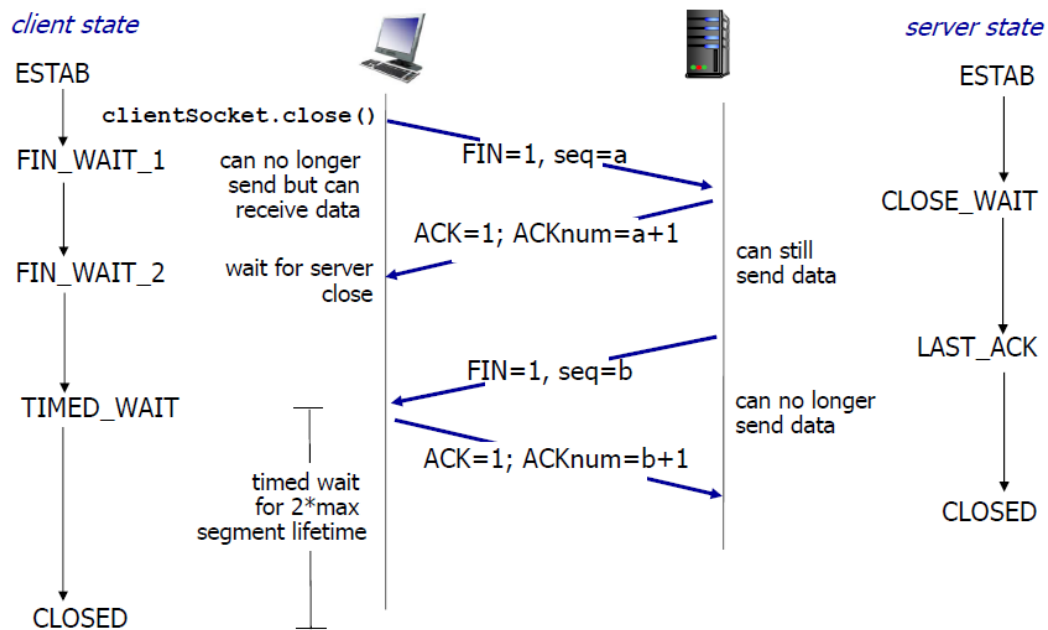


Figure 7. Closing a TCP connection [KR13]

the header is broken into lines using carriage return plus newlines ('\r\n'). There are request and response messages. Request messages typically use GET or POST methods to retrieve resources identified by a uniform resource locator (URL). The hostname of the server and other information is also included in the header which is followed by a blank line. Figure 8 show an example of a GET request. Response messages include a status line, date, content-length, and other header lines, a blank line, and finally the data. Figure 9 shows an example of a HTTP response message.

```
GET /index.html HTTP/1.1\r\n
Host: 192.168.1.206\r\n
Connection: keep-alive\r\n
Pragma: no-cache\r\n
Cache-Control: no-cache\r\n
Accept: text/html,application/xhtml+xml,application/javascript\r\n
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64)\r\n
Accept-Encoding: gzip, deflate, sdch\r\n
Accept-Language: en-US,en;q=0.8\r\n
\r\n
```

Figure 8. HTTP GET request from Wireshark

```
HTTP/1.0 200 OK\r\n
Date: THU JAN 29 01:34:11 1970\r\n
Server: GoAhead-webs\r\n
Last-modified: TUE JAN 01 00:01:52 1980\r\n
Content-length: 1209\r\n
Content-type: text/html; charset=utf-8\r\n
Connection: Close\r\n
\r\n
```

Figure 9. HTTP response from Wireshark (data not shown)

EtherNet/IP and CIP were developed to bridge the communication gap between automation devices and to also capitalize on the prevalent use of Ethernet and TCP/IP [Bro01]. Modern automation network architectures are required to provide control, configuration management, and data collection capabilities. Often separate networks are used for each task, but devices must be able to communicate efficiently together. This used to be difficult because different devices communicated with dif-

ferent protocols across different media. Then the standard protocols of EtherNet/IP, ControlNet, DeviceNet, and CompoNet were all unified under a standard application protocol—CIP.

CIP is supported by several vendors and Open DeviceNet Vendors Association (ODVA). It is able to effectively communicate by using an object-oriented approach based on a producer/consumer model [Bro01]. Contrary to the traditional IT paradigm of using source and destination addresses, CIP is designed to allow data to be identified by class objects and IDs. A standard set of device profiles and objects with attributes are defined which allow for plug-and-play interoperability between devices from various vendors. Typical objects include those such as for device identity, type, and functions. Vendors are encouraged to use the standard public objects, but class IDs 100-199 are reserved for vendor specific objects. Typically, vendors define their own object values [Dun13].

CIP has two types of messages: explicit and implicit. Explicit messages are sent over TCP port 44818 for multi-purpose communication that follow the typical request/response paradigm. All explicit messages are sent through the Message Router object. Implicit messages are sent over UDP port 2222 and are used for application-specific I/O transfers. The primary implicit types of messages are polled, cyclic, strobed (multi-cast poll), and changed state [Bro01].

As can be seen in the CIP object model in Figure 10, access to any internal object model is obtained through the Unconnected Message Manager (UCMM) or Connection Manager (CM) [Bro01]. As the names imply, a connection must first be established (have a Connection ID) prior to routing through the CM.

The last protocol of interest, ISO-TSAP, is a protocol used by Siemens devices over TCP port 102. In a similar fashion as CIP, ISO-TSAP enables encapsulation of Siemens' proprietary control protocol called PROFINET. PROFINET allows control

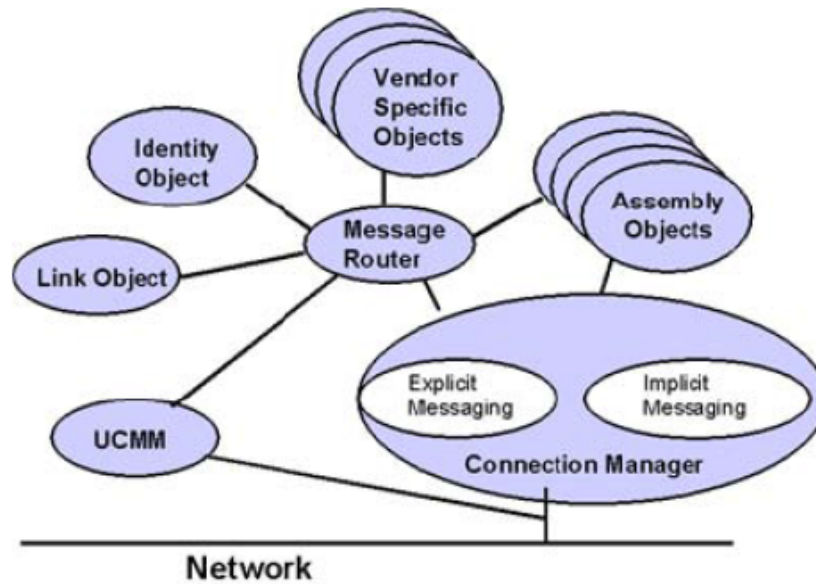


Figure 10. CIP object model [Bro01]

networks to utilize Ethernet versus the older PROFIBUS protocol that only allowed communication on specialized buses.

2.2.6 Honeypots.

Honeypots are a type of network sensor that attempt to entice unauthorized network traffic by acting as a decoy of a target device of interest [PH07]. They are typically passive in nature (i.e., they do not speak unless spoken to). Because of this, honeypots do not suffer from exhibiting false positives that signature-based intrusion detection systems (IDSs) do since *all* traffic received is suspicious. Additionally, honeypots can detect unknown attacks and overcome evasion or obfuscation techniques that are common with signature-based tools. Their uses include trapping malware samples for research, stopping botnets in their tracks, characterizing attacks, catching “bad guys” in the act (attribution), slowing down or deflecting attacks, and performing security training.

In [STO⁺11] researchers used honeypots to monitor attacks across the Internet for three years resulting in a treasure trove of information. The collected data set, called “Kyoto 2006+”, included 425,719 sessions of unknown attacks. These sessions did not trigger IDS alerts, but the extracted shell code was detected as malicious. The collected shellcodes also included 27 new variants of those used by the *Win32/Conficker.A* worm during its development period. Finally, statistics were done on all of the data to determine such things as the distribution of attacks, shellcode, end-point addresses, ports, countries involved, and more.

Honeypots are typically characterized by the services they provide and the level of interaction of those services. There are a variety of designs that tradeoff fidelity, security, and performance [PH07]. Fidelity refers to how realistic the honeypot service is, while performance refers to how many interactions or connections a honeypot can handle at a time. Finally the security of a honeypot is determined by how difficult it is for an adversary to reconfigure it or misuse it to harm other systems. The following discussion gives a brief survey of honeypot types and their enabling technologies.

2.2.6.1 High interaction honeypots.

A HI honeypot provides a real system with real services [PH07]. Because of this they have high fidelity. However, more effort must be taken to ensure that they are not misused to harm other systems. Additionally, their performance might not be scalable. HI honeypots are implemented using a monitored physical device or VM that resembles the target device. Monitoring tools can include those typical of regular operated systems such as a host-based IDS, system logs, etc.

A better monitoring alternative is Sebek or its less detectable successor Qebek [SHZ10]. Sebek stealthily logs user activity by using replacing system calls with its own [PH07]. In doing so, the normal system logs are bypassed, and all the user

activity data is sent to a remote server for analysis. Replacing system calls is one of the techniques rootkits use to avoid detection, but unfortunately Sebek is still detectable since its packets leaving the machine are inconsistent with the system logs.

The server that collects the user activity is often a Honeywall server [hon09]. Honeywall can capture, control, and analyze data [PH07]. It acts as a transparent bridge between external networks and the network of honeypots (honeynet). As mentioned previously, one of the dangers of using a HI honeypot is that an adversary can use it to harm other machines. The Honeywall helps to prevent this by limiting outgoing connections and using *Snort-inline*. The latter essentially sterilizes packets seen as harmful before they leave the network.

Next, Argos is an interesting HI honeypot that uses a technique called *dynamic taint analysis* to detect 0-day exploits [PH07]. Dynamic taint analysis works by marking untrusted data streams such as those received over network. Any data structures that are modified by the marked data become *tainted* as well. The marked data is tracked in memory, and once it is used to affect execution flow an execution trace is generated. This allows Argos to detect how and when an exploit takes place. As will be discussed in the Section 2.3, dynamic taint analysis can also be used to reverse engineer network protocols.

2.2.6.2 Low interaction honeypots.

A LI honeypot provides a simulated environment that only emulates selected services. Because only selected services are emulated, LI honeypots are typically easier to configure and maintain, and they can focus their resources on handling more connections than a HI honeypot would [PH07]. Lacking a full operating system that can be exploited, they also are less likely to be used by an adversary to attack other

machines. The tradeoff to these benefits are that they are much easier to identify (fingerprint) as fakes.

Due to their selectivity, there are also a much greater variety of LI honeypots than HI honeypots. Some, such as LaBrea, *tarpit* (slow down) traffic by manipulating TCP window sizes [PH07]. Others, such as tinyHoneypot, provide adversaries with easy access to a limited root shell via vulnerable services. Arguably one of the most popular LI honeypots is Honeyd.

Honeyd is not really a honeypot itself; it is a honeypot factory. Honeyd allows the creation of thousands of virtual hosts running arbitrary services by simply editing a few text configuration files [PH07, Pro13]. Honeypot personalities and services are selected using *templates*. Templates support inheriting attributes from other templates, and they can be used to replicate as many honeypot virtual hosts as desired.

However, Honeyd actually only emulates at the network and transport protocol layers [PH07]. It even emulates network topologies. Application layer emulation, on the other hand, requires service scripts. These scripts are written in shell or Python, and many are included with Honeyd. Additionally, Honeyd can use real network services as *subsystems*. Honeyd accomplishes this by hooking into low level networking calls.

Another noteworthy honeypot is nepenthes [BKH⁺06] and its successor dionaea [BK13]. Nepenthes is optimized to catch malware (particularly self-propagating worms). This is done by providing many vulnerable service modules. Once a module is exploited, a shellcode module unpacks the malware and downloads the real payload. This can then be automatically stored in a database or submitted to anti-virus vendors. In the case where a suitable vulnerability module is not found, the adversary's packets can be redirected to a HI honeypot.

2.2.6.3 Active honeypots.

The previously discussed honeypots are all passive in nature. These all take the role of a server and expect malevolent traffic to find them. However, client system vulnerabilities are just as important a security issue. Client honeypots *actively* seek out malicious content by pretending to be an end-user [PH07]. This might be as simple as having an automated email client or web browser that clicks on every link (e.g., HoneyClient). The more advanced HI client honeypots use techniques like dynamic taint analysis discussed previously to assess exactly how their found payload affects them.

A more recent project uses an active honeypot methodology called *bait and trap*. Beeswarm attempts to lure in attackers by having drone clients leak tantalizing information in communications with a honeypot server [Ves15]. The leaked information, also known as a *bait session*, provides potential attackers with vital information such as login credentials.

2.2.6.4 Hybrid honeypots.

As mentioned previously, honeypot designs make a trade between fidelity, security, and performance [PH07]. LI honeypots typically have lower fidelity, but are easier to secure and can focus resources for better performance. HI honeypots, on the other hand, sacrifice performance and demand more security measures in order to provide high fidelity. Hybrid honeypot designs attempt to provide the benefits of both approaches. They typically achieve this by filtering traffic prior to it being sent to a HI honeypot.

Collapsar is one example of a filter and redirect approach [JXW06]. Collapsar uses distributed *redirectors* to tunnel only traffic of interest to a centralized honeyfarm. The honeyfarm consists of a group of HI honeypot VMs, a *management station*,

and a *correlation engine*. The correlation engine coalesces data from the distributed networks to detect attack trends. Additionally, the honeyfarm has a frontend gateway that acts similar to a Honeywall server. The frontend and redirectors both use Snort-inline to throttle and sterilize outgoing traffic. Overall, the scalability of this approach hinges on the efficacy of the frontend. Multiple frontends may have to be employed to achieve the desired results.

Potemkin, on the other hand, achieves scalability by efficient resource usage [VMC⁺05]. Specifically, Potemkin only provides honeypot instances *as* they are needed. Once an instance is no longer required, its resources are reclaimed. A set of reference VMs are cloned on demand using copy-on-write methods. This means that only changed files require new memory to be used. A gateway router then dynamically binds a destination IP address to the new honeypot, allowing traffic to flow to it. The router also performs load balancing, proxies selected outbound services, and prevents unauthorized traffic from leaving the network by reflecting it inward. The reflection capability is particularly interesting in that instead of allowing malicious traffic to leave the network, the traffic is redirected to a brand new honeypot instance. Thus, propagating malware can be observed spreading. Finally, a scan filter limits how fast any one external IP can cause VMs to be instantiated.

A third hybrid honeypot example is honeybrid, which was developed by Robin Berthier [Ber35, Ber13]. Honeybrid is an open-source project that uses Honeyd as a LI frontend filter. Argos or nepenthes is used on the backend for HI emulation. Honeybrid’s architecture consists of four main components, namely: 1) a decision engine, 2) a control engine, 3) a redirection engine, and 4) a log server. The decision engine controls the frontend filter. Essentially, it defines and selects a Honeyd template. Note, that the application layer emulation is completely dependent on the service script selected for the Honeyd template. Next, the control engine limits outgoing

connections similar to the previous projects. Finally, the redirection engine acts as a proxy and replayer to the selected backend honeypot. Replaying the session from the frontend is important in order for the backend to catch up to what it missed. Otherwise, the backend honeypot's state would be out of sync with the session, and the resultant response would be incorrect. This replaying technique is used in other hybrid honeypots that is discussed in Section 2.3.5.

2.2.6.5 Summary.

In summary, honeypots are a very useful tool for network security. There are many variations that can cater to most needs. By using a creative mix of LI and HI honeypots a scalable, yet high-fidelity honeynet can be created.

However, all of the LI honeypots discussed so far can only handle published protocols. Emulating proprietary software and applications is difficult [vdHvW05]. In order for emulators to handle proprietary or unknown protocols in a scalable manner, techniques must be used to automatically reverse engineer the protocols. The techniques required build upon algorithms and constructs discussed in Section 2.2.7.

2.2.7 Common algorithms and constructs used in reversing protocols.

Reverse engineering protocols is similar to trying to learn a new language. First, a vocabulary of words must be built. Second, appropriate grammar must be learned to group and order the words correctly. For network protocols, the equivalent respective constructs are messages formats (vocabulary) and state machines (grammar). Both are required in order for a network protocol to be completely understood.

The method used to accomplish the above greatly depends on what is available as input. Most techniques use network traces and/or program binaries to gather the required information [LC11]. For inferring a message format using network traces, generally most techniques first cluster messages, and then they align the messages

in each cluster in order to partition the fields [BGH14]. Inferring the state machine usually involves building up a model based on the identified message formats, and then minimizing the model using probabilistic methods.

2.2.7.1 Clustering.

UPGMA. The primary clustering algorithm of interest for this research is UPGMA [NTT83]. The algorithm is considered a hierarchical clustering algorithm, and in the context of this research it is used to create guide trees for multiple sequence alignment (MSA). A general expression for the UPGMA algorithm is

$$d_{ij} = \frac{1}{|C_i| + |C_j|} \sum_{p \in C_i, q \in C_j} d_{pq} \quad (1)$$

where d_{ij} is the distance between clusters C_i and C_j , and d_{pq} is the distance between cluster elements [Bed04a].

In words this means that the distance between any two clusters is equivalent to the average of all the distances between pairs of elements, with each cluster contributing one element for each pair. Overall, the algorithm's final tree reflects the structure of the distance matrix in a hierarchical fashion.

The distance between elements is defined in a *distance matrix*. One way to create this is to perform an alignment algorithm to calculate the scores (see Section 2.2.7.2).

The tree is built from the bottom up [Edw13]. First find the shortest pairwise distance and cluster the two elements. The branch distance is then equivalent to half the shortest pairwise distance that was found. Then, all of the other distances must be updated with respect to the new cluster. This process continues until all clusters are joined. The root is implicitly set at a depth of half the mean pairwise distance. Figure 11 shows an example of a tree generated using UPGMA. The right side of the figure shows the depth of the tree at each level.

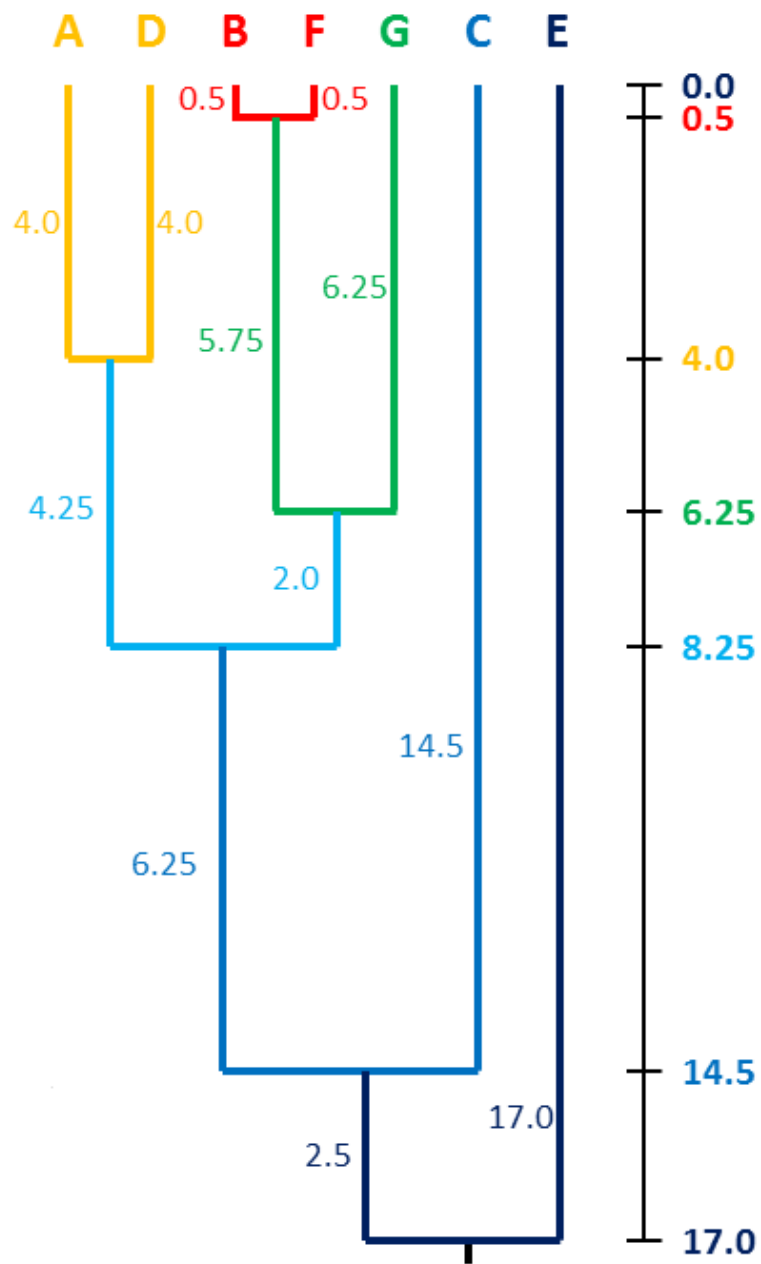


Figure 11. An example tree generated using UPGMA [Edw13]

k -means and k -medoids. Another group of clustering algorithms is based on k -means and k -medoids [Ber06]. Both algorithms are considered partitional and attempt to minimize average dissimilarity between selected centers of k clusters and the other members within each respective cluster. The difference between the two is on the choice of cluster center. k -means uses a mean value (centroid) to determine the center, while k -medoids (e.g., Partition Around Medoids (PAM)) uses the centrally located data point (the one that has minimal dissimilarity to other members of the cluster). As a result, k -means is more sensitive to outliers, but the centroids have more statistical meaning [Ber06].

Both algorithms begin by selecting a random set of k centers. Based on the centers, k clusters are computed. Next, non-centers are swapped with the centers and new clusters are computed. The centers that yield the best clusters (minimal average distance) are chosen. This repeats until convergence has been reached.

One of the difficulties with k -means and k -medoids is in choosing an appropriate value for k . Often various values are chosen until optimal cluster quality is achieved. Cluster quality can be determined by a metric called a *silhouette*, which for one cluster data point, i is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \quad (2)$$

where $a(i)$ is the average dissimilarity of i to the other points in its cluster, and $b(i)$ is the lowest average dissimilarity of i to the other points in any other cluster [Rou87].

If $s(i)$ is close to 1, this implies that the data point is in the correct cluster. If $s(i)$ is close to -1 , this implies that the data point should be in the neighboring cluster. Finally, a value near 0 indicates a weak correlation to either cluster. $s(i)$ is then calculated for every data point in order to measure if the appropriate number

of clusters are representing the data. These values can be plotted for visual analysis or simply used to generate a mean silhouette [Rou87].

Information Bottleneck. Unlike the previous clustering algorithms, the Information Bottleneck method uses data probabilities instead of a distance matrix [TPB99]. This method attempts to find the ideal tradeoff between accuracy and complexity by using a joint probability distribution between a set of random variables (X) and a set of related observed variables (Y). Essentially, the algorithm determines what information from X can predict the maximum amount of information from Y . This has a corollary to rate-distortion theory in which X maps to a set of code words, \tilde{X} , and \tilde{X} maps to Y .

Non-negative matrix factorization. Briefly, non-negative matrix factorization (NMF) is a method of clustering by factoring a matrix into two matrices with no negative values [SD05]. Solving this problem, however, is NP-complete. One method for accomplishing this in a time efficient manner is the alternating least squares approach. The least squares approach essentially involves solving a set of linear equations. Constraining one or both of the factored matrices is required for solving in a time-efficient manner.

2.2.7.2 Alignment.

The primary alignment algorithms related to the material in this research are Smith-Waterman [SW81], Needleman-Wunsch [NW70], and a progressive multiple sequence alignment algorithm. The first two algorithms efficiently align only pairs of sequences. Both Needleman-Wunsch and Smith-Waterman attempt to determine an optimal alignment based upon matches, mismatches, and gaps. They accomplish this efficiently using a technique called dynamic programming.

Dynamic programming is a technique that breaks up a problem into overlapping sub-problems, and iteratively builds up solutions to larger sub-problems using prior computations [KT06]. The sub-problem solutions are typically stored in matrices. Once the overall optimal solution is found, a trace-back through the solution matrix can be used to identify the steps to achieve the desired solution.

Needleman-Wunsch is considered a *global* alignment algorithm in that it aligns two sequences from beginning to end [NW70]. An example of an alignment for the strings CTACCG and TACATG is given in Figure 12. The figure shows two gaps denoted by a dark gray box with a ‘-’, and one mismatch at (x_5, y_4) denoted by a blue highlighted box.

x_1	x_2	x_3	x_4	x_5		x_6
C	T	A	C	C	-	G
-	T	A	C	A	T	G
	y_1	y_2	y_3	y_4	y_5	y_6

Figure 12. An example sequence alignment for two strings X and Y [KT06]

The alignment is performed by optimizing a total score based upon individual matches, mismatches, and gaps [NW70]. Algorithm 1 shows the steps to align two sequences, X , and Y ($x_1x_2 \dots x_m, y_1y_2 \dots y_n$). M represents the matrix containing the optimal solutions at each step. G denotes the *gap penalty*, and is thus a negative number (or 0). Finally, S is the *similarity matrix* where each pair is given a *similarity score*. In bioinformatics, similarity scores represent the mutation probability of a given character in a sequence. A good similarity matrix will reduce the number of incorrect gaps [Bed04a].

Algorithm 1 Sequence Alignment [KT06]

```
function SEQUENCE-ALIGNMENT( $m, n, x_1x_2 \dots x_m, y_1y_2 \dots y_n, G, S$ )  
    for  $i = 0$  to  $m$  do                                      $\triangleright$  Initialize solution matrix,  $M$ , with values  
         $M[0, i] = iG$                                             $\triangleright$  for one string compared to an empty string  
    end for  
    for  $j = 0$  to  $n$  do  
         $M[j, 0] = jG$   
    end for  
    for  $i = 1$  to  $m$  do                                      $\triangleright$  find the optimal score at each step  
        for  $j = 1$  to  $n$  do  
             $M[i, j] = \max(S[x_i, y_j] + M[i-1, j-1], \ G + M[i-1, j], \ G + M[i, j-1])$   
        end for  
    end for  
    return  $M[m, n]$   
end function
```

The optimal score at each step is derived from considering three cases [KT06]:

Case 1 M matches (x_i, y_j) :

- obtain similarity score for (x_i, y_j) and add to maximum score from aligning two strings $x_1, x_2 \dots x_{i-1}$ and $y_1, y_2 \dots y_{j-1}$

Case 2 M leaves x_i unmatched:

- pay gap for x_i and add to maximum score from aligning $x_1, x_2 \dots x_{i-1}$ and $y_1, y_2 \dots y_j$

Case 3 M leaves y_j unmatched:

- pay gap for y_j and add to maximum score from aligning $x_1, x_2 \dots x_i$ and $y_1, y_2 \dots y_{j-1}$

It is important to note the similarity scores are the inverse of distance scores. Therefore, another way to realize this algorithm is to minimize distance (mismatch and gap penalties) between two strings [KT06]. Finally, once an optimal solution is obtained, a trace-back through the solution matrix will yield the optimal global alignment.

Figure 13 shows an example solution matrix for aligning two HTTP GET requests. In this example, $G = 0$ and S values are 1 for a match and 0 for mismatch.

As can be seen in Figure 13, the optimal alignment score is 0xE. The highlighted cells correspond to the path that a trace-back would follow. Horizontal and vertical shifts insert gaps into the alignment. Figure 14 shows the alignment solution after the trace-back through the solution matrix.

Smith-Waterman. The other alignment algorithm, Smith-Waterman, is considered a *local* sequence alignment algorithm in that it is used to identify highly

	G E T / i n d e x . h t m l										H T T P / 1 . 1									
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
E	0	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
T	0	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	0	1	2	3	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
/	0	1	2	3	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6
H	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	7	7	7	7
T	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8	8	8	8
T	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8	9	9	9
P	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8	9	A	A
/	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8	9	A	B
1	0	1	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8	9	A	B
.	0	1	2	3	4	5	5	5	5	5	5	6	6	6	6	7	8	9	A	B
1	0	1	2	3	4	5	5	5	5	5	5	6	6	6	6	7	8	9	A	B

Figure 13. Example Needleman-Wunsch solution matrix [Bed04a]

G	E	T		/	i	n	d	e	x	.	h	t	m	l		H	T	T	P	/	1	.	1
G	E	T		/	-	-	-	-	-	-	-	-	-	-		H	T	T	P	/	1	.	1

Figure 14. Alignment obtained from trace-back of previous solution matrix [Bed04a]

similar subsequences within a pair of sequences [SW81]. It is simply a variation of the Needleman-Wunsch algorithm where any negative scoring cells are set to 0 and the trace-back stops at the occurrence of a cell with a value of 0. Note, that the simplified HTTP example shown previously would yield the same results with Smith-Waterman as Needleman-Wunsch. However, if the gap or mismatch penalties were not 0, then the two algorithms would yield different solution matrices.

Multiple Sequence Alignment. While Needleman-Wunsch is an efficient algorithm for aligning two sequences, it is not efficient for aligning multiple sequences. In fact, if a large number of N sequences of length L were aligned, then deriving the solution would involve traversing a hypercube. This solution is NP-complete and $O(L^N)$ time [WJ94, Bed04a]. However, a number of heuristics have been developed that can estimate the optimal solution in a time-efficient manner. The heuristic of most interest for this research is the progressive technique often called the hierarchical method. This technique first creates a binary guide tree of pairwise alignments using a technique such as UPGMA. Then the MSA solution is gradually built up by globally aligning the most similar sequences and ending with the most distantly related sequences. After the end of each stage, the aligned string becomes the reference to align the next sequence. Care must be taken when using this method to ensure that the sequences are not too divergent, because any errors (i.e., gaps) in one stage's alignment are propagated throughout the rest of the MSA [Bed04a]. This is a tradeoff between efficiency and accuracy. Another heuristic for performing MSA involves using Hidden Markov Models described next.

2.2.7.3 Markov models.

Markov Chains represent finite random processes where a future state depends only on the present state [WBC10]. The probabilities of the state transitions are speci-

fied in a transition matrix. Hidden Markov Models (HMMs) are Markov Chains which include unknown internal states that can only be inferred by observation [Rab89]. That is, the probability of a given hidden state (and the probability of a given output) can be determined. In addition to the state transition matrix, HMMs have a symbol emission matrix which yields the probability distribution of outputs for a given state. One of the challenges with using HMMs lie in choosing appropriate model parameters such as the number of states and number of distinct observations per state. Additional algorithms such as Baum-Welch can be employed to find optimal parameters [WBC10]. Figure 15 shows an example of a HMM which could account for the results of a hidden coin toss experiment. In this experiment, described in [Rab89], it is unknown how many coins were tossed to yield a series of given heads or tail observations.

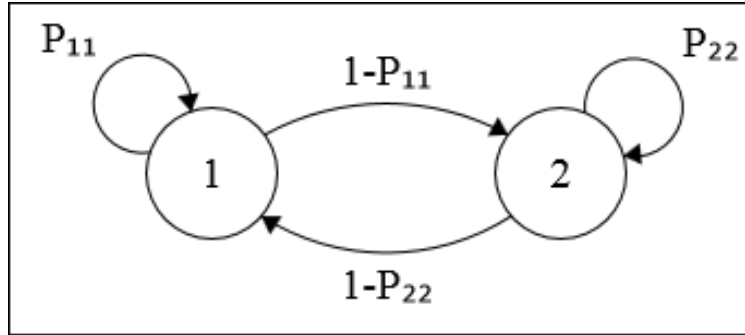


Figure 15. Two-coin Hidden Markov Model for hidden coin toss experiment [Rab89]

2.2.7.4 Other algorithms.

This final algorithm section includes common algorithms that are relevant to this research, but are not necessarily common to reverse engineering protocols.

Weighted interval scheduling is yet another common dynamic programming algorithm. The algorithm finds the subset of non-overlapping jobs that yield a maximal weight [KT06]. Each job, j , is assigned a starting time (s), a finish time (f),

and a value (v). The basis for the algorithm hinges on computing p -indexes. After the jobs are sorted by finish times, $p(j)$ corresponds to the largest job index, $i < j$, where job i is compatible with job j [KT06]. Figure 16 shows an example of eight jobs sorted by finish times. In this example, $p(7) = 3$ and $p(3) = 0$.

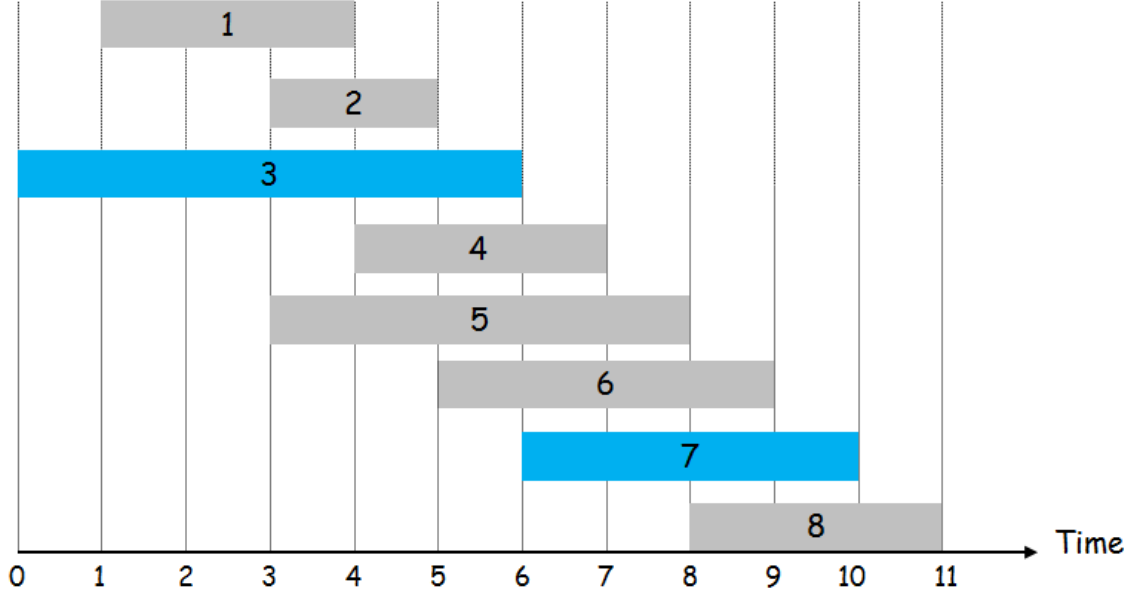


Figure 16. Example schedule (highlighted blue) determined by Weighted Interval Scheduling [KT06]

The algorithm then simply consists of finding the optimal set of binary choices [KT06]:

Case 1 M selects job j

- $M = v_j + M[p(j)]$

Case 2 M does not select job j

- $M = M[j - 1]$

Algorithm 2 shows the pseudo-code to compute a solution for the weighted interval scheduling problem. Algorithm 3 then shows how to trace-back through the solution

matrix in order to derive a schedule. In the previous example shown in Figure 16, the optimal schedule (highlighted in blue) yields a total weight of 10. This assumes each job's value is equivalent to its length. In the general case, values can be arbitrarily assigned.

Algorithm 2 Weighted Interval Scheduling (bottom-up approach) [KT06]

Input: $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \dots \leq f_n$.

Compute $p(1), p(2), \dots, p(n)$

function ITERATIVE-COMPUTE-OPT

$M[0] = 0$

for $j = 1$ to n **do**

$M[j] = \max(v_j + M[p(j)], M[j - 1])$

end for

end function

2.3 Related Research

2.3.1 Manually configured ICS honeypots.

Several ICS honeypot projects have been created in the past. Most either required real PLCs to act as HI honeypots, or required substantial effort in order to create believable emulators.

Digital Bond provides a SCADA honeynet at <http://www.digitalbond.com/tools/scada-honeynet/>. Their system uses a virtual Honeywall instance (described in Section 2.2.6.1) as the gateway to their honeypot network. Honeyd provides LI SCADA honeypots with File Transfer Protocol (FTP), HTTP, and Modbus services. Alternatively, a real PLC can be employed.

Algorithm 3 Weighted Interval Scheduling Find Solution [KT06]

```
function FIND-SOLUTION( $j$ )  
    ITERATIVE-COMPUTE-OPT  
    FIND-SOLUTION( $n$ )  
    if  $j = 0$  then  
        output nothing  
    else if  $(v_j + M[p(j)] > M[j - 1])$  then  
        print  $j$   
        FIND-SOLUTION( $p(j)$ )  
    else  
        FIND-SOLUTION( $j - 1$ )  
    end if  
end function
```

Next, as previously discussed in Section 2.2.4, Kyle Wilhoit deployed ICS honeypots at various worldwide locations [Wil13b, Wil13a]. The first experiment included a Dell server with PLC web server software to act as an HMI, a real ‘Nano-10’ PLC web server to mimic a temperature controller, and a Honeyd VM deployed on Amazon Cloud [Wil13b]. The Honeyd VM was made to look like a PLC with HTTP, FTP, and Modbus interfaces. The second experiment had a similar set of services, but web pages were updated with localized text [Wil13a]. Additionally, the Browser Exploitation Framework was used to aid in attribution by injecting active scripts into attacker’s machines who attempted to access a “secured” web page. The scripts provided remote identification and other features.

ICS honeypots have also been implemented on embedded hardware. In 2013 Jaromin emulated a Koyo DirectLOGIC 405 PLC on a Gumstix single board computer with an embedded distribution of Linux called Angstrom [Jar82]. The honeypot behavior was implemented by creating custom firewall rules, modifying TCP settings, and creating Python service scripts. HTTP and Modbus were implemented.

Another researcher created a honeypot to mimic a Siemens Simatic 300 using a customized Ubuntu VM [BJM⁺14]. The project, called CryPLH, implemented HTTP, HTTP Secure (HTTPS), Simple Network Management Protocol (SNMP), and Siemens' alternate TCP implementation called ISO-TSAP. As with the previously described project, emulator behavior was implemented by using custom firewall rules and TCP settings. MiniServer was installed to serve HTTP, and Nginx was installed to provide HTTPS. Some web pages, such as the login page, were manually edited. The SNMP agent used information from `/proc` and records previously extracted from a real PLC. Finally ISO-TSAP was implemented in Python.

The last project of interest is an open-source project called Conpot located at <http://conpot.org/>. They have released service scripts that can be used to implement Modbus and SNMP. By default, Conpot emulates a Siemens SIMATIC S7-200 PLC with an input/output module and a CP 443-1 (communications module).

2.3.2 Dynamic honeynets.

There are several honeypot projects that are automatically configured, but they only automatically configure the network and transport layers. The application layer traffic, if handled, is done by pre-configured VMs. Their primary purpose, then, is to essentially create a dynamic honeynet that can cover large address spaces and handle traffic from network scanning tools.

One project, for example, aimed to capture IPv6 address scans [KOY⁺12]. It relied on the assumption that media access control (MAC) addresses are often used

to generate IPv6 addresses. When an adversary scans for a particular IPv6 address, the network router will send out neighbor solicitation (NS) messages to find out which local machine has that address. The honeypot system then extracts the MAC from the NS and selects a pre-configured VM whose vendor and model matches the MAC. The corresponding requested addresses are then configured on the VM via a secure shell (SSH), and the machine sends out a neighbor advertisement (NA) to respond to the router's original request.

The remaining dynamic honeynet research all use Honeyd for their emulation, so the auto-configuration then just requires automatic handling of the Honeyd configuration files. The primary difference between the various implementations is based upon whether they use active sniffing, passive sniffing, or a combination of the two in order to obtain configuration information. Active sniffing injects packets into the network in order to solicit a response, so those methods can usually obtain information faster. Passive methods, on the other hand, listen only and are thus more stealthy.

As an example, Hieb used `p0f` and `tcpdump` for passively detecting Operating Systems (OSs) and ports in order to combine a dynamic honeynet with Snort IDS [Hie04]. Hecker, Nance, and Hay, instead used `Nmap` to actively obtain information [HNN06]. The honeynet developed by Kuwatly, Sraji, Masri, and Artail used both methods interchangeably [KSAMA04].

This concept was later extended by Hecker and Hay's dynamic honeynet that defined various noise factor levels [HH13]. Five increasingly active levels were defined based on the combination of tools used for sniffing. At the lowest level, only `p0f` and `tcpdump` were to be used. `xprobe2` was then additionally employed with increasingly aggressive options for the next two levels in order to obtain OS and port information, respectively. At the medium-high setting `Nmap` was used alone. Finally, `Nmap` and `xprobe2` were used together for the highest noise factor level. All of the

noise factor levels were tested, and it was discovered that the active scanning tools often degraded the p0f's performance. Finally, another unique feature of Hecker and Hay's research is that they used Extensible Markup Language (XML) files for representing honeypot configuration information.

XML is a powerful markup language because it provides a standard format for various software to interpret information. This is the reason why Vollmer and Manic used XML in their dynamic ICS honeynet [VM14]. Instead of using the tools mentioned previously, their research used Ettercap to passively scan the network and generate XML output files. This allowed them to automatically identify each device's OS, ports, and MAC address. The identified ports were matched up against against common OS ports in order to select the most probable OS version for their VM configuration. The MAC address could also be used to select the OS. Next, by identifying the hardware vendor from the MAC, a common set of ports, protocols, and services could be selected for the honeypot configuration. Future changes to the network configuration could be detected by the differences in the XML files. The required VM changes would then be done dynamically using Honeyd's Honeydctl interface. This dynamic ICS honeynet was later integrated into Vollmer, Manic, and Linda's Automatic Intelligent Cyber-Sensor (AICS) [VML14].

The AICS combined a fuzzy logic driven IDS with the dynamic honeynet described previously [VML14]. The components were integrated by a two-tier communication scheme. External communication was done using XML messages over HTTPS, while internal communication was accomplished via interprocess communication (D-Bus). This communication scheme provided a modular design where sensor components would be isolated from direct network contact with other devices. By shadowing an existing ICS network, the honeynet provided a large attack surface to attract anomalous traffic. Finally, the IDS's sensitivity level would automatically adjust based upon

traffic trends. This tightly integrated hybrid network sensor system identified 99.9% of anomalous packets generated during experimentation.

2.3.3 Automatically configured emulators using protocol knowledge.

While the research in Section 2.3.2 focused on dynamically instantiating honeynets, this group of research dynamically creates application layer emulators based on *a priori* protocol knowledge. For example, Jiang and Xu developed a system of “catering” honeypots called BAIT-TRAP [JX04]. Essentially, a set of desired services was determined based upon destination ports found in network traffic. A template was then selected from a set of VM images. Additional Linux services could be provided automatically using package managers.

Another project called Adaptive Honeypot Alternative (AHA) created a self-learning SSH honeypot [WSED11]. AHA used reinforcement learning techniques, combined with an abstract state machine and actions such as command substitution, in order to maximize attacker interaction.

Next, Chowdhary, Tongaonkar, and Chiueh used a modified version of classical data mining techniques they called “service mining” in order to learn FTP [CTC04]. Their approach requires an initial database of protocol keywords and arguments. Then, collected network traces are broken down into sets of interaction that are correlated to the keywords.

Another text-based protocol emulator was developed in 2014 by Fink for ICS honeypot research [Fin90]. The web interface of a Koyo DirectLogic 405 PLC was emulated, along with partial emulation of TCP behavior. `Wget` was used to automatically mirror the PLC’s web site while capturing the traffic using `tcpdump`. To partially emulate transport layer behavior, several TCP parameters such as time to live (TTL), timestamps, and window scaling were extracted from the packet capture

and exported to the appropriate Linux kernel interfaces. Finally, during replay a client request dictionary was used to select the appropriate HTTP response header. The combined header and corresponding HTML were then sent back to the client via a Python web server.

ICS emulators have also been used previously for firmware verification. In 2012 McMinn created a tool to verify Allen-Bradley PLC firmware [McM00]. This was done by first capturing pairs of firmware upload traffic. Each reference pair of captured bytes were then lined up and compared in order to identify stateful protocol fields. This process was aided by a set of heuristics based on knowledge of the DF1 protocol. After the emulator was trained, firmware could then be validated by checking future upload traces against the trained set in order to identify any significant differences.

A similar technique has also been used to verify network traffic. In 2014 researchers developed network dialog difference measures that are used for HTTP traffic clustering [RCHJ14]. Traffic clustering, like other forms of clustering, determines what objects (messages in this case) belong to what group. As such they are very useful for generating IDS signatures. The distance measures were created by comparing the request-response pairs (RRPs) between two traces. Similarity scores were computed based on same request type, URL path, content length, and other HTTP protocol fields. The messages were then aligned by treating the trace pair as a weighted bipartite graph where the goal is to maximize weight (similarity) through unique assignments. Total dialog similarity was then computed by dividing the total similarity weight by the number of nodes.

2.3.4 Reverse engineering protocols.

2.3.4.1 Overview.

As previously discussed in Section 2.2.7, reverse engineering protocols involves inferring message formats and the proper order in which those messages can be used (a state machine). It is important to note, however, that the degree to which this must be done depends on how the output will be used. Pure protocol reverse engineering programs attempt to define a protocol well enough to create a packet dissector, while honeypots only need to understand a protocol well enough to accurately mimic previous conversations [BGH14]. Thus, the computational complexity and/or number of traces required for honeypots could potentially be less than those of pure reversing programs.

What follows is a survey of 21 tools that reverse engineer protocols. Nearly all of them use network traces to infer formats, but many of them also use dynamic taint analysis (DTA) of execution traces described in Section 2.2.6.1. The tools use various methods for aligning (by byte, token, or n -gram probability) and clustering (positional, distance, action, etc) messages. In Chapter III it is shown how the techniques used by some of these tools can be leveraged to create a flexible framework for developing protocol-agnostic honeypots.

2.3.4.2 Reversing engineering protocols using program binaries.

In 2006 the research team behind Replayer showed how the traffic replay problem could be represented concisely as a set of solvable equations [NBFS06]. Solving them required a program execution trace along with assembly code of the network program. Additionally, the client’s initial state must be known either through direct knowledge or by inference of traffic session fields. Unfortunately, the derived solutions (i.e.,

server responses) could not be guaranteed to always be found. Additional constraints also have to be enforced to ensure that the solution has correct static fields.

The remaining tools all use some form of DTA in order to reverse engineer protocols. For instance, a typical method for finding field delimiters is to look for program operations that compare a tainted input byte with an untainted value. Similarly, field keywords can be inferred by finding comparisons of a string of tainted bytes against a string of untainted bytes. Bytes that lead to file accesses are probably filenames. Finally, finding length fields involves searching for comparisons that terminate loops, and padding can be detected by identifying unused bytes.

Polyglot is the first published tool to use DTA techniques such as above [CYLS07]. A year later AutoFormat improved on those techniques by identifying hierarchical message formats [LJXZ08]. The key insight was that bytes being used in the same execution context (e.g., function) are most likely a part of the same field. However, this does not account for functions that parse multiple fields at a time. Concurrently, other research developed DTA techniques fine grained enough to produce a parsing grammar [WCKK08]. Hierarchical delimiters were discovered and then extended byte by byte to identify multiple byte delimiters (e.g. ‘\r\n’). Additionally, bytes that were found in both client requests and server responses were semantically linked. Finally, messages across the trace were correlated by aligning them with Needleman-Wunsch (weights 1 for a match, -1 for a mismatch and gap). This allowed messages to be generalized using regular expressions.

In 2009 the same research team created Prospex that went one step further by inferring a message format and SM [CWKK09]. In this work, distance matrices were created using various metrics. First Needleman-Wunsch scores were calculated on aligned fields. Next a Jaccard index (intersection of similar elements divided by union of similar elements) was calculated on five execution metrics: system, function, and

library calls, process activity, and executed instruction addresses. These similarity scores were then used to cluster via PAM. Finally, the SM, which was initially created in the same order as seen in the traces, was minimized by recognizing patterns in their sequences and using the Exbar algorithm to merge equivalent states.

Researchers have also developed tools to reverse encrypted protocols. ReFormat [WJC⁺09] and Dispatcher [CPKS09, CS13] both read encrypted data while it is unencrypted in memory buffers. Dispatcher is able to handle more complex encryption schemes, however [CS13]. Additionally, Dispatcher is unique in that it does not use any network traces. Instead, it derives message fields by discovering how they are divided into various memory buffers [CPKS09, CS13]. This is accomplished first by monitoring socket calls to divide execution traces into a set of messages. The message fields are then correlated to the buffers that make up the network buffer. Finally, DTA is used to determine the semantic meaning of the fields and how they link between client and server messages.

The last reversing program that uses application binaries is Tupni [CPC⁺08], which uses DTA along with the tokenization and recursive clustering techniques of Discoverer (described in the next section).

2.3.4.3 Reversing engineering protocols using only network traces.

One of the first attempts at protocol reverse engineering using network traces was the work by Beddoe in 2004 [Bed04a]. Protocol Informatics (PI) used UPGMA clustering based on Smith-Waterman distance matrices in order to create guide trees for Needleman-Wunsch MSA. Consensus statistics (most common byte, most common data type, mutation rate) were then performed on each aligned byte in order to infer message formats.

While PI's inferred message formats provide only basic semantic meaning, Script-Gen extended them using *Region Analysis* in order to create protocol-agnostic hon-

eypot scripts for Honeyd [LMD05]. ScriptGen first builds an initial finite state machine (FSM) by creating a branch for every session from a packet capture. This tree is then generalized by using PI to cluster based on local alignment scores and regional mutation rates. Clusters are byte-aligned via MSA and regular expressions are created to match against incoming client requests during replay. The second iteration of ScriptGen included improved dependency handling between messages (i.e., sessions IDs) and between sessions [LDM06]. Additionally, proxy support was added to allow ScriptGen to incrementally update itself based upon responses from real devices.

Roleplayer is another protocol-agnostic honeypot that uses some of the algorithms from PI [CPWK06]. Specifically, Needleman-Wunsch is used (weights: 1 for match and -1 for gap) to align a pair of sequences by blocks and find differences. The blocks are created by splitting sequences up by tokens of found end-point addresses and *known* arguments. A set of heuristics based on common protocol idioms are used to determine if certain blocks contain length fields or cookies. Smith-Waterman is used during replay to decide when received data consists of a complete message. The main drawback to Roleplayer’s approach is that it can handle only one scripted session at a time, and the client must not deviate at all from that scripted session.

Next, Advanced Protocol Reversing Tool (APRT) combines some of the techniques from both ScriptGen and Roleplayer [PHDW12]. Similar to ScriptGen, APRT also uses PI’s MSA to align sequences. Unlike ScriptGen’s byte alignments, APRT aligns by ASCII and binary blocks to make the process more efficient. Field identification is then determined by successive bytes with similar variability coefficients (number ASCII bytes divided by number sequences). Finally, length and other fields are inferred using Roleplayer’s heuristics.

Other research aligns sequences based upon tokens. For instance, Discoverer creates tokens out of common delimiters (e.g., ‘SPC’ and ‘TAB’) and clusters based on the

order of tokens [CKW07]. Similar message formats are then merged after recursively refining length and other fields using Roleplayer’s heuristics. Certain discriminating tokens are used to separate formats. ReverX is another example that uses delimiters in order to infer message formats [ANV11]. However, it also builds a SM while parsing a network trace and later generalizes it using transition frequencies. Since both Discoverer and ReverX use text delimiters, their approaches are most effective for text-based protocols.

Another group of research uses more probabilistic methods in order to reverse protocols. All of these use n -grams, which are simply permutations of byte sequences of n length. Researchers at Johns Hopkins University, for instance, developed a LI web honeypot using n -gram probabilities of various space, text, and binary tokens [SMM⁺08]. k -medoids were used to cluster and a Markov model was generated using probabilistic simulations. Additionally, global message dependencies were determined by Needleman-Wunsch aligning client messages to their respective server responses. Bytes that were the same in 80% or more of messages were globally linked across all message pairs.

Next, another pair of projects gives the user a choice between using tokens and n -grams (optimal for text or binary protocols, respectively) rather than combining them [KKR11, KGKR12]. Frequency distributions are then calculated and the extremes are filtered out. Keywords with low variability are likely fixed fields, and keywords with high variability are likely random data. The reduced set is then clustered via NMF [KKR11]. This work was extended in the open-source Protocol Inspection and State Machine Analysis (PRISMA) project [KGKR12]. PRISMA added a honeypot replayer, performed additional clustering by keyword order, and inferred a SM. The SM was created by building a Markov model based on clustered transition sequences and then reducing it to a HMM by aligning messages with equivalent states and number of tokens.

Veritas also uses tokens and n -grams [WZY⁺11]. It initially filters based on the sample distances of an empirical distribution function. Then a Jaccard index is used as distance input into PAM for clustering. Next, the order of the states and their transition probabilities infer an initial SM which is then minimized.

The final probabilistic protocol reverser is ProDecoder [WYS⁺12]. It is fully automated and uses n -gram frequency distributions as input to a Gibbs sampling (Markov chain simulation) algorithm to identify keywords. Message clustering is performed using the Information Bottleneck method, and then MSA is performed on the clusters to identify fields.

In addition to the previous methods, genetic programming has been used for protocol reversing [LBZH13]. Unfortunately, NimsGP requires a protocol-specific instruction set in order to properly minimize the SM.

Finally, the open-source Netzob infers message formats and state machines by using a variety of the previous techniques plus some unique ones [BGH14]. For inferring message formats Netzob first requires that the input trace have different tasks identified. This is either done manually or by using execution monitoring. The sessions are then split into a series of tasks, and messages with no associated task of interest are used to filter out background noise. Next, messages are tokenized and clustered using methods similar to Discover, and contextual information (IP addresses, etc) is extracted from the trace. Every half-byte of the tokenized message sequences are tagged to identify semantics. These tags are used to weight scores in extended forms of Needleman-Wunsch and UPGMA. Half-bytes with the same value and tags are then merged. Finally, linear and non-linear dependencies are exhaustively determined.

2.3.4.4 Summary.

In summary, there are a variety of techniques for reverse engineering protocols. Most of them use network traces, but many of them also use DTA on application binaries. Using binaries provides more accurate measures for reversing the protocol, but protocol application binaries are not always available. Often clients and servers also use different versions. Trace-based approaches, on the other hand, have the limitation that their inference is limited by the quality of the traces. Thus, long or multiple traces are often required. This is particularly true of the methods that rely on statistics (e.g., n -grams) to identify important keywords. Finally, it has been shown that there are a variety of clustering and alignment techniques that are used at varying levels of abstraction (bytes, tokens, blocks, messages, and tasks).

2.3.5 Putting it all together: Hybrid honeypots using smart replay.

This final section of related research gives a brief overview of advanced hybrid honeypots that integrate protocol-agnostic replay with proxying and self-training. All of them use either Roleplayer or ScriptGen for replaying protocols in order to create a malware analysis framework.

GQ was the first advanced hybrid honeypot, and it was originally designed for tracking worm propagation [CPW06]. GQ uses Roleplayer for its LI replay, and its architecture uses a central gateway to create a virtual infected machine network of *inmates*. VMs are managed using a VM monitor. As in the other examples from Section 2.2.6.4, traffic forwarding and other manipulations (filtering outgoing connections, etc) are done by the gateway. A newer iteration provided better containment policies and allowed for handling other types of malware [KWK⁺11].

Unlike GQ’s centralized architecture, SGNET provides a *distributed* deployment that handles worms and server-based code injection attacks [LD08]. SGNET uses

ScriptGen as its LI replayer, and its architecture is similar to that of Honeybrid described in Section 2.2.6.4. In particular, SGNET uses different backend honeypots for malware handling. Argos collects and contains malware samples or forwards code injection attacks to the exploit modes in nepenthes. ScriptGen acts as a general vulnerability module for nepenthes, and it uses the traffic generated by Argos to update its configuration. Additionally, SGNET integrates with Anubis and VirusTotal for malware analysis. Finally, a unique feature of SGNET is that its distributed architecture allows partners around the world to deploy sensors using a common VM image. Traffic found by the sensors are used to update the FSMs of other distributed nodes.

Variations of SGNET have also been used in two other advanced hybrid honeypots [GLB12]. Mozzie extended the containment capabilities of SGNET by using ScripGen to model both sides of a malware conversation. This way command and control traffic, for instance, would not have to leave the network in order to allow for full exploitation. Instead, ScriptGen is used to model the victim in one virtual instance and the master controller in another virtual instance. Finally, AWESOME, which stands for Automated Web Emulation for Secure Operation of a Malware-Analysis Environment, uses ScriptGen to power its replay [FB13]. Like the other projects, Argos is used for malware collection. AWESOME, however, uses external script handlers from Honeyd and Nitro for malware analysis.

2.4 Chapter Summary

This chapter shows that the United States depends on ICS for providing its Critical Infrastructure. However the devices, such as PLCs, that make up these control systems are vulnerable by design. In order to defend them from adversaries a new state-based paradigm must be employed along with advanced network sensors using honeypots.

Honeypots have many variations that can cater to most security needs. By using a creative mix of LI and HI honeypots a scalable, yet high-fidelity honeynet can be created. Additionally, tools such as Honeyd can be used to rapidly deploy dynamic honeynets. However, emulating proprietary protocols used by PLC vendors is difficult and prevents researchers from being able to quickly develop effective ICS honeypots.

Protocol reverse engineering techniques have been shown to be effective in automatically learning and emulating many protocols. GQ and SGNET makes use of these technologies in order to deploy advanced hybrid honeypots for catching malware.

Unfortunately these tools have not been publicly released, nor have their effectiveness in ICS networks been published. Thus, there is a need to develop an open-source protocol-agnostic honeypot that is proven to be effective with ICS protocols. The design of this honeypot discussed in Chapter III requires leveraging the right combination of protocol reverse engineering techniques discussed in this chapter.

III. Framework Design

3.1 Overview

This chapter provides a detailed account of the framework design for automatically configuring PLC emulation. The framework design parameters require the resultant product to be accurate, flexible, and efficient. After careful consideration of the various protocol reverse engineering techniques discussed in the previous chapter, a design based on ScriptGen is selected. Other previously discussed techniques are also used, along with several novel contributions of this research. These contributions include a *backtracking* algorithm that can handle unknown transitions and enables *session looping*, a default response mechanism, and a set of algorithms (inspired by ScriptGen) to determine and filter protocol dependencies.

The resultant ScriptGenE Framework includes several modular components that can be used to generate, manipulate, replay, and assess protocol FSMs.

3.2 Design Parameters

The following are the design parameters (organized by category) that were considered while creating the framework:

Accuracy:

1. Emulator produces traffic that can fool industry standard tools for interacting with PLCs (e.g., RSLogix)
2. Application layer responses are not statistically different than real PLC responses for a given conversation/task

Flexibility:

1. Protocol-agnostic (No assumptions about protocol type e.g., text, binary, mixed)
2. Able to provide repeated patterns of short responses for a session of indefinite length in order to emulate ICS device polling
3. Able to provide a response that is acceptable by the client even when the current request is unknown or out of order according to the replay script
4. Replay as stand alone or with Honeyd integration
5. Effective as stand alone or future integration in hybrid honeypot
6. Extensible design allows for integrating future enhancements

Efficiency:

1. Configuration and emulation can be automated
2. Requires few traces (1-7) to produce an accurate emulator
3. Can build emulation script within reasonable amount of time (five minutes)

3.3 Technique Suitability Evaluation

Next, the various protocol reverse engineering tools and techniques discussed from the previous chapter are considered. Netzob's published comparison study of Automatic Semantics-aware Analysis of Network Payloads (ASAP), Discoverer, ScriptGen, and Netzob is also considered [BGH14]. While the study only includes a limited number of tools, the selection does cover different technique classes (probabilistic, delimiters, positional clustered and byte-aligned).

First, in the interest of flexibility and requiring complete automation, approaches only requiring network traces are considered. Requiring a binary in order to perform DTA would limit the tools to only be able to learn directed (i.e., manual) tasks.

Second, approaches that depend primarily on probabilistic methods are removed from consideration since they require a significant number of traces to be accurate. Additionally, the Netzob team’s findings show that approaches that rely only on statistical analysis of n -grams identify less than 20% of the correct message formats [BGH14]. PRISMA and ASAP also have the issue that the keyword identification approach (n -gram or token) and the clustering approach must be manually selected (Some heuristics, however, may be able to do this automatically).

Third, approaches using delimiters are only effective when the protocols contain the delimiters. Discoverer, for example, makes assumptions about text delimiters that would not be appropriate for binary protocols. Higher level tokens would potentially be useful, but they would have to be carefully defined so as to not only benefit reversing text-based protocols. At a higher level of abstraction, identifying fields by binary or ASCII blocks as APRT did is only beneficial for mixed protocols [PHDW12]. Regarding APRT, it is unclear how fields would be identified for binary protocols since identification depends on differing ratios of ASCII bytes between aligned bytes.

PI and Netzob are particularly attractive since they are both open-source [Bed04b, BGH13]. Both have been tested for suitability. PI requires careful trace filtering and a lot of packets in order to infer a good enough message format for replay. Netzob provides a graphical user interface (GUI) to select different methods to parse sequences. It will also replay based on client inputs once a SM has been created. Overall, Netzob is determined to be good for “semi-automatic” protocol reverse engineering, but is not suitable in its current form for automatic learning and replay.

Roleplayer and ScriptGen, however, stand out from the other projects since they have been used in hybrid honeypots as described in Section 2.3.5. Neither project can handle unknown transitions during replay without the help of a proxy. Roleplayer is particularly limited in that it can only emulate one task (i.e., script) at a time. ScriptGen cannot dynamically update environmental information, and it requires more traces than Roleplayer due to the statistical variety (regional mutation rates) required for its microclustering. In the Netzob team’s analysis ScriptGen is found to be very accurate, but not concise [BGH14]. Accuracy is aided by the fact the messages are positionally clustered, but this is also the reason why it does not concisely represent a protocol. Messages arriving in a different order lead to new transitions. However, concisely representing a protocol is not crucial for a honeypot, while accuracy is important.

The selected solution for this research is to use a mix of techniques from ScriptGen and Roleplayer, while introducing some new novel concepts to fill the requirement gaps. As such, the chosen design is primarily based on ScriptGen. In reference to this, the framework is named *ScriptGenE* (pronounced ‘script genie’), which stands for ‘ScriptGen Enhanced’. Additionally, some code from PI is reused (alignment and phylogenetic tree creating algorithms). In order to clearly delineate PI and ScriptGen from the current design, a more in depth introduction is provided.

3.4 Protocol Informatics In-depth

The PI Framework provides a set of distance functions (entropic, pairwise, and local alignment), alignment functions (Smith-Waterman, Needleman-Wunsch, and MSA), and phylogenetic tree creation via UPGMA clustering. The framework was developed in Python/Pyrex (C extensions to Python similar to Cython).

The provided main script takes a packet capture as input, and it outputs byte sequences representing each message cluster. First, PI parses all packets from an input trace and creates distance matrices using Smith-Waterman (weights 2 for match, -1 for mismatch, and -2 for gaps). These matrices are used to create guide trees using UPGMA, which are then in turn used to perform MSA using Needleman-Wunsch (weights 1 for a match and 0 for all else). Each cluster of aligned sequences is then shown in a colorful output display.

An example of the output that PI provides for an aligned cluster is shown in Figure 17. Each numbered line represents a different aligned sequence of hexadecimal bytes. In the figure sequence labels can be seen on the far left, and the most common data types (DTs) and mutation rates can be seen on the bottom two rows. Mutation rates are defined as

$$\frac{\# \text{ unique bytes}}{\text{total bytes}} \quad (3)$$

The colors are based on the DT of each byte: ASCII values ('AAA') are colored green, zeros ('ZZZ') are magenta, gaps ('GGG') are red, spaces ('SSS') are yellow, and other bytes ('BBB') are white. In addition to what is shown in the figure, an *ungapped consensus* is provided (guessed message format). Instead of listing each byte as shown in Figure 17, the consensus sequence simply lists the most common byte (MCB) for each byte or '???' if there is not one. Any bytes where the MCB is a gap are omitted from the consensus sequence.

3.5 ScriptGen In-depth

While PI's inferred message formats provide only basic semantic meaning, ScriptGen extends them using *Region Analysis* in order to create protocol-agnostic honey-pot scripts for Honeyd [LMD05, LDM06]. ScriptGen accomplishes this in four main parts.

Output of cluster 1																		
0020	x6f	x00	x26	x00	x00	x01	x02	x10	x00	x00	x00	x00	x16	x00	x00	x00	xe0	x50
0004	x64	x00		x02	x00	x00	x00	x00										
0001	x04				x00	x00	x00	x00										
0002	x64				x00	x00	x00	x00										
0005	x65	x00	x04	x00	x00	x00	x00	x00										
0006	x65	x00	x04	x00	x00	x01	x02	x10										
0014	x6f	x00	x28	x00	x00	x01	x02	x10	x00	x00	x00	x00	x13	x00	x00	x00	xe0	x50
0010	x6f	x00	x24	x00	x00	x01	x02	x10	x00	x00	x00	x00	x0e	x00	x00	x00	xe0	x50
0026	x6f	x00		x00	x00	x01	x02	x10	x00	x00	x00	x00	x19	x00	x00	x00	xe0	x50
0024	x6f	x00	x18	x00	x00	x01	x02	x10	x00	x00	x00	x00	x18	x00	x00	x00	xe0	x50
0018	x6f	x00	x18	x00	x00	x01	x02	x10	x00	x00	x00	x00	x15	x00	x00	x00	xe0	x50
0028	x6f	x00	x2e	x00	x00	x01	x02	x10	x00	x00	x00	x00	x1a	x00	x00	x00	xe0	x50
0022	x6f	x00	x2a	x00	x00	x01	x02	x10	x00	x00	x00	x00	x17	x00	x00	x00	xe0	x50
0016	x6f	x00	x2a	x00	x00	x01	x02	x10	x00	x00	x00	x00	x14	x00	x00	x00	xe0	x50
DT	AAA	ZZZ	AAA	ZZZ	ZZZ	BBB	BBB	BBB	ZZZ	ZZZ	ZZZ	ZZZ	BBB	ZZZ	ZZZ	ZZZ	BBB	AAA
MT	028	014	064	021	000	014	014	014	014	014	014	014	071	014	014	014	014	014

Figure 17. Message cluster output from PI

First, a packet capture is parsed and all TCP segments with message payloads are stored in an array with TCP sequence numbers being used as the index [LMD05]. Using the sequence numbers handles out of order messages. Next, the array is used to build an initial FSM where each edge label represents a client request and each node label represents a server response. Consecutive message flows in one direction are consolidated. Each TCP connection from the input trace represents a different branch of the FSM, where the number of nodes in the branch is the number of server messages in the session. All branches are initially connected at the root, forming a simple tree.

The third step is where ScriptGen generalizes the FSM [LMD05]. In a breadth first search (BFS) visit of the tree, each node's outgoing edges are clustered/combined. Each cluster then becomes a singular branch with the edge label being converted to a regular expression that represents the group. The edge is also given a weight equivalent to the number of sequences that make up the group. This serves to identify the most frequent transitions in the FSM.

It should be noted that ScriptGen inherently clusters messages based upon their position in a session since only a node's outgoing edges are considered for clustering.

This makes the resultant emulator more accurate since it will expect and mimic sessions using the same order of messages that it has stored in its FSM. For example, Figure 18 shows two sessions that would be seen as separate branches in the FSM. The first session on the left side of the figure shows that the client sent two consecutive messages which were responded to consecutively by the server. In the second session on the right side of the figure, both the client and server send one message at a time. In this example, even though the client and server each sent their message in the same order, the messages are not in the same order from a third party’s perspective. Furthermore, since ScriptGen consolidates consecutive message flows in one direction, the consolidated message [cA, cB] would not be equivalent to message [cA] or equivalent to message [cB].

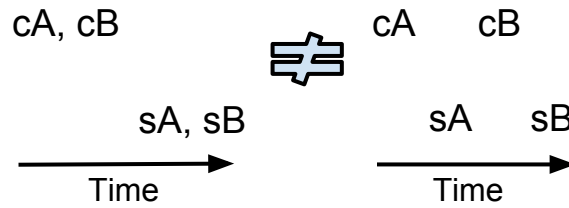


Figure 18. Sessions with different message order are not equivalent

In addition to the inherent positional clustering, ScriptGen uses two more rounds of clustering. The first serves to group major classes of messages (e.g., GET requests), while the second attempts to cluster based on specific types of messages (e.g., ‘GET *.png’ versus ‘GET *.html’). The initial clustering, or *macroclustering*, is done using PI [LMD05]. PI’s output is then used to perform *Region Analysis*. Regions are defined as bytes with the same type, similar mutation rates, the same “kind” of data, and the same gap presence [LMD05]. Additional clustering, described as *microclustering*, is then done by creating a distance matrix based on region-wide byte frequencies. The researchers’ insight is that frequent values probably have semantic meaning. Next, as a result of microclustering, previous *mutating regions* become *fixed*

regions. A regular expression is then generated that matches the order of regions and their type (i.e., mutating or fixed).

Regions are also used to determine dependencies, or *links*, from client to server messages within a session [LDM06]. This means that bytes from a client message are duplicated by the server in its response. These *intra-protocol links* include fields such as session IDs, cookies, etc. To identify the links, first a set of markers are identified in the client messages. These *significant fixed regions* only have one occurrence in the client message. All of the other bytes are checked for matches in the corresponding server response. A minimum of two bytes are used when considering matches up to the maximum size of the match.

Once links are identified, they are represented by a tuple consisting of the client request ID, starting and ending markers, and the marker offsets [LDM06]. Representations of these links then replace corresponding content in the server response. Once this is done for all of the server responses, the proposed messages are consolidated into one proposal that represents the cluster. This is done by aligning the bytes as shown in Figure 19 and picking the most frequent byte for every aligned byte. Any proposal that does not have the most frequent byte for the currently evaluated aligned byte is discarded from the proposal list. The chosen proposal (number 5 in Figure 19) must be equivalent to at least one of the proposals.

ScriptGen also includes two features that are not implemented in this research. First, ScriptGen detects another dependency type that Leita, Dacier, and Massicotte called *inter-protocol dependencies* [LDM06]. Inter-protocol dependencies refer to links between sessions such as a message in one session leading to an open port or other action in another session (e.g., FTP). Inter-protocol links are identified using simple heuristics based on the order of client and server interactions between groups of TCP connections.

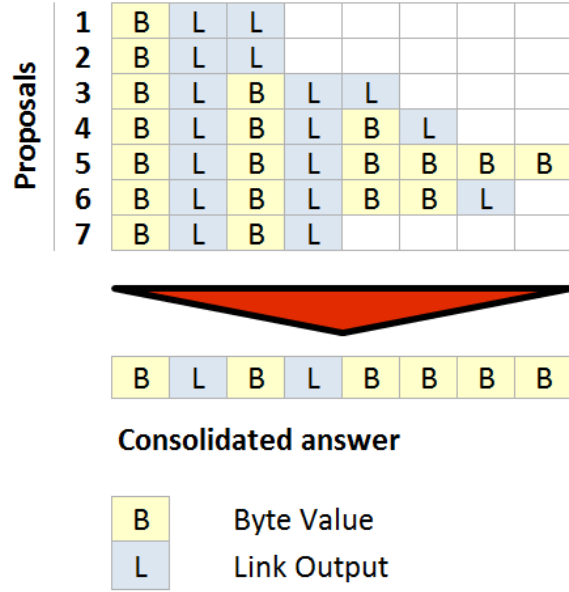


Figure 19. Link consolidation [LDM06]

Second, ScriptGen’s newer iteration also has the ability to hand off a session to a proxy if it does not know how to respond during replay. For details on this the reader is referred to [LDM06].

Finally, it should be noted that ScriptGen’s replay script is intended to be used only in Honeyd. The Python replay script sends responses by printing to `stdout` and receives requests by reading `stdin` [LMD05].

3.6 The ScriptGenE Framework

3.6.1 Framework overview.

The ScriptGenE Framework, or simply ScriptGenE, is developed in Python and consists of approximately 5,600 lines of code (not including third party libraries such as PI). The primary functions of the framework are generating, manipulating, replaying, and assessing protocol trees (P-trees). Figure 20 shows an overview of how the framework builds and replays P-trees. ScriptGenE.py builds P-trees by parsing a set of input traces according to given build options. The generated P-tree is then

imported by ScriptGenEreplay.py according to a set of replay options in order to emulate the network responses given in the tree. P-trees are ScriptGenE's FSM. Each node or edge represents a server or client message, respectively. Several attributes are embedded into these objects, some of which are described later.

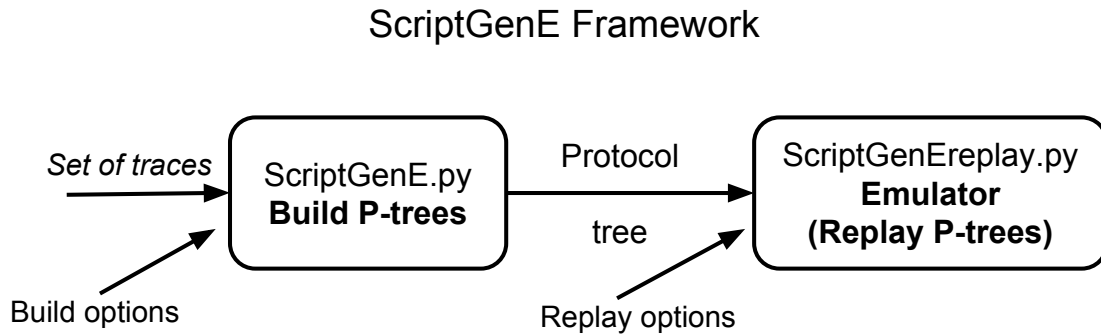


Figure 20. ScriptGenE Framework Overview

The ScriptGenE Framework includes other important scripts in addition to the two previously mentioned. The primary ScriptGenE Framework files and their functionality are:

ScriptGenE.py

- Primary script of the framework
- Parses packet capture (Pcap) files and builds message lists
- Builds initial and generalized protocol trees
- Outputs / logs report
 - Conversation summaries
 - Region Analysis output
 - Link summaries
 - P-tree build summary

GeneralizeTree.py

- Takes initial tree (ScriptGenE.py intermediate output) and generalizes it
- Outputs / logs report (same as ScriptGenE.py)

CombineGtrees.py

- Combines trees at ROOT and merges ROOT attributes

ScriptGenEreplay.py

- A server that loads a protocol tree and replays messages based on how the client messages it receives match edges in the tree
- Outputs progress
- Logs hexdump of client messages in INFO messages

clientReplay.py

- Replays client requests stored in a protocol tree by choosing random transitions along a path
- Replay can either stop after a tree leaf is reached or repeat from random nodes until a keyboard interrupt is received
- Server responses are ignored
- Can be used to test protocol trees locally or against a real server

diffPcaps.py

- Finds the number of different bytes in consolidated messages between two Pcap files and prints/logs those differences
- Differences in quantity or length of messages (not connections) are counted as different bytes

It should be noted that all of the ScriptGenE Framework scripts provide help instructions when run with ‘-h’ or ‘--help’. Also included in the framework is a gzipped build script folder for building a Honeyd package using the latest sources. ‘honeyd_Ubuntu_buildscript.tar.gz’ is the name of the gzipped tarball containing the build script. In addition, several custom scripts are developed for running experiments (discussed in Section 4.9.3). Finally, a few thirds party tools are also used and included in the framework:

PI.py by Marshall Beddoe

Modified version of PI that provides same enhanced output as ScriptGenE

hexdump.py by Anatoly Tecthonik

Returns or prints a hexdump of a given sequence

redemo.py by Donald Stufft

Regular expression test program with Tkinter GUI

The next two sections provide a detailed discussion of the protocol tree building and replay scripts (ScriptGenE.py and ScriptGenEreplay.py, respectively).

3.6.2 ScriptGenE.py.

3.6.2.1 Overview.

ScriptGenE.py takes a set of input arguments including a list of Pcaps. These Pcaps are all parsed to build a protocol tree for every connection. Environmental links are found while building the connection trees. Next, all connection trees that share the same port are combined into a port tree. These initial trees are exported as Python pickle files in the chosen export directory. Optionally, a graphic of the tree can be saved as well.

Next, `GeneralizeTree.py` is called by `ScriptGenE.py` to generalize the trees. This is done by first consolidating all edges that share the same data and parent node. After the initial consolidation, the main consolidation phase clusters/combines all edges that share a parent and are *semantically* equivalent. The combined edge data for the consolidated edges is represented by a regular expression. Intra-protocol dependencies and a default error message are found. Once this is done, the final trees are exported as Python pickle files. These pickle files can then be imported by `ScriptGenEreplay.py` or `clientReplay.py` for emulation.

3.6.2.2 Requirements.

First, `ScriptGenE` is developed in Linux and is intended to be used in Linux. Additionally, several Python libraries are needed:

- `python-numpy`
- `python-dpkt`
- `python-pcap`
- `python-pydot`

`Networkx` version 1.9.1 or later is also required. `Networkx` creates and manipulates graphs in Python [HSS08]. It has been adapted in this research to create protocol trees. At the time of this writing the Ubuntu package is out of date. Version 1.9.1 has some additional methods for trees that are used in `ScriptGenE`. A simple `'python setup.py install'` as root will install the library under `/usr/local`.

Finally, if `PI's align.so` needs to be rebuilt `'python-pyrex'` needs to be installed. Then run:

```
rm PI/align.so
make all && make clean
```

3.6.2.3 Usage.

Figure 21 shows the usage of ScriptGenE.py. As can be seen in the figure there are many options and two required arguments. A default target IP is required, since it will be used to filter the Pcap file. If the Pcap file name ends with an IPv4 address, however, then that IP will be used to filter the corresponding Pcap file instead. At least one Pcap file must be specified, but any number of files can be used. Note, that instead of specifying the options at every run, a configuration file can be used. For instance, Figure 22 shows an example configuration file called ‘argtest.conf’. Each option takes up one line. Using the example configuration file is then as simple as calling ‘./ScriptGenE.py @argtest.conf’.

3.6.2.4 Processing a Pcap file.

Each Pcap is processed packet by packet. An initial filter on target IP and port (if specified) is done. Additionally, any packets that are not IPv4/TCP are filtered out. Messages are represented by packet number, sequence number (SN), acknowledgment (ACK) number, data, and data length. These messages, or pmsg objects, are used in a TCP connection. Each tcp_connection object consists of a 4-tuple, state, client messages, and server messages. Connection objects that have matching 4-tuples are considered equal. These object are used and maintained in a TCP connections table.

While processing the Pcap file, *valid* TCP connections are added to the table as they are found. A TCP connection is considered valid if a 3-way handshake is observed or if the server is observed sending data to a client. The latter is called a ‘missed handshake session’. Retransmitted packets or those with no payload are dropped. Figure 23 shows the state machine that is used to determine a connection’s next state based on the TCP flags found in the packet and who the packet came from (i.e., client or server). As can be seen in the figure, there are different states than what was previously discussed during the networking review in Section 2.2.5.1.

```
usage: ScriptGenE.py [-h] [-p PORT] [--host HOST] [-e EXPORTDIR] [-n MAXNODES]
                  [-d MAXDEPTH] [-s] [-i] [-x] [-a] [-M MACROTHRESH]
                  [-m MICROTHRESH] [--debug | -v]
                  target pcaps [pcaps ...]
```

ScriptGenE version 0.1 by Phillip Warner.

ScriptGenE (ScriptGen Enhanced) generates protocol trees from pcaps. The protocol trees can be used in a honeypot server or client that replays them.

... Snip ...

positional arguments:

target	default Target IPv4 address (dotted quad with no leading 0's). If an IP address is found in pcap filename it will be used to filter that file (See below)
pcaps	.pcap format filenames. If it ends with '{IPv4 dotted quad address}' then that IP will be used to filter that pcap file instead of the default target.

optional arguments:

-h, --help	show this help message and exit
-p PORT, --port PORT	Port number [1-65535]
--host HOST	Host name of target (e.g. www.example.com). Used to search packets for this environmental information
-e EXPORTDIR, --exportdir EXPORTDIR	tree Export directory. (Default: current directory)
-n MAXNODES, --maxnodes MAXNODES	max tree Nodes (Default: 65535)
-d MAXDEPTH, --maxdepth MAXDEPTH	max tree Depth - limits the size of emulated conversation (Default: 65535). Warning: large transfers will require multiple TCP messages
-s, --stateless	generate Stateless tree. All request response pairs will attach to ROOT node. Note, this causes maxdepth option to be ignored.
-i, --initialonly	Generate only the initial protocol tree. Do not generalize it (Default: off)
-x, --exportonly	tree eXport only - Do no generate dot file or tree drawing (Default: Off. Generate drawing 'tree_{IP}_port{PORT}.png')
-a, --displayascii	Display bytes with 7-bit ASCII values in alignment (PI) output
-M MACROTHRESH, --macrothresh MACROTHRESH	Macro-clustering threshold [0-1.0] (Default: 0.5)
-m MICROTHRESH, --microthresh MICROTHRESH	Micro-clustering threshold [0-1.0] (Default: 0.4)
--debug	Print DEBUG messages (Default: off)
-v, --verbose	Print INFO messages (Default: off)

Figure 21. ScriptGenE.py Usage


```
$> cat argtest.conf
192.168.1.206
pcaps/plc/L61_19_015/WhoActiveGoOnline_0.pcap
pcaps/plc/L61_19_015/WhoActiveGoOnline_1.pcap
-e tree-exports/plc/L61_19_015/WhoActiveGoOnline-all
-M 0.5
```

Figure 22. Example ScriptGenE.py configuration file

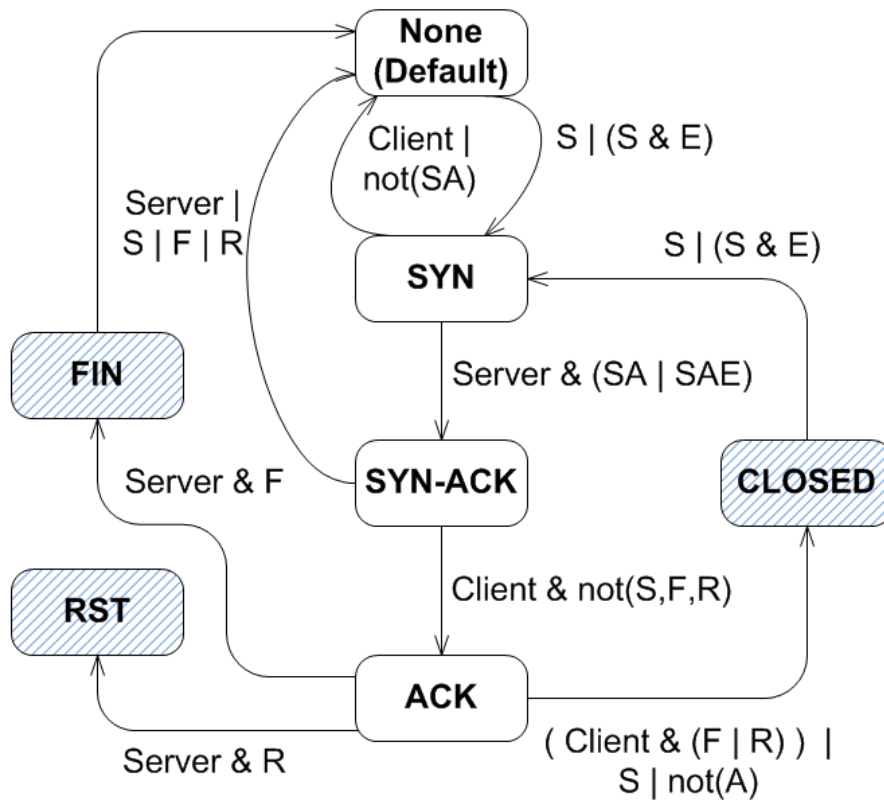


Figure 23. tcp_connection object life cycle

States are described as follows:

None - (Default) connection that never was or was recently FIN'd

CLOSED - closed connection that was previously established

SYN - start of 3-way handshake (SYN correctly received)

SYN-ACK - second part of 3-way handshake (SYN-ACK correctly received)

ACK - 3rd part of 3-way handshake received (ESTABLISHED connection)

The reason the state machine looks different than what is typically found in literature is because this state machine does not represent a client or server state. It represents the state of the session from a third party's point of view. Additionally, the FIN and RST states are special because they affect how the server will respond. If a client sends a FIN or RST, the corresponding protocol tree will simply end with the previous node (server message). This is considered a normal closed session. However, if the client sends something that makes the server respond with a RST or FIN, then the emulator should follow suit. The replay options, which will be discussed in Section 3.6.3 allow different repeat behavior depending on the type of final node (CLOSED, FIN, or RST).

Also, note the transition from 'FIN' to 'None'. This is there because at least one more ACK is expected to properly close the connection. If it were closed immediately then the next ACK (from the server) might be confused as belonging to a 'missed handshake session'. There is no need to wait around for more data after the first FIN-ACK, so the connection is dropped and any other packets for it are ignored. Once the TCP connection is concluded (highlighted states in Figure 23), a connection tree is built.

Building initial trees. The initial connection tree is built by first sorting the client and server message lists by sequence numbers. This accounts for out of order packets. Figures 24 and 25 show the initial output while parsing a Pcap file. Each client and server message object are printed (minus data). Note, the message number is equivalent to the unique frame identifier found in Wireshark.

```
#####
Attempting to parse PCAP file pcaps/plc/L61_Traces/RWho_refresh1.pcap
#####
Building tree ending with 'CLOSED' node for connection Client: 192.168.1.201:3
240; Server: 192.168.1.206:44818; State: ACK

33 client msgs:
Msg 10: sn: 0; ack: 0; data length: 0
Msg 13: sn: 1; ack: 1; data length: 24
Msg 14: sn: 25; ack: 1; data length: 24
```

Figure 24. Initial output while parsing Pcap file

```
33 server msgs:
Msg 11: sn: 0; ack: 1; data length: 0
Msg 17: sn: 1; ack: 73; data length: 66
Msg 18: sn: 67; ack: 73; data length: 50
```

Figure 25. Server messages from Pcap file

Next, the ACKs are used to correlate which messages are in response to the other. From this the connection tree is iteratively built, resulting in one branch. If the server sends a response upon connection, then this data is included in the *ROOT* node. Also, the ‘missed handshake’ sessions are specially handled by creating a dummy client message with data set to ‘MISSING’. Later these sessions can be merged into the P-tree where their respective message sequences align with other message sequences in the P-tree.

It should also be noted that since this framework is designed to work at the application layer, consecutive messages from one host are combined into one message. This is a simpler approach than adding in dummy nodes and edges to represent no

responses between consecutive messages. Figure 26 shows an example conversation summary output that will make message consolidation more clear. In the figure it can be seen that 30 RRP's have been found in the connection, and three of them are shown (Note, the term RRP was originally used in [RCHJ14] as described in Section 2.3.3). The first RRP consolidated three messages identified by their respective packet IDs. The client actually sent three requests before the server began responding. Another situation where messages are merged is if they are split due to their size (e.g., large HTML pages). In this case, the algorithm merges them if the data length is greater than the maximum segment size (MSS), which is assumed to be 1460 bytes for now. This MSS value is derived by subtracting 40 bytes for the TCP/IP headers from 1500, which corresponds to the common maximum transmission unit size for Ethernet.

```
Conversation Summary:
-- 30 RRP's --
c1: (72 bytes) [13, 14, 15]
s1: (142 bytes) [17, 18, 20]
c2: (28 bytes) [21]
s2: (28 bytes) [22]
c3: (59 bytes) [23]
s3: (24 bytes) [24]
```

Figure 26. Conversation summary

Environmental links. While building each connection tree, each consolidated message is checked for environmental links. This is similar to what Roleplayer does, but the implementation is currently not as thorough. IP addresses, ports, and the server hostname are inferred by simply finding matching strings in the consolidated messages. The IP addresses and ports are inferred from the trace itself, but currently the hostname must be specified via a command line option. Future work can pull hostnames from Domain Name System (DNS) messages within the trace. It would also be worthwhile to search for other representations (hexadecimal, little

endian, unicode, etc) and more types of environmental links (e.g., length fields). The links themselves are stored in a dictionary called ‘`env_links`’ that is embedded into the node or edge. The type of link (e.g., ‘`server_ip`’) is the key, and it points to a tuple consisting of a list of offsets where matches were found and also the original value itself.

3.6.2.5 Generalizing Trees.

Generalizing the P-trees is done by going through two rounds of consolidations. First, however, a client lookup table aptly called ‘`clientLUT`’ is generated that contains a unique index for every client *Edge* object. Each Edge object contains only selected attributes from a P-tree client edge. Specifically, a parent node ID, child node ID, and data are included. Every edge in the P-tree will potentially have its data modified and merged, so the clientLUT provides a reference for the original client message data. This is crucial for deriving and maintaining intra-protocol links.

Initial consolidation. After the clientLUT is created, the initial consolidation begins. Its goal is to simply find and merge trivially matched edges before starting the main consolidation. In a recursive depth first search (DFS) of the tree, any edges that have the same parent and have equal data (other than environmental links) are consolidated. To take differences due to environmental links into account, markers are used to replace the linked data. Before comparing sequences, the linked data is replaced with a marker equal to ‘`#E-{type}#`’, where ‘`type`’ is the link type (e.g., ‘`#E-client_ip#`’). While merging, the *weight* of the edge is increased by the number of merges. Additionally, if the corresponding server responses do not match, then the merge simply appends another response choice to the server node’s data list. This helps account for variability in the server messages that is not due

to the client messages (e.g., time-driven states). Finally, note that after this initial consolidation all of the ‘missed handshake’ sessions will share the same consolidated edge with ‘MISSING’ as its data. This makes it easier to use or remove this branch later. Figure 27 shows an example of the output seen after initial consolidation.

```
Done building all initial trees
1 trees created for ports: [44818] in 0.10 seconds
Exporting initial trees...
Generalizing trees...

Initial tree has 90 nodes... reduced to 86 nodes after 4 initial consolidation
s
```

Figure 27. Output from initial consolidation

Main consolidation. After the initial consolidation, the main consolidation begins. Again, in a recursive DFS of the tree, any edges that have the same parent are considered for consolidation. First, however, since handling ‘missed handshake’ sessions is not fully implemented, that branch is simply deleted for now. Remaining neighboring edges are then *Macroclustered* using the Smith-Waterman local alignment and UPGMA clustering from PI. However, instead of performing *microclustering* using region-wide mutation rates like ScriptGen, ScriptGenE checks if the corresponding server messages cluster in the same groups. The intuition here is that even a one bit change is significant if it causes the server’s response to be significantly different. Thus, server message clustering is used to break up the client clusters as needed. As a result, ScriptGenE is expected to require less samples than ScriptGen to correctly microcluster. The downside to this approach, however, is that it can be computationally expensive for large clusters of large server messages (e.g., HTML responses). ICS traffic, on the other hand, is typically expected to consist of many short messages that will not require as much computations to process. It is recommended that future work set a cluster and message size limit in order to always efficiently build P-trees re-

ardless of the input sequences. Also, microclustering by region-wide mutation rates would still be useful, because small changes that are not detectable by Macroclustering (client or server messages) can have significant meaning. With microclustering, significant changes in even one bit can be detected given enough samples that show that bit as significant.

Region Analysis. Once a Macrocluster has been created and aligned by Needleman-Wunsch, Region Analysis is performed on the *gapped* consensus sequence. This algorithm analyzes the aligned byte sequences and defines consecutive sequences of bytes as a region based on:

1. Same DT (Zeros, Binary, Space, ASCII, or GAP)
2. MR within 0.1 (10%) and all bytes are either zero or non-zero
3. Gap presence the same (yes/no)
4. Same ‘kind’ of data for MCB (If ASCII: alpha, digit, or punctuation)

Notice ScriptGenE defines ‘kind’ as various forms of ASCII. It is unknown what the ScriptGen authors actually intended to be a ‘kind’ of data.

Figure 28 shows an example of Region Analysis (RA) output for part of a HTML GET request. This example uses ScriptGenE.py’s ‘--displayascii’ option to display printable text for each MCB on the ‘TXT’ rows. The regions are identified by the ‘RA’ row at the bottom of each block of aligned bytes. The ‘RA’ labels are boxed in blue for easier identification. Fixed regions are identified with a ‘F’ and mutating regions are identified with a ‘M’. Mutating bytes are identified by the entries in the ‘MR’ rows that are highlighted in red.

Notice the example in Figure 28 has gaps in the alignments, and some regions do not have a MCB. In the case where the MCB *is* a gap, the *GAP region* is labeled with

a ‘G’ instead of a ‘M’. Note, GAP regions are *always* mutating regions because the alignment algorithms only produce gaps when aligned bytes differ. Therefore, while gaps are artificially inserted into the sequences during alignment, they represent important information. This is why RA is performed on the *gapped* consensus sequence instead of the *ungapped* consensus sequence. If the ungapped consensus sequence was used to define regions for the sequences in Figure 28, then one byte would be missing from the regions. Sequences 15 and 12 both have one gap in different places (circled in blue over regions G00 and G03, respectively), but two gaps would be removed from the ungapped consensus sequence. Thus, it is important to retain the gaps, but it still must be recognized that they are not always to be used.

Output of cluster 3 (3 sequences)																
0001	x47	x45	x54		x2f											
0015	x47	x45	x54		x2f		x69	x6e	x64	x65	x78	x2e	x68	x74	x6d	x6c
0012	x47	x45	x54		x2f	x72	x6f	x62	x6f	x74	x73	x2e		x74	x78	x74
DT	AAA	AAA	AAA	SSS	AAA	GGG	AAA	AAA	AAA	AAA	AAA	AAA	GGG	AAA	AAA	AAA
MR	000	000	000	000	000	066	100	100	100	100	100	066	066	066	100	100
MCB	x47	x45	x54		x2f		???	???	???	???	???	x2e		x74	???	???
TXT	G	E	T		/									t		
RA	F00	.	.	F01	F02	G00	M01	M02	G03	M04	M05	.
0001		x48	x54	x54	x50	x2f	x31	x2e	x31			x55	x73	x65	x72	x2d
0015		x48	x54	x54	x50	x2f	x31	x2e	x31			x55	x73	x65	x72	x2d
0012		x48	x54	x54	x50	x2f	x31	x2e	x31			x55	x73	x65	x72	x2d
DT	SSS	AAA	AAA	AAA	AAA	AAA	AAA	AAA	AAA	SSS	SSS	AAA	AAA	AAA	AAA	AAA
MR	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000	000
MCB		x48	x54	x54	x50	x2f	x31	x2e	x31			x55	x73	x65	x72	x2d
TXT		H	T	T	P	/	1	.	1			U	s	e	r	-
RA	F03	F04	.	.	.	F05	F06	F07	F08	F09	.	F10	.	.	.	F11

Figure 28. Example RA output - HTML

Next, Figure 29 shows an example of RA output for part of an EtherNet/IP (ENIP) message. The naming convention is ‘{packetNumber}_{filename}’ for parent and child IDs shown at the top. These client IDs correspond to the numbers identifying each sequence row. The mutating regions highlighted in red are parts

of the protocol’s ‘session handle’ and ‘sender context fields’. Note, region *F07* is a coincidental fixed region since it is actually a part of the variable sender context field (discussed later in Section 5.3.2).

Finally, by comparing Figures 28 and 29 to PI’s original output in Figure 17, several enhancements can be seen. RA, MCB, and TXT rows are added. The mutation rate (MR) line is relabeled from ‘MT’ to ‘MR’ to better reflect its meaning. MRs above a threshold (set at 0 currently) are highlighted, and the number of sequences is provided. Additionally, the code providing output is rewritten with added efficiencies.

Create regular expression. After defining regions, ScriptGenE creates a regular expression to represent the cluster based on the various fixed and mutating regions. Fixed region bytes are represented *as is* in the regex, while MRs are represented by *any* byte over the size of the region. The regular expression could represent MRs using only the byte types that are represented in the samples, but it is assumed that MRs do not have a clearly defined DT with a limited number of samples. An example regular expression is shown in Figure 30 that corresponds to the cluster in Figure 29. Note, in Python regular expressions ‘\’ characters must be escaped with a ‘\’, and ‘\x’ denotes a hexadecimal byte. The regular expression, or regex, is compiled with the ‘re.DOTALL’ option so that ‘.’ corresponds to *any* character. Therefore, parts of the regex such as ‘.{2,2}?’ (highlighted by red boxes in Figure 30) mean that any character is allowed over the *m* to *n* range (where *m* and *n* are both ‘2’ in this case). *m* is set at the number of aligned bytes in consecutive mutating regions that never had a gap, and *n* is set at the number of aligned bytes seen across the consecutive mutating regions plus an optional *flex factor*. The flex factor takes effect when gaps are present and by default adds 10% to *n* rounded up to the nearest byte. Thus, by default the flex factor will cause the regex to accept at least one additional variable byte for regions with gaps. This additional flexibility is

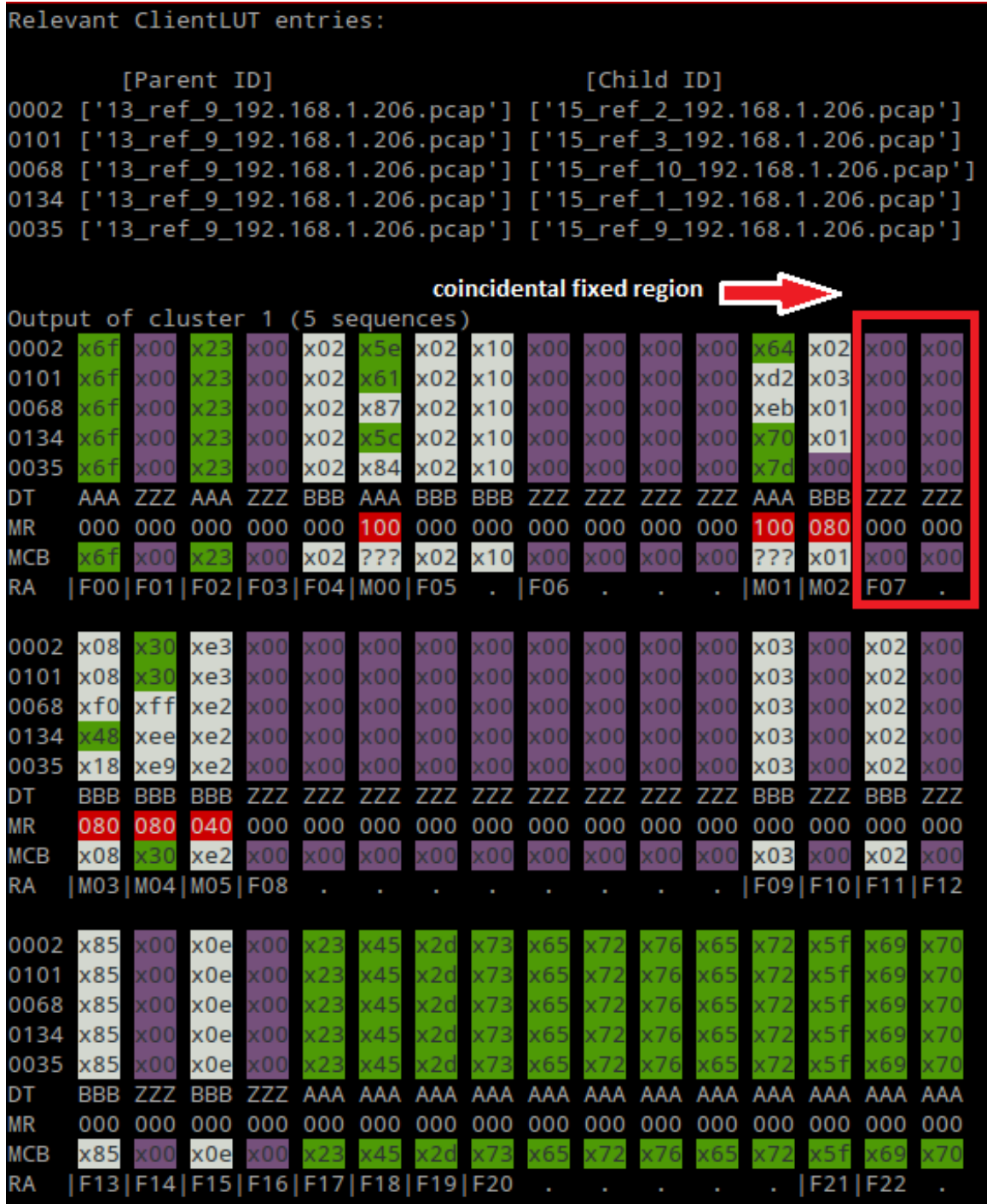


Figure 29. Example RA output - EtherNet/IP

desired to account for cases where the client request may actually be slightly longer than previously seen in the traces. This follows the intuition that regions with gaps do not have a well defined length with a minimal set of traces.

```
Regex str for this cluster is:
'\x6f\x00\x23\x00\x02.{1,1}?\x02\x10\x00\x00\x00\x00.{2,2}
?\x00\x00.{3,3}?\x00\x00\x00\x00\x00\x00\x00\x00\x00\x03\
\x00\x02\x00\x85\x00\x0e\x00\x23\x45\x2d\x73\x65\x72\x76
\x65\x72\x5f\x69\x70\x23\x00\x91\x00\x05\x00\x06\x00\x0
1\x04\x03'
```

Figure 30. Regular expression for example EtherNet/IP cluster

To better illustrate how gaps affect the regular expression, Figure 31 shows the regular expression corresponding to the HTTP request in Figure 28 (Note, the regex is for the entire message while only part of the message is shown in Figure 28). As can be seen in Figure 31, the match interval representing the mutating regions has values of 0 for m and 13 for n . m is 0 because sequence 1 has only gaps in the mutating regions. n is 13 because there are 11 bytes in the mutating regions plus two bytes ($\lceil(11 * 0.1)\rceil$) added from the flex factor.

```
Regex str for this cluster is:
'\x47\x45\x54\x20\x2f.{0,13}?\x20\x48\x54\x54\x50\x2f\x31
\x2e\x31\x0d\x0a\x55\x73\x65\x72\x2d\x41\x67\x65\x6e\x7
4\x3a\x20\x57\x67\x65\x74\x2f\x31\x2e\x31\x33\x2e\x34\x
20\x28\x6c\x69\x6e\x75\x78\x2d\x67\x6e\x75\x29\x0d\x0a\
\x41\x63\x63\x65\x70\x74\x3a\x20\x2a\x2f\x2a\x0d\x0a\x48\
\x6f\x73\x74\x3a\x20\x23\x45\x2d\x73\x65\x72\x76\x65\x72
\x5f\x69\x70\x23\x0d\x0a\x43\x6f\x6e\x6e\x65\x63\x74\x6
9\x6f\x6e\x3a\x20\x4b\x65\x65\x70\x2d\x41\x6c\x69\x76\x
65\x0d\x0a\x0d\x0a'
```

Figure 31. Regular expression for example HTTP cluster

Finally, the '?' means to use characters non-greedily (i.e., do not use them unless the match will fail otherwise). This is crucial for protocol fields that have variable lengths, because the corresponding mutating region bytes may not need to be used (particularly GAP regions as described previously). Overall, the described method

for creating regular expressions provides flexible, yet bounded matches for mutating regions, while the expression parts describing fixed regions provide anchors to ensure accuracy of the overall match.

Final cluster processing. After RA is complete on a cluster, it needs to be further processed by finding intra-protocol dependencies (discussed in Section 3.6.2.6), merging RRP, and setting an attribute called ‘max_client_data_len’. The attribute tells the replay engine the longest it should expect a client message to be. This prevents unnecessary waiting.

Find default error message. After all of the tree clusters are processed, a default error message is automatically selected. The default error message is used during replay when the emulator does not have a matched response for an incoming client request. Its purpose is to try to send something that looks believable enough that the client will continue sending more requests. The algorithm searches for messages *without* links to select within the pool of server messages by the following precedence:

1. Shortest message
2. Least frequency
3. First message of what is left

The intuition behind the selection criteria above is that common error messages are probably shorter and less frequent than regular messages. For instance, common HTTP error status codes such as ‘404 Not Found’ are communicated rather plain and tersely, while normal HTTP responses typically include HTML pages that are full of content. The reason nodes with links are not used is because protocol error messages are often sent due to invalid requests and links are based off of previous requests. Lastly, if no suitable messages are found without links, then ‘ ’ (space) is returned

as the default error message. It is important to note that ScriptGenEreplay.py has multiple sources it can use for error messages (discussed in Section 3.6.3.9).

Output build summary. The final act of ScriptGenE.py is to output information summarizing the qualities of the built P-trees. Figure 32 shows an example. In this case, the example shows a perfect merge between five different sessions since the final number of RRP is $\frac{1}{5}$, or 20%, the initial number (and there are no unclustered edges or gaps). If there is an unclustered edge or gap, the previous edge to that is listed in this section along with the total number of RRP prior to the event. This information is very useful since an unclustered edge indicates that the tree diverged and split off a singular branch. Since that branch has no clusters it has no general regex to represent it, and it does not have intra-protocol links identified! Identifying links that are globally present (e.g., session IDs) would partially alleviate this issue. For now, if the replayer goes down an unclustered path there is a good chance the accuracy of replay will drop if the responses require linked information. Similarly, if gaps are found in alignments, this indicates that there are errors in the alignment. This can also lead to accuracy errors during replay.

Near the bottom of Figure 32 the actual bytes from the default error message are displayed. Finally, statistics are given on how long building the tree took. Most of the time is typically spent on clustering and aligning while generalizing the trees.

3.6.2.6 Intra-protocol dependencies.

Overview. Intra-protocol dependencies are linked bytes between a server response and the preceding client message (e.g., a session ID). This means that linked bytes from a client request will be duplicated in the server response at offsets defined in the link. Server bytes can be linked to any matching client message substring of at least two bytes. The process of finding intra-protocol dependencies first starts

with finding a proposal for every RRP from an aligned cluster. Then, any duplicate proposals are merged, and the weight of the proposal is increased by the number of merges. Third, the list of unique proposals is printed. Fourth, proposals are consolidated by picking the one with the highest per byte consensus or least number of links. Finally, the cluster link summary is printed. The next sections cover the above steps in detail.

Finding proposals. The first step for finding a proposal for a given RRP is to find the *significant fixed regions*. Consecutive fixed regions are considered one large fixed region for this purpose. Every region in the list of fixed regions is checked to see if it is a unique sequence of bytes in the client message. If it is unique, then it is added to the list of significant fixed regions.

Once the list of significant fixed regions is determined, they are considered markers for potential links. Any “chunk” of bytes between two significant fixed regions is checked for links. The marker preceding the chunk is the *starting marker*, and the marker succeeding the chunk is the *ending marker*. The minimum size for a match is two bytes. Therefore, if the chunk size is only one byte, the first byte of the ending marker is also considered.

For every chunk of client data, a window of two bytes is defined at the start of the client message and the corresponding server message. The window slides to the right on the server message until a match is found between the windows. Both windows are then expanded to the right while there is still a match and bytes are left to check on each respective message. If the client chunk runs out of bytes first, an extra byte from the ending marker can be used to check for a match. Once the window is at its maximum width, a link is created. The link does not include the extra byte, so a link of one byte is possible. After creating the link, the windows are reset to two bytes. Then the server message window continues sliding to the right from where it last left

off in order to check for more matches. Once the server window can no longer slide to the right, it is moved back to the beginning of the server message and reset to two bytes. The client window then slides one byte to the right and is reset at two bytes. This process continues until the client window can no longer slide to the right inside the chunk.

Processing a chunk as described previously results in a list of ‘LinkAttrib’ (link attribute) objects, where each object consists of a tuple of a link, server offset, and the original length of the linked data. This raw list must be *deconflicted* because links cannot overlap on the server message. This is done by using a weighted interval scheduling algorithm (previously described in Section 2.2.7.4). The weight, however, is the original length of the link squared in order to give preference to longer substrings versus the sum of many smaller substrings. The intuition here is that longer links are less likely to be trivial.

It is important to note that a single sequence of bytes on the client message can be linked to the server message at multiple offsets. This fact is used to consolidate the links. Once a list of non-overlapping links is obtained, all of the matching links have their server offsets combined into one list. Using the list of offsets, only one link copy is needed.

The list of links is then checked for trivial links. Any link that has more occurrences (i.e., server offsets) than the ‘MAX_OFFSETS’ limit (set at two) is filtered out of the list. A good example of why this is necessary is shown in Figures 33-35. Figure 33 shows an example of a trivial link that has eight server offsets. In this example, both the client and server messages have numerous groups of zeros. Figure 34 shows the resultant proposals where a total of 10 links (denoted by ‘# L #’) can be found. Finally, Figure 35 shows the resultant proposals after filtering out eight trivial links.

After filtering the list of links, a proposal is created by replacing linked bytes with ‘#L#’ markers. This is similar to what was described in Section 3.6.2.5 for environmental links. The markers are used to make comparing different proposals easier. The linked data may be different between proposals, but the dependencies represented by the links could be the same.

After a proposal is generated for every RRP in the cluster, duplicates are merged and the weight of the respective proposal is increased by the number of merges. Then, the list of unique proposals is printed as was shown in Figure 34 and Figure 35.

Consolidating proposals. After the list of unique proposals is generated, one proposal must be selected to represent the group. This filters out coincidental links. Picking this consolidated answer is done using an algorithm similar to what was described for ScriptGen in Section 3.5. However, the general algorithm for ScriptGen’s link consolidation described in [LDM06] leaves a lot of ambiguities and questions. ScriptGen’s algorithm immediately discards a proposal once it does not possess the MCB for an aligned byte. However, what happens if there are an even number of proposals, half of them have a certain value in the first byte, and the other half share a different value? How are ties broken, and how are the correct half of proposals chosen? Secondly, referring to Figure 19, the longest proposal (number 5) seems to be arbitrarily chosen. Proposal number 6 could just have as easily been picked on the seventh byte. What is truly the best criteria for breaking ties, and what is the specific algorithm for guaranteeing that the consolidated answer matches at least one proposal?

Thus, this research presents a novel algorithm, as shown in Algorithm 4, that answers all of these questions. One of the main ideas behind it is that proposals with less links are less likely to have coincidental links. This means that longer proposals are not always better. Additionally, Algorithm 4 does not dismiss a proposal early on

simply because it does not have the MCBs for *every* aligned byte up to the last byte position checked. Rather, it calculates how many MCBs each proposal has over the *total* number of aligned bytes checked. Each aligned byte may have *multiple* MCBs (i.e., ties), so each one could potentially add to a proposal’s score. Also, calculating the MCBs for each aligned byte is only done while most of the proposals have a byte (i.e., not a GAP) at that position. Once the list of MCBs for an aligned byte contains a GAP, that list is discarded and byte frequencies are not counted further for that round. The intuition here is that if most of the proposals are only n bytes long, then checking further will only skew the results (and inherently give preference to longer proposals).

Next, all of the proposals that have the highest score (i.e., most MCBs) are put in the ‘bestProposals’ list. The algorithm then repeats recursively on smaller, more focused proposal lists until the list size no longer shrinks, or it gets down to two proposals or less. The smaller lists are ‘more focused’ because previous rounds filtered out less important proposals. The MCBs at each aligned byte may change due to the filtering. Thus, each round converges further to the overall solution.

Once the algorithm has converged as much as possible using MCB scores, other measures are used to break ties. If more than one proposal is left then the list is paired down further by selecting those with the least number of links. If there are still multiple proposals left, then the first one is arbitrarily selected.

Additionally, as can be seen in Algorithm 4, the MCBs for each aligned byte are only calculated if more than two *unique* proposals are given as input. As discussed in the last section, all of the proposals are unique, and they have a weight assigned to them describing how many copies of the proposal there were prior to consolidation. Thus, given a good cluster with highly similar proposals, this algorithm can very efficiently select a proposal.

Algorithm 4 Consolidate Link Proposals

Input: initial proposals list where each proposal is a tuple of (ID, message, weight)

and all messages are unique

function CONSOLIDATELINKPROPOSALS(proposals, done = FALSE)

if done or ≤ 2 proposals left **then**

if > 1 proposals left **then**

if 2 proposals left AND weights are not equal **then**

return the proposal with the highest weight

end if ▷ Otherwise, pick proposal by least links

 selectedProposals = proposals with least number of links

if only 1 selectedProposal **then**

return selectedProposal

end if

end if

return first proposal of what is left in list ▷ default return

else ▷ not done finding best proposals

 taking weight into account, find the MCBs for each aligned byte in proposals
 up to the most frequent message length

 ▷ (i.e., stop when *one* of the most frequent bytes is a ‘GAP’)

delete last MCB list entry that contains ‘GAP’

 bestProposals = list of proposals with the highest number of MCBs

if no change in proposals list length **then**

 done = TRUE

end if

return CONSOLIDATELINKPROPOSALS(bestProposals, done)

end if

end function

Finally, to solidify understanding Algorithm 4, it will be applied to the example proposal list from Figure 19. All of the seven proposals are assumed to have a weight of one. During the first round the MCBs are B, L, B, and L, respectively, for the first four aligned bytes. For the fifth aligned byte, both B and GAP have the highest frequency. That MCB list is discarded, and the round ends with proposals 1 and 2 being discarded. During round two the MCBs are the same until the fifth aligned byte. B is the MCB for the fifth byte. B and GAP are the MCBs for byte six. Proposals 3 and 7 are discarded. During round three proposal 4 is discarded. This leaves only two proposals. Proposal 5 is then selected because it has only two links, while proposal 6 has three links.

Cluster Link Summary. After proposals for every RRP have been found and consolidated down to one, the final list of links for the cluster is printed in the cluster link summary. As can be seen in the example in Figure 36, the cluster link summary shows the actual links from the chosen proposal along with information about the proposal list. The number of *valid* link proposals is the number of proposals that matched the consolidated proposal. Thus, it gives a measure of confidence in the answer. In the example from Figure 36, all of the proposals were the same, so it is very likely the chosen proposal is correct. Finally, the ‘Representative Child ID’ is the corresponding server node ID for the chosen proposal. This node ID is the container for merging the cluster RRP.

Link limitations. There are a few limitations regarding finding intra-protocol dependencies:

1. Links will only be found for clustered messages
2. Links are only found for the previous client message, but the link may actually be to a client message much earlier in the session

```

-----
Cluster Link Summary
-----
Total proposals found: 3
Unique link proposals found: 1
Valid link proposals: 3
Representative Child ID: 175_WhoActiveGoOnline_0b.pcap

2 Links:

Link(clientID=955, s_marker=bytearray(b'\x02\x10\x00\x00\x00\x00'),
      r_marker=bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00 \x00\x02
\x00\x00\x00\x00\x00\xb2\x00\x1a\x00R\x02 \x06$\x01\x07\xf9\x0c\x00
\x03\x03!\x00\x1a\x03$\x00\x01\x00\x01\x00\x01\x00\x01\x01'), offse
t_s=0, offset_r=0)
Server offset(s): [12]
Original length: 7
Original data: '\xcf\x00\x00\x00\xa8\x8e\xe2'

Link(clientID=955, s_marker=bytearray(b'o\x00*\x00\x00'), r_marker=
bytearray(b'\x02\x10\x00\x00\x00\x00'), offset_s=0, offset_r=0)
Server offset(s): [5]
Original length: 1
Original data: '6'

```

Figure 36. Cluster link summary

3. Only one consolidated server message is generated in output (required to filter out incidental matches)
 - this means that unique server messages due to external factors (e.g., time) will be dropped
4. Arbitrary links can still be found since two or more bytes may coincidentally match from the client message to the server message
 - this is particularly true when considering text protocols such as HTML

3.6.2.7 Protocol Tree Exports.

Three different kinds of files are exported from ScriptGenE. First, there are Python pickle files that are exported for both the initial and final trees. Python pickle files

are binary representations of a Python object. Because of this, the classes that the object requires must be able to be loaded into memory when importing the pickle file. Thus, to import a P-tree, the ScriptGenE module files must be in place.

The second kind of exported files are pictures of the protocol trees. These PNG format pictures are optionally created, and they allow the structure of the P-tree to be quickly assessed. Figure 37 shows a simple example of an initial protocol tree built from two similar traces. Figure 38 shows the corresponding final tree after it has been generalized. It should be noted that there is a limit to how wide of a tree can be represented using this format. It is observed that an excess of 200 branches will cause the picture to not be created. This is a limitation of the pygraphviz library that networkx uses to create the picture. However, long sessions appear to be able to be represented. Viewing individual nodes then requires zooming in on the areas of interest. Improved visual representations of large protocol trees is left as future work.

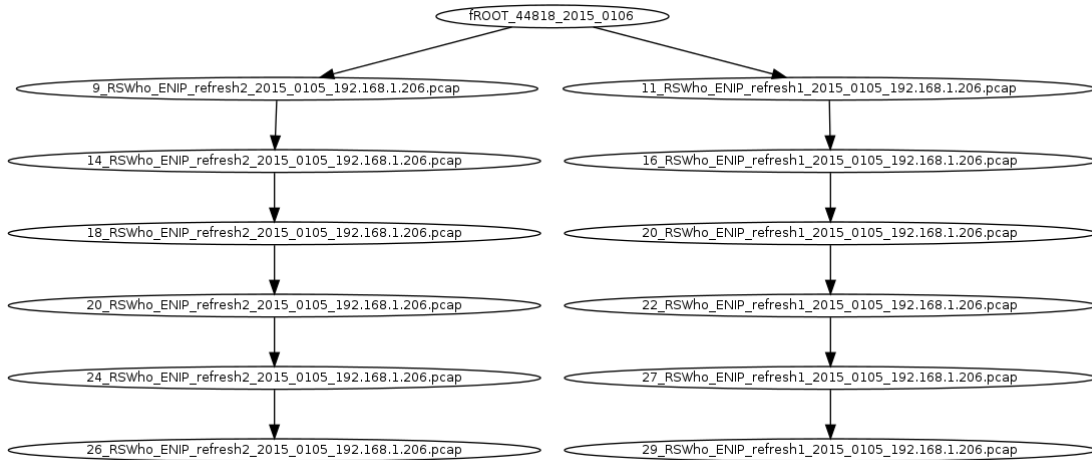


Figure 37. A simple initial protocol tree

The final exported file type is a Graph Exchange XML Format (GEXF) of the initial tree. Figure 39 shows part of the GEXF file opened in Firefox web browser. Different parts of the XML structure can be collapsed and opened by clicking on the ‘-’ or ‘+’ sign to the left of the tags. Unlike the picture format, this representation

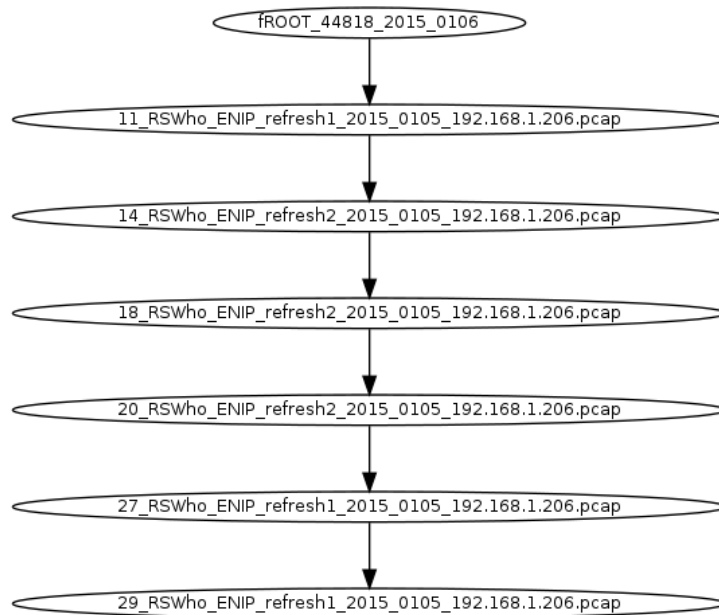


Figure 38. A simple final protocol tree

can include attributes, and the XML format is well suited for importing information into a variety of programs. However, binary data cannot be represented without first encoding it. Also, the networkx export function for the GEXF format only allows certain types of Python structures to be included. As a result, many attributes have to be converted prior to exporting them. Currently, only the node ID, type, and weight are exported. Future work can include other attributes if they are found to be needed.

3.6.3 ScriptGenEreplay.py.

3.6.3.1 Overview.

ScriptGenEreplay.py takes a set of input arguments that define the server to be emulated. The specified P-tree is quickly loaded, and a server socket is opened on the chosen port. The IP address is determined by the chosen interface. Upon a client connection, the server sends a “welcome” message if it has one stored in


```

- <gexf version="1.1" xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-instance">
- <graph defaultedgetype="directed" mode="static">
- <attributes class="edge" mode="static">
  <attribute id="1" title="ntype" type="string"/>
</attributes>
- <attributes class="node" mode="static">
  <attribute id="0" title="ntype" type="string"/>
</attributes>
- <nodes>
- <node id="fROOT_44818_2015_0106" label="fROOT_44818_2015_0106">
  - <attvalues>
    <attvalue for="0" value="ROOT"/>
  </attvalues>
</node>

```

Figure 39. Viewing a GEXF file in Firefox

ROOT's data. The TCP connection provides the data from the client in a stream of bytes, so every chunk of data received is checked for a match. This process continues until either the expected number of bytes have been received (as determined by `max_client_data_len` described in Section 3.6.2.5), a match is found, or an error occurs.

If a match is found, the corresponding server message is retrieved and sent if it has valid data. The matched client data is then removed from further consideration. If no match is found, or if the server response has no valid data (e.g., it is an empty FIN/RST node or has missing linked data), then an unknown transition is handled. Depending on how the emulator options are configured, the server will either close the connection, backtrack to a different state with a matching transition, or remain in the same state while sending a default error message.

During the whole session a counter keeps track of the matched client messages. This is not only used to determine how many messages the server was able to handle, but it also is used to determine if backtracking is allowed. The following sections describe the usage and major parts of `ScriptGenEreplay.py` in detail.

3.6.3.2 Usage.

The usage and options for `ScriptGenEreplay.py` are shown in Figure 40. As can be seen in the figure, the only required arguments are the P-tree file name to import and the port the server should run on. The initial IP address for the server will be automatically detected by the chosen network interface (default interface is `lo`). The server host name should be specified if the P-tree was created with environmental links for the server host. This is particularly important if the host name in the original traces is a different length than the chosen host name for the emulator. The `'--original_data'` option is intended for cases where it is more desirable to use the original environmental data rather than the environmental data of the emulator's session (see Section 3.6.3.5).

The `'--repeat'` and `'--strict'` options define how unknown transitions are handled (see Section 3.6.3.7). Repeating is defined by the `'--repeat'` option and only takes effect when the end of the tree is reached. If set to `'open'`, then backtracking will only occur if the final node is a regular `'MSG'` node. Setting the option to `'always'` ignores the final node type. Finally, setting the option to `'never'` prevents repeating all together.

Whether or not backtracking is allowed, however, is dependent on the number of matched client messages and the `'--strict'` option. The strict option determines how many client messages must be matched before backtracking is allowed. For instance, the strict option can be used to make sure a valid application session is setup prior to allowing the client to diverge from the order of states in the P-tree. If the user knows a valid session starts after five transitions, then the strict option is set to five. By default, there is no restriction set, and setting the option to `-1` disables backtracking.

Next, the ‘`--default_error`’ option allows the default error message that is stored in the tree to be overridden (see Section 3.6.3.9). Finally, the ‘`--debug`’ and ‘`--verbose`’ options change the message level to DEBUG or INFO, respectively. Setting either of these will allow the hexdump of the matched client messages to be viewed in INFO messages.

3.6.3.3 Initialization.

After checking that the program input arguments are valid, the P-tree is imported and a server socket is opened. This socket uses two modified TCP options. First, *nagling* is turned off so that small TCP packets are not combined and buffered before being sent. While the emulator is designed at the application layer, it is desirable for messages to be sent as represented in the protocol tree. Secondly, keep alive packets are also sent every eight seconds of idle time. This is to help keep the connection open with the client and encourage an adversary to continue using the honeypot.

As mentioned previously, the server IP address is automatically detected by the chosen network interface. However, after the socket is created using that IP address, `getsockname()` is called to obtain the IP address and port of the socket. This is done so that ScriptGenEreplay can also be used as a Honeyd *subsystem* (i.e., stand alone service script). As discussed in [PH07], Honeyd hooks into the `getsockname()` function call in order to allow programs to determine the virtual IP and port that Honeyd assigned to them. If Honeyd is not being used, then the same IP address and port will be returned.

Next, the obtained server IP and port, along with the host name (if provided), is added to a dictionary storing the session environmental information. Once a client connects, their IP address and port are also added to the dictionary. Additionally, if the P-tree ROOT node has data in it (i.e., the server has a “welcome” message),

```

$> ./ScriptGenEreplay.py -h
usage: ScriptGenEreplay.py [-h] [-i IFNAME] [--host HOST] [-f]
                           [-o {client,server,all}] [-r {never,open,always}]
                           [-s [n]] [-d [DEFAULT_ERROR]] [--debug | -v]
                           tree_file port

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.

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)
  -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)
  -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 not specified (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 correponding
                        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 40. ScriptGenEreplay.py Usage

then the data is sent to the client. Currently, only one client connection is supported at a time.

3.6.3.4 Matching client requests.

As discussed previously, every chunk of client data received is checked for a match. All of the current node's outgoing edges are compared against the received client data. If there are no outgoing edges (i.e., the current node is a tree leaf), then an unknown transition must be handled. Before comparing the client data to tree edge data, bytes in the client data must be replaced by environmental link markers as specified in the edge `env_links` attribute. The edge data itself can either be a byte string (for unclustered edge) or a regular expression (for a clustered edge). If more than one edge matches the marked up client data, then an edge is chosen in order of preference by:

1. largest weight
2. largest number of descendants
3. random selection

Choosing an edge by the largest number of descendants is unique to the ScriptGenE Framework. An edge with the most descendants is desirable because this maximizes the number of future transitions that can be handled before backtracking. Finally, the selected edge ID is printed, and a hexdump of the matched client data is optionally logged in an INFO message.

3.6.3.5 Marking up client data.

Replacing bytes in client data with markers is done in increasing order of offsets. There may be a difference in the length of the marker and the length of the replaced

bytes. Thus, one change can affect offsets of later bytes. While replacing bytes with the markers, these potential length differences are factored into a cumulative offset difference.

In addition to using correct offsets, replacing bytes with markers also requires that the correct number of bytes be replaced. The number of bytes to replace for a given marker is equal to the length of the respective environmental link's value for the current session. However, if the emulator options specify to use original data for client messages, then the length of the original linked data is used.

These length differences are one of the reasons why the emulator allows original data to be used for environmental links. Since the current implementation does not detect length fields, it may be desirable to use original data instead of data detected from the environment. If the contextual data has a different length than the original environmental data, then length fields in the resultant message may not be correct. Since client applications often use length fields to parse messages, an incorrect length field will typically cause the client to either truncate the server message or wait for the rest of it to arrive. The other reason to use the original data is for testing the protocol tree. `clientReplay.py` also has the option to use original data, so `ScriptGenEreplay.py` can be used with it to replay the original session.

3.6.3.6 Checking for FIN or RST nodes.

As discussed in the Overview in Section 3.6.3.1, it is important to know if the next server message is a FIN or RST. If the next node's type is normal (i.e., 'MSG'), then replay continues normally. However, if the next node's type is 'FIN' or 'RST', then it must be specially handled. If backtracking is allowed and the '`--repeat`' option is set to 'always', then the node is treated like a normal leaf node. Thus, if the node has data, then the data will be sent. If, on the other hand, it does not have data,

an equivalent edge must be found in the tree that leads to a node with data (i.e., an unknown transition must be handled).

For the cases where repeating or backtracking is not allowed, the connection will be closed. If the node has data, then it is included while sending a FIN flag to close the connection. If there is no data to send, then the method of closing the connection depends on the node type. In the case of a RST node the connection is abruptly closed; no further data is read or written by the server's network buffers.

3.6.3.7 Handling unknown transitions.

Handling unknown transitions is required any time a client message cannot be matched against any of the outgoing edges of the current state. This can happen when the end of the tree is reached (i.e., current state is a leaf and has no outgoing edges), or when the client message simply does not match the available transitions from the current state. In the case that the current node is a leaf, backtracking will occur if backtracking *and* repeating are allowed. For non-leaf nodes, backtracking will occur as long as it is allowed. If backtracking is unsuccessful, then the current state remains the same and a default error message is sent. If, on the other hand, backtracking is not allowed then the connection is closed after sending the default error message.

3.6.3.8 Backtracking.

Backtracking is one of the novel algorithms presented in this research that make ScriptGenE more flexible in its emulation of ICS devices. It is used whenever an unknown transition must be handled. The goal of the algorithm is to simply find the earliest P-tree edge that matches the current client request and return the corresponding next state (i.e., server response).

The algorithm is as follows. First, the path from ROOT to the current node is determined by iteratively getting each node’s parents. Then, starting from ROOT and moving down the found path, each node’s outgoing edges are checked until a valid match is found. If no match is found along the path, then a BFS of the edges in the tree is performed until a match is found. The BFS is modified to skip any previously checked edges. If no matches are found then ‘None’ is returned as the next node.

The described algorithm makes emulating ICS devices more flexible because these devices often enter long polling sessions where data values are reported. There are typically repeating patterns in the traffic generated by these sessions because control systems attempt to maintain equilibrium in the controlled process. Because of backtracking, the P-tree only needs to represent part of the polling session. Once, the end of the tree is reached, the next response is found by backtracking to the earliest matching request. Thus, backtracking enables looping of session content. Emulating the repeating behavior of polling sessions is why the path to the current node is checked first. However, when the session diverges from previous content, the BFS of the other edges allows the P-tree to handle the client request regardless of the current state. As previously described in Section 3.6.3.2, the ‘--strict’ option controls backtracking by setting the minimum number of matched client requests required for backtracking to be enabled. Overall, backtracking is designed to be a powerful and flexible emulation technique.

3.6.3.9 Sending default error message.

In the event, however, that backtracking cannot find an appropriate server state to use for responding, a default error message is sent. If a default error message was found during the tree building process, then the tree’s ROOT node will have a tuple stored in an attribute called ‘default_error’. The tuple consists of a node ID and

a data string. The data string is the chosen error message, and the node ID identifies which node the message came from. The ID can then be used to lookup any node information (e.g., `links` and `env_links`) that is required to manipulate the error message. The current implementation only inserts environmental data into the error message. Since environmental information remains the same throughout a session, once the information has been inserted into the error message the new value replaces the original string in the `'default_error'` attribute.

However, using the error message from the `ROOT` attribute is not the only source that `ScriptGenEreplay.py` can use. As previously described in Section 3.6.3.2, the `'--default_error'` option allows the default error message that is stored in the tree to be overridden. The option's value can be a file name, raw byte stream (with hex values escaped), `'#REPEAT#'`, or blank. Using a file name allows loading up to a 10KB message from a file. Not providing a value with the option or specifying `'#REPEAT#'` causes the server to repeat the last message sent whenever a default error message is requested. Currently, this message is re-sent as is without considering links.

3.6.3.10 Getting the server response.

Prior to sending out a server response, the raw marked up data must have data inserted into it. Specifically, linked data must first be inserted, and then environmental data must be inserted. While inserting linked data, a default error message is sent if the linked data cannot be found. This unusual scenario could occur if the client data is missing a marker, which infers that the regular expression is not specific enough. Once all of the linked data for a server message is successfully determined, it is inserted into the message in offset order similar to the environmental link insertion procedure described in Section 3.6.3.5. A cumulative difference is the offset is maintained here as well.

Finally, in the event that more than one server response is available to choose from, the one with the highest weight is selected. If the weights are equal, then a random selection is made.

3.7 Design Summary

In summary, this chapter describes the ScriptGenE Framework design. The framework is designed to be accurate, flexible, and efficient for emulating ICS devices. Its design is based off of techniques from ScriptGen, Roleplayer, and some novel contributions of this research.

The features unique to ScriptGenE include:

1. handles unknown transitions with novel backtracking algorithm
2. mimics long polling sessions through looping
3. sends default error message to maintain conversation when expected response is unknown
4. determines environmental *and* protocol dependency links while using unique filtering algorithms to remove trivial links
5. includes initial handling of partial (missed handshake) connections
6. provides stand alone server and client replay applications with flexible options

IV. Research Methodology

4.1 Goals

This research focuses on developing a framework for automatically configuring PLC emulators and providing a working proof of concept implementation. The goals in testing the implementation align with the design parameters discussed in Section 3.2. Namely, the following questions related to accuracy, flexibility, and efficiency must be addressed:

Accuracy

1. What percent of the time can the emulator produce traffic that is accepted as normal behavior by standard tools?
2. How similar are the bytes between a conversation using an emulator versus a conversation generated with a real PLC?

Flexibility

1. Can the same emulator configuration tools be used for different protocols without requiring any protocol-specific knowledge?
2. Can the emulator recover from errors or unknown transitions?

Efficiency

1. Can the protocol tree building and replay process be successful while being *completely* automated?
2. How many input traces are required for configuration in order to achieve a 100% acceptance rate by the tool interrogating the emulator?
3. How fast can the protocol trees be built?

4.2 Approach

A framework called ScriptGenE is developed that builds protocol trees by parsing conversations from network traces. The protocol trees are then loaded into a server that uses them to send responses from tree nodes based on the client requests that correspond to tree edges. The details of this process are covered in Chapter III.

Testing the framework involves generating network traffic of various ICS-related protocols and capturing traces of the traffic. Standard tools are selected for each protocol, and they are used to interrogate a PLC to create *reference packet captures*. Subsets of increasing order of these traces are then used to build emulator profiles. Next, instead of the PLC, each emulator profile is interrogated by the selected standard tools to create *experimental captures*. Measurements are taken on the interrogator's response to determine if the tool found the emulation acceptable. Measurements on the byte variability differences between the reference captures and the experimental captures are used to determine if they are statistically equivalent. Finally, logs are parsed or viewed to determine how fast trees are built, how well the trees are built, and if the emulator successfully recovered from unknown transitions.

Each protocol is tested in a different experiment since different build parameters, replay parameters, and workload are used. A web protocol and two PLC control protocols are selected for experimentation. Most PLCs use a web protocol and control protocol, and they are very different in structure and use.

4.3 System Boundaries

As shown in Figure 41, the system under test (SUT) is the ScriptGenE Framework. The components of the system are the protocol tree building software (ScriptGenE.py) and the protocol tree replay software (ScriptGenEreplay.py). Both of these components under test (CUT) provide metrics for evaluating the system.

The workload into the system is a set of unique *reference traces* of varying number. For each experiment, reference traces are generated using a different *task* against a PLC configuration. The input traces are then processed by ScriptGenE.py using a set of *build options* for each experiment. The resulting protocol tree is then imported into ScriptGenEreplay.py in order to generate *experimental traces* using a set of *replay options*.

The metrics to evaluate the system include three parts. First, ScriptGenE.py's log file is parsed for every emulator profile to collect the number of nodes that were processed per second. Second, the interrogator's response to the emulator is evaluated as a PASS or FAIL. Finally, the server response accuracy is determined by statistical analysis of the difference in variability between the reference traces and experimental traces.

4.4 Services and Outcomes

The service that the SUT provides is the automatic configuration of a PLC emulator. The specific emulator services are determined by the PLC tasks that are encompassed within the set of input traces.

There are three outcomes for each emulator service. These include: an accurately configured emulator, an inaccurately configured emulator, or a non-functioning emulator. An accurately configured emulator will always produce responses that the scripted interrogator task expects, and the resultant experimental trace will be statistically equivalent to the set of reference traces.

An inaccurately configured emulator, however, will not always generate the correct responses required. Thus, it *may* not always produce responses that the scripted interrogator task expects, and the resultant experimental trace *will* be statistically different than the set of reference traces. The latter is measured by byte differences

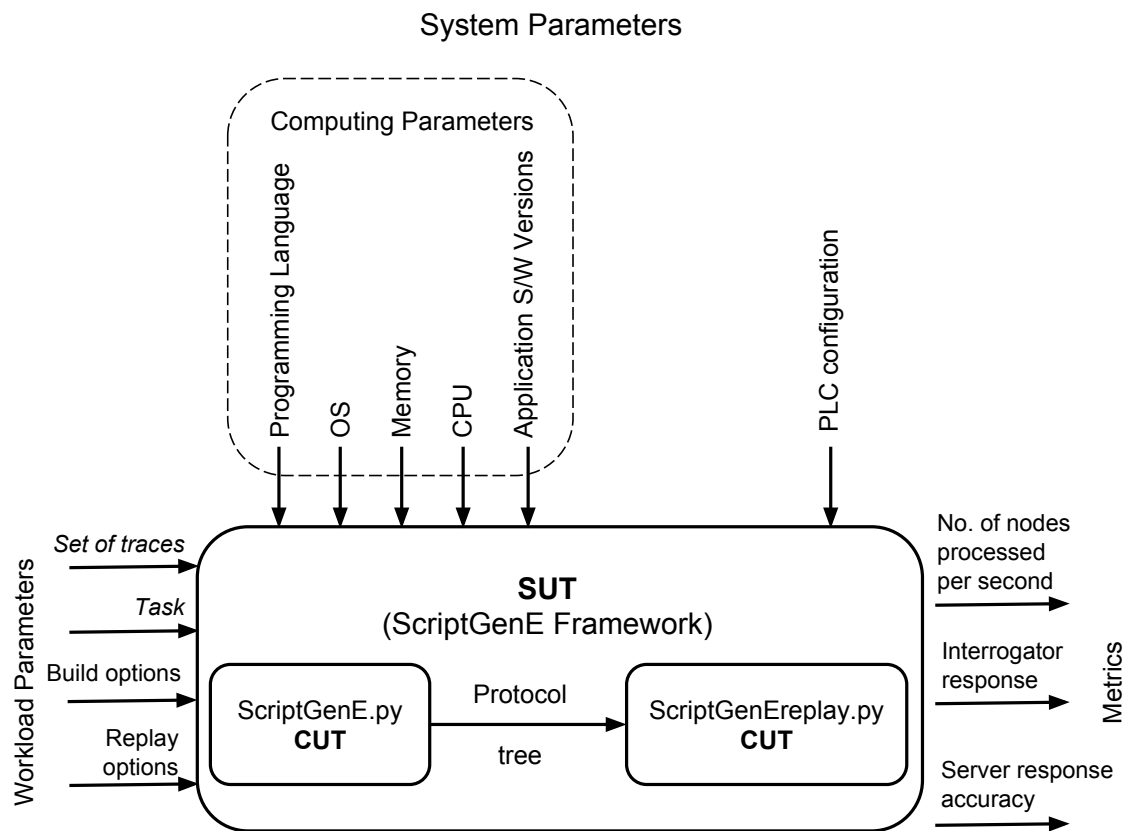


Figure 41. ScriptGenE Framework

per each message. It is assumed the interrogator sends equivalent request messages for a task when given a set of correct responses. Thus, any statistically significant byte differences are due to incorrect responses from the emulator.

Finally, the emulator is considered non-functioning if the P-tree build process does not result in a usable P-tree pickle file for ScriptGenEreplay.py to import. This can happen, for instance, if improper build options are used that filter out all of the necessary packets from the input traces. Improper replay options can also prevent the emulator from functioning properly.

4.5 Parameters and Factors

There are workload parameters and system parameters; parameters that vary are called *factors*. The values that these factors hold are called the *levels*. The remainder of this section describes the parameters and how they apply to the SUT.

4.5.1 Workload parameters.

4.5.1.1 Task.

One application task is selected for each of the chosen protocols to experiment on. In particular, `wget` is used to mirror a PLC's web server pages over HTTP, and RSLinx and STEP7 are used to browse for a PLC on the network and view its modules over EtherNet/IP and ISO-TSAP, respectively. The options used when running `wget` are:

```
-q -r --page-requisites --no-parent -w 0.2 --tries 1 $server
```

which does a quiet recursive download of all pages beneath the root directory of the selected `$server`. All page requisites, including images and style sheets, are also downloaded. Additionally, there is a wait time of 0.2 seconds *between* requests.

Pilot testing shows that the time delay is required to avoid overloading the PLC with requests. Finally, the `wget` options above also specify that each download is only tried once. If there are errors preventing a download, `wget` will quit. This option makes it more obvious if the emulator is not accurately replaying the expected responses.

Unlike `wget`, however, `RSLinux` and `STEP7` are applications driven by a GUI. `SikuliX` [Hoc15] is employed for this research in order to automate tasks using `RSLinux` and `STEP7`. More details on how `SikuliX` is included later in Section 4.9.

All of the above tasks are chosen because they are representative of actions an adversary might initially take when gathering information about a PLC. An attacker may, for instance, view or download a PLC’s web page in order to view system configuration information, uptime, etc. However, more information can be gleaned from interfacing the PLC with `RSLogix` or `STEP7`. If the attacker wants to learn about the ladder logic on the PLC, for instance, they can perform a program ‘upload’ to obtain a copy. Before this or other remote actions can be performed via `RSLogix`, however, `RSLinux` must be used to establish communication with the PLC (see Section 2.2.2). This task is accomplished by using `RSWho` to find the PLC on the network and populate a list of available modules. Similarly, in `STEP7` station objects (e.g., controller module information) must be uploaded to the programmer by configuring a network node.

Additionally, `HTTP`, `EtherNet/IP`, and `ISO-TSAP` are the chosen protocols for the experiments in this research because they represent different classes of protocols (see Section 2.2.5.2). Thus, experimenting with the protocols provides an indication of the SUT’s protocol-agnostic robustness.

4.5.1.2 Set of traces.

By performing the application tasks described in the last section, network traffic is generated. This traffic is randomly captured for each task run in order to generate a set of reference captures. Next, random combinations of n traces from the pool of reference captures serve as input to the SUT. Thus, both experiments use the number of traces per emulation profile as a factor. Each level is tested on a separate experimental run. The HTTP experiment uses two levels, and the EtherNet/IP and ISO-TSAP experiments use four levels.

For the HTTP experiment, 60 reference captures are generated for each experimental run. n is set at one for the first run, and two for the second run. Both experimental runs build 60 emulator profiles using the combinations of reference captures.

For the EtherNet/IP and ISO-TSAP experiments, 12 reference captures are generated for each experimental run. The value of n ranges from three to six captures per emulator profile. Additionally, 32 emulator profiles are generated per experimental run.

For each experiment, the number of traces per emulation profile are chosen based on design features and pilot testing results. For instance, pilot testing shows that intra-protocol dependencies are unimportant for emulating the PLC's HTTP responses. Thus, the importance of clustering for this particular task is found to be negligible. As a result, it is hypothesized that as little as one trace is required to accurately replay the HTTP responses. An additional run with two traces is also tested in order to test the effect of providing multiple server responses for a given request.

For the EtherNet/IP experiment, pilot testing shows that a minimum of two traces is enough to accurately emulate responses if the system's initial state is *consistent*.

This experiment, however, is *randomized*, so it is hypothesized that a minimum of three traces are required to accurately emulate the EtherNet/IP responses. More traces makes it less likely that the SUT will consider variable protocol fields as fixed regions. Thus, the cluster regular expressions will be more accurate. Using three traces also follows the intuition that an odd number of input traces is preferable, because an odd number of proposals more likely break any ties during the link consolidation phase (Section 3.6.2.6). Six was chosen as the highest number of traces per profile to test because it is double the value of the minimum level while still being within the design parameter of a maximum of seven traces (Section 3.2).

Finally, pilot testing shows that the traffic generated by the ISO-TSAP task has less variability than the EtherNet/IP experiment. It is hypothesized that only two traces are required to accurately emulate responses, so a range of two to five input traces is tested.

4.5.1.3 Build options.

The build options inputted into ScriptGenE.py for the HTTP experiment are

```
-p 80 -x -a -M 0
```

This tells ScriptGenE.py to filter traces on port 80, exclude creating tree pictures, show printable text in any RA output, and disable Macroclustering.

Macroclustering is disabled for the HTTP experiment for two reasons. One, as described in the previous subsection, clustering is not required to accurately emulate the responses. Two, the current implementation does not include the microclustering capability required to separate some of the GET requests into different groups. In particular, pilot testing shows that many of the GET requests identify desired resources by a single or two digit ID. Macroclustering is not fine grained enough to separate these GET requests appropriately, and clustering is made more difficult be-

cause nearly all of the requests are from a different session (i.e., positional clustering does not help). Macroclustering the server responses, on the other hand, does separate some of the requests as needed, but clustering all of the relatively large server responses is very slow. In fact, the average number of nodes processed per second is nearly 100 times greater if Macroclustering is disabled instead. For all of these reasons, Macroclustering is disabled for this experiment.

Next, the build options for the EtherNet/IP experiment are

```
-p 44818 -x -M 0.5
```

This tells ScriptGenE.py to filter traces on port 44818, exclude creating tree pictures, and use 0.5 for the Macroclustering threshold.

Macroclustering in this experiment is set at ScriptGenE.py's default value. It should be noted, however, that the default value for ScriptGenE.py is selected based on pilot testing various protocols and also findings in previous research [LMD05, BGH14]. In fact, this is the reason why the experiment does not use the Macroclustering threshold as a factor; a sane default is determined for clustering protocols in general.

Finally, the build options for the ISO-TSAP experiment are

```
-p 102 -x -M 0.5
```

The only difference from the EtherNet/IP experiment options is the port that is being targeted.

4.5.1.4 Replay options.

The replay options inputted into the ScriptGenEreplay.py script for the HTTP experiment are

```
-i $IFACE -f --strict -v
```

This tells ScriptGenEreplay.py to use the value of variable ‘\$IFACE’ for selecting the interface. Additionally, the server will run forever (‘-f’) or until instructed to shutdown by a keyboard interrupt. Third, the ‘--strict’ option is used to disallow any backtracking. Finally, the verbose switch enables capturing hex dumps of the client message in INFO messages.

For the HTTP experiment, it is important that the server stay up, because each wget session has multiple connections. Unfortunately, this means the server must be forcefully brought down after the task is complete. Next, backtracking is disallowed to ensure that the emulator will send the default error message and close the connection for any unknown request. Pilot testing shows that the majority of all RRP’s for this task are separate TCP connections, so there is no benefit to searching other branches of the tree (all of them spread out from ROOT).

The replay options for the EtherNet/IP experiment are

```
-i $IFACE -v -d
```

The difference from the previously shown options for HTTP is that backtracking is allowed (by default), the server will shutdown after a connection ends (by default), and the default error message is set to repeat the previous server message.

Ideally, the server will shut itself down after every task is complete. Since pilot testing shows that the network traces for the EtherNet/IP task include only one TCP connection, it is safe to have the server shutdown after a connection closes. This prevents having to forcefully bring down the server via an interrupt, which takes more time.

Next, the default error message is set to repeat previous server messages in order to test this unique feature. Additionally, pilot testing shows that the default error message selected based on the algorithm in Section 3.6.2.5 is unsatisfactory for Ether-

Net/IP traces. It is hypothesized that limiting the message selection to only unlinked responses is a contributing factor.

Finally, the replay options for the EtherNet/IP experiment are

```
-i $IFACE -f -v -d
```

The only difference from the EtherNet/IP replay option is that the server will not shutdown after a client disconnects. Just like in the HTTP experiment, multiple connections must be handled for every task conversation.

4.5.2 System parameters.

As shown previously in Figure 41, the system parameters consist of the computing parameters and PLC configuration. The former will be discussed in the experimental setup (Section 4.9). The PLC configurations, however, are as follows:

Allen-Bradley ControlLogix5561 (L61) PLC

- Firmware version 19.015
- Slot 0 - EtherNet/IP ENBT
- Slot 1 - L61 Controller with mode set to REM Run (remote Run)
- Slot 6 - DC Output

Siemens SIMATIC S7-300 PLC

- Firmware version 3.2
- Slot 2 - CPU 315-2 Controller with two Ethernet ports
- Slot 3 - Discrete I/O (DI8/DO8)
- Slot 4 - Analog I/O (AI4/AO2)

Note, REM (remote) PLC modes allow RSLogix to change the mode of the PLC when it is online with the software. However, the current experiment with RSLinks does not require this functionality. Remote Run is thus equivalent to the normal Run mode for the purposes of this experiment. Finally, the Run mode is more desirable than the Program mode because the former allows ladder logic to execute. This allows the PLC to affect changes in the DC Output module, which may be reflected in interactions with the PLC via the network. Thus, having the PLC running potentially provides more variability in the network traces. The program executed on the L61 performs a lighting demo by providing power for lights (in a pattern) from the DC output module.

4.6 Performance Metrics

This research uses three performance metrics. The first one measures how quickly protocol trees are built for a given task. Specifically, the total wall clock time and the number of nodes processed per second is extracted from each build log.

The second performance metric measures how well an emulator profile produces traffic that is accepted as normal behavior by standard tools (i.e., `wget` and `RSLinx`). In both cases, each emulator's performance is graded as a PASS or FAIL based upon output of the interrogating tool. If the task completes without errors, then it is considered a PASS. Otherwise, it is graded as a FAIL.

In the case of the HTTP experiment, the graded output is the downloaded web content. Web content downloaded from the PLC serves as a baseline that the downloaded content from the emulator can be compared to. If the content is evaluated to be complete based on the comparison to the baseline group, then the task is considered a PASS.

For the EtherNet/IP and ISO-TSAP experiments, the graded output is the RSLinx and STEP7 GUIs, respectively. For instance, if the RSWho window shows the emulator as a “valid” PLC with all of the correct modules populated, then the task is considered a PASS. If any error messages or icons are shown, or if the task does not complete, then it is considered a FAIL.

The third performance metric measures the accuracy of the traffic produced by the emulator by assessing byte differences between reference and experimental captures. In contrast to previous research [Jar82, Fin90], however, the metric used in this research does not attempt to *whitelist* any bytes as acceptable because they are known to belong to stateful protocol fields (i.e., fields that are expected to vary). There are two reasons for this. One, the whitelist method requires that *all* of the variable regions are known and accounted for. This may not always be the case, and this research is intended to be protocol-agnostic. Therefore, it makes sense for the evaluation techniques to be protocol-agnostic as well. Secondly, just because certain bytes in a protocol field *can* vary does not mean that they *will* vary during typical sessions. Indeed, even if they do vary, the rate of their variance and the values they take on is likely to be *specific* to their use in the session. As an example, consider a typical number counter. All of the digits can vary, but the least significant digits change much faster than the most significant digits. If the counter is big enough and resets between sessions, the most significant digits may *never* vary during typical sessions.

Therefore, the metric adopted in this research for byte-level accuracy is to measure the *differences* in byte variance between captures. This is accomplished by first establishing a baseline by measuring the byte variances between all reference captures. Then, each experimental capture is compared to each reference capture. If the differences of the two groups are statistically equivalent, then the emulator is determined to be accurate.

Finally, it should also be noted that previous research [Jar82, Fin90] also evaluated honeypots using Nmap fingerprinting and timing. This research, however, focuses on the application layer. As a result, both Nmap fingerprinting and packet timing are out of scope.

4.7 Experimental Design

4.7.1 Overview.

The experiments are designed to address the questions posed in Section 4.1. In order to test how well the SUT can handle various protocols, HTTP, EtherNet/IP, and ISO-TSAP are chosen for experimentation. Randomness is introduced into the experiments in order to test accuracy and flexibility. Additionally, the experiments are completely automated, and they test various numbers of input traces to determine how many are needed to achieve accuracy. The next two subsections discuss how variability is introduced into the experimental design and how the number of samples is chosen.

4.7.2 Introducing variability.

As previously discussed, measuring emulator accuracy for this research requires assessing differences in the variability of bytes between reference and experimental captures. Ideally, the variability present in the reference traces is equivalent to that of traffic generated during normal application use. Additionally, byte variability is necessary for the emulator to be most effective since this is what allows ScriptGenE.py to detect mutating regions. Thus, the number of traces required in order to build a robust emulator is directly proportional to the variability of the input traces.

As a result, variability is purposely introduced while assessing the SUT. Each task's packet captures are *randomly* created for both the reference and experimental

groups. Additionally, the protocol trees are built using random combinations of the reference captures. Third, software and hardware are purposely not reset to a known state unless necessary. The PLC, for instance, is in a running mode and is never reset during experiments.

RSLogix, however, does have to be reset occasionally. This limitation is due to the fact that RSLogix continually increments a counter throughout sessions. Thus, at a minimum, RSLogix must be reset once after completing the reference captures. Otherwise, all of the experimental runs will potentially use higher counter values that are not represented in the reference traces. The same problem can occur if the randomly selected reference traces were both captured around the same time period. To account for this limitation, RSLogix is reset after a random number of task runs. The random number's range is bounded from roughly 33% to 50% of the total expected number of reference task runs. The expected number of reference task runs is based on the total number of captures required and the probability a capture will occur.

4.7.3 Determining the number of required captures.

Next, the number of samples required for the experiments is determined in two ways. For the HTTP experiment, pilot testing shows that the number of bytes in the downloaded HTML directories have a standard deviation of about 1.15 bytes. This is used with a two sample power test in R (statistical software) to determine the number of samples required to achieve 95% confidence of a one byte difference with 90% power. The result shows that 59 samples are required. 60 reference samples are randomly generated for the HTTP experiment, which yields 60 possible emulation profiles when choosing one capture and 1770 possible emulation profiles when choosing two captures. 60 emulation profiles are created from the reference captures for each

level of traces. Thus, for the HTTP experiment 120 total reference and experimental captures are randomly generated. The chosen capture probability is one in three traces.

For the EtherNet/IP and ISO-TSAP experiments 60 samples are also desired for achieving 95% confidence, but the primary samples of consideration are comparisons. Therefore, the number captures is determined by the number of combinations required. 12 reference captures are randomly generated. This yields 66 comparisons for the reference group. Additionally, the reference capture pool yields 220, 495, 792, and 924 possible emulator profiles, respectively, for each trace level. 32 emulator profiles are selected from each of these groups, yielding 384 comparisons using the experimental captures. In total, for each of the two experiments, 48 reference captures and 128 experimental captures are randomly generated. The chosen capture probability is one in five traces.

4.8 Evaluation Technique

Overall, the results are evaluated by performing a statistical analysis of the metrics in R [R14]. Time elapsed and the number of nodes processed per second is evaluated by performing a one-sample t-test and computing standard statistics such as the standard deviation, mean, and 95% confidence interval.

Next, as previously discussed in Section 4.6, the PASS/FAIL evaluations for task output is graded differently for each experiment. For evaluating `wget`'s response, one measure to gauge whether or not the task completed is to simply see if the expected number of bytes were downloaded. Since the replay and `wget` settings are setup so that any error will cause the task to exit, this is a reasonable measure of completeness. The command used to accomplish this is `'du -sb'`. Any HTML directories downloaded from the emulators that have less bytes than the minimum number downloaded from the PLC is considered a FAIL.

A more accurate measure, however, is to find any byte differences between downloaded content in reference directories and experimental directories. For static web sites, a simple recursive `diff` between two downloaded site copies will yield no differences if the emulation is perfect. However, the web server on the L61 is a *dynamic* web site. As a result, performing a simple recursive `diff` as described will yield many differences. However, these differences are usually in the same locations and occur at the same frequencies. Therefore, a method of measuring if the `wget` task completed is to use a similar technique as the byte accuracy measurement. Instead of comparing byte-level variability, however, the number of different and identical files is compared between the reference and experimental groups. This is done using a combination of `diff` and `grep` as shown in Figure 42. As a result, this higher level abstraction is used to gauge if the `wget` tasks completed prior to analyzing the byte-level accuracy using network traces.

```
diff=$(diff -qsNaur ${d2} ${d1} | grep -c differ) # num different files
ident=$(diff -qsNaur ${d2} ${d1} | grep -c ident) # num same files
```

Figure 42. Calculating the number of different and identical files between a pair of directories

The other PASS/FAIL evaluation is for output response of RSLinx and STEP7 in the EtherNet/IP and ISO-TSAP experiments. By default, all tasks are assumed to be failures unless they complete successfully with the modules being populated as expected. In RSWho, the modules expected are shown in Figure 43. In particular, modules 00, 01, and 06 must be present on the backplane as shown in the figure. Figure 44 shows an exact image of how the corresponding modules are expected to be seen in STEP7.

The third metric, byte-level accuracy, is evaluated by performing a two-sample Mann-Whitney-Wilcoxon test to compare the variability between the reference and experimental groups (normal distributions are *not* assumed). The variability in this

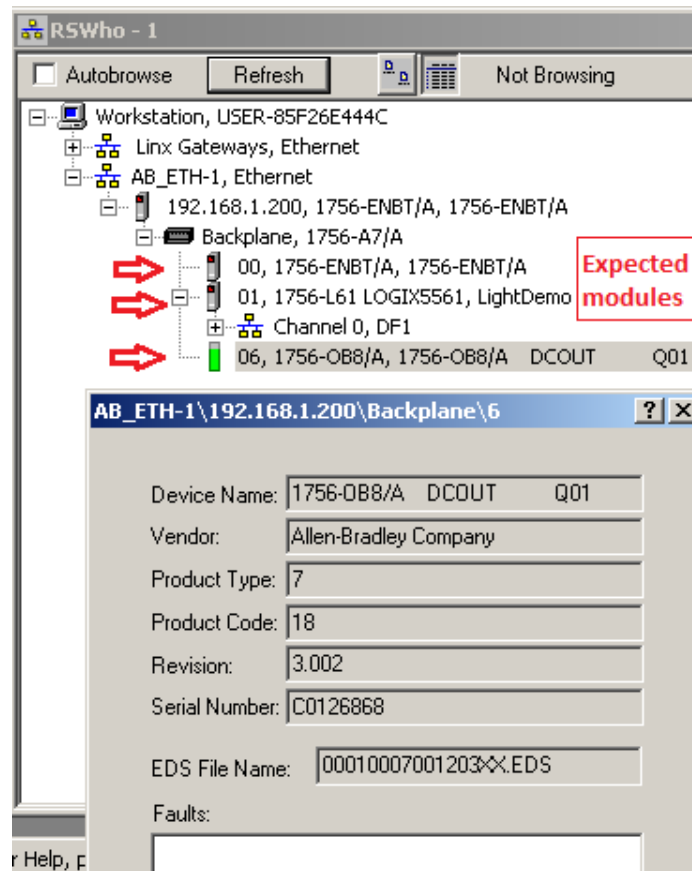


Figure 43. A successful emulation session in RSLinux

Slot	Module	Order number	Firmware	MPI address
1				
2	CPU 315-2 PN/DP	6ES7 315-2EH14-0AB0	V3.2	2
X1	<i>MPI/DP</i>			<i>2</i>
X2	<i>PN-IO</i>			
X2	<i>Port 1</i>			
X2	<i>Port 2</i>			
3				
4	DI8/DO8xDC24V/0.5A	6ES7 323-1BH01-0AA0		
5	AI4/AO2x8/8Bit	6ES7 334-0CE01-0AA0		

Figure 44. A successful ISO-TSAP emulation session shows these modules in STEP7

case is defined as the percentage of different bytes between two captures files. Any significant differences must be accounted for. Note, differences in the number of connections are not counted, but differences in the number of messages are counted.

Additionally, the number of different bytes per each client and server *message* is evaluated to determine where the conversation changes, how much it changes, and if the changes are significant. An example of the variability found in reference group server messages is shown in Figure 45. Each dot in the scatterplot represents the number of different bytes found when comparing two server messages from different reference captures. The x-axis corresponds to each of the 35 server messages in the trace files. In total, there are 12 captures represented in the figure, yielding 66 comparisons. As can be seen in the figure, the first server message never varies, and the second message varies by one byte. The remainder of the conversation varies consistently between three and six bytes until the near the end (at which point two byte differences are also included).

The maximum variability expected from the reference group is then subtracted from the corresponding experimental group. Any number of bytes remaining are *potentially incorrect* and should be investigated. Referring back to Figure 45, any byte differences more than six bytes for the third message and beyond would be unexpected differences unless they can be attributed to environmental links described in Section 3.6.2.4 (e.g., the PLC IP address ended in ‘.206’, but the emulator IP address ends with ‘.200’ resulting in a difference of one byte). If the byte differences are due to environmental links, then this effectively causes the overall percentage of different bytes to be uniformly increased. This shift is corrected by subtracting the percentage of byte differences due to environmental links.

Overall, the described approach is similar to the whitelist approach previously discussed in Section 4.6, but it is done in a way that automatically detects how many

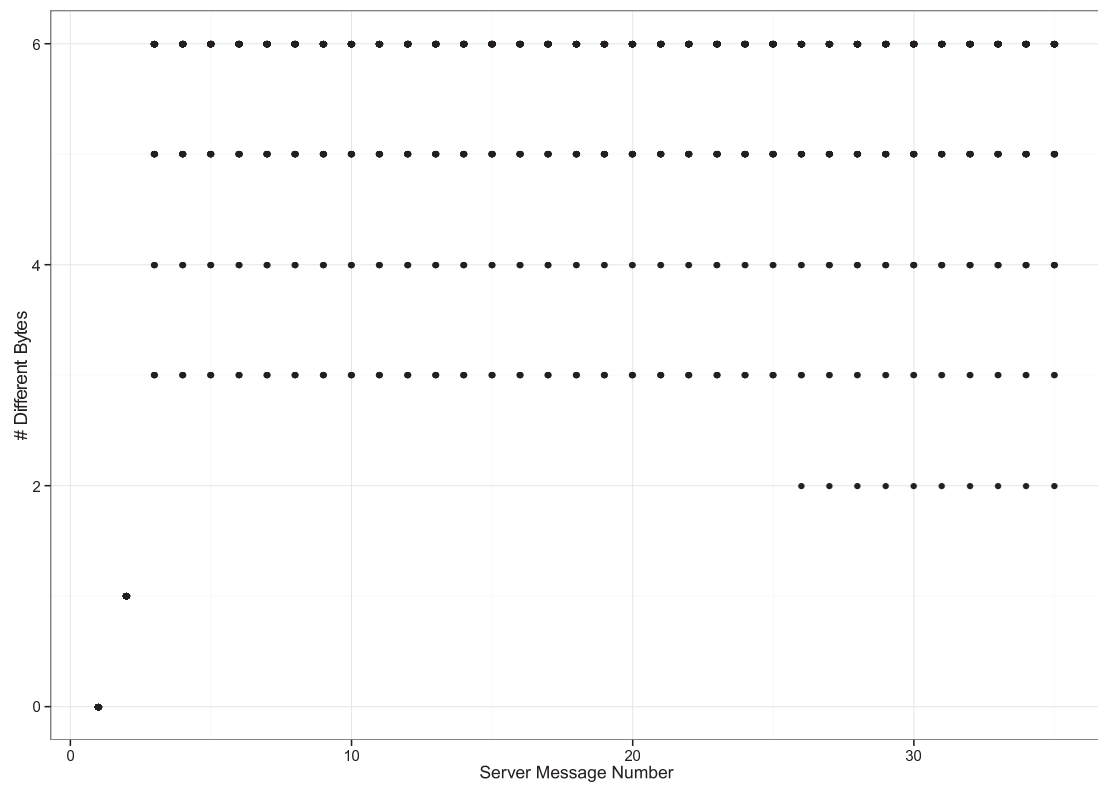


Figure 45. Number of different bytes per server message as compared to other reference captures

bytes per message should be whitelisted. Finally, by using this method to key in on the different messages of interest, byte-level differences can be examined in the accompanying `diff` outputs of the hexdumps.

Evaluating failed tests. Finally, for both experiments, any test cases that fail are evaluated to determine the cause. This involves checking the logs for error and warning conditions, along with examining message differences. Unknown transitions in the replay log, for instance, should be investigated. The tools described in Section 4.9.3 provide the means to accomplish the required analysis. Additionally, Wireshark is used to manually inspect packet captures.

4.9 Experimental Setup

4.9.1 Overview of experimental setup.

The experimental setup, as shown in Figure 46, includes three physical machines. A L61 PLC, S7-300 PLC, and a HP laptop are connected by a Cisco SG 100D-08 switch. The laptop hosts three VMs whose experimental network interfaces are bridged from the laptop's physical interface. The Honeydrive Linux VM contains the SUT and runs `wget`. The Windows XP VM on the left runs RSLinx, and the other Windows XP VM runs STEP7. Because the VMs share a physical network interface, they can see each other's network traffic as if they were connected by a hub. Thus, all packet captures can occur on one VM. Network traffic is captured using `tshark` with a capture filter set to the port of interest. Next, the Linux VM is used to run the experiments and collect all data using a set of custom scripts. Finally, in order for RSLinx and STEP7 to be fully automated, SikuliX scripts are used. These are controlled remotely by a simple server setup on each of the Windows XP VMs.

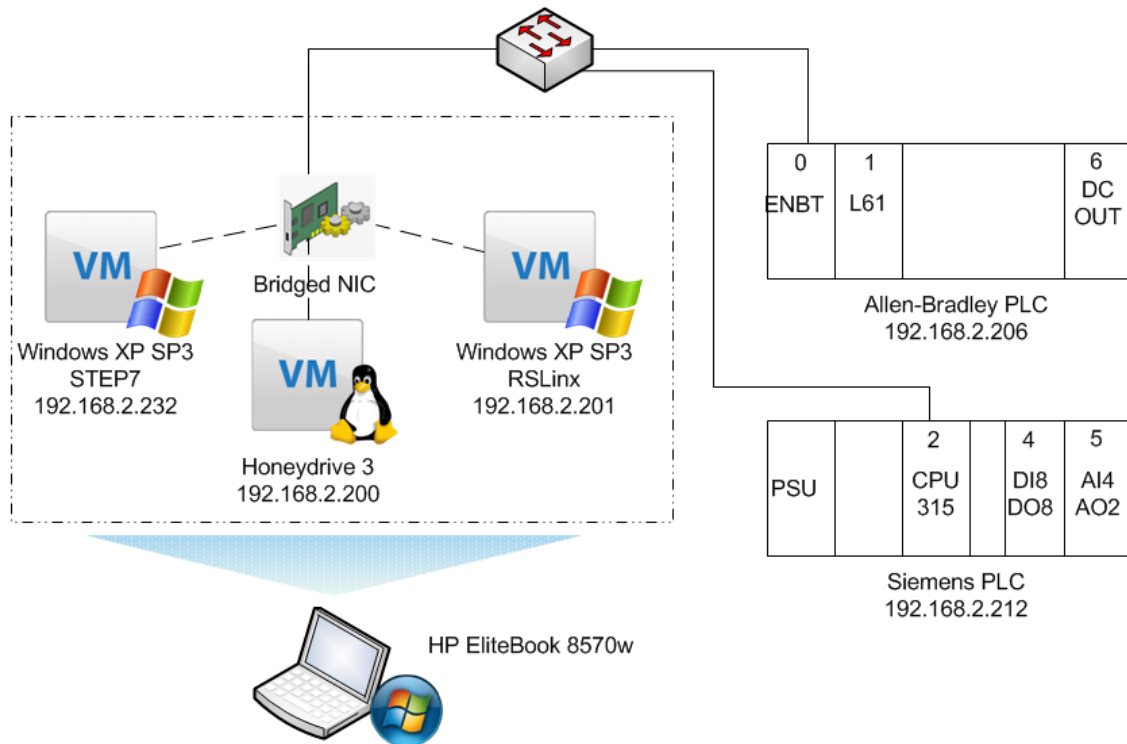


Figure 46. Experimental Setup

4.9.2 Machine configurations.

As discussed in the overview the experimental setup uses three physical machines and three VMs. The PLC configurations were discussed in Section 4.5.2. The laptop configuration is as follows:

Hewlett-Packard EliteBook 8570w

- Microsoft Windows 7 Service Pack 1
- 2.6GHz Intel Core i7-3720QM processor (4 cores)
- 16GB RAM
- Plugable USB 2.0 Ethernet adapter
- VMware Workstation version 10.0.1 build-1379776

Next, the honeypot VM is a distribution of Linux called Honeydrive [Kon14]. Honeydrive is a third-party variation of Xubuntu LTS that adds over 10 pre-installed and pre-configured honeypots packages (such as conpot discussed in Section 2.3.1). Additionally, it contains many scripts and utilities for analyzing logs and malware, forensics, and network monitoring. As such, it is a natural choice for honeypot research. The Honeydrive VM settings and relevant software versions for this research are listed in Table 1.

Table 1. Linux VM configuration

Honeydrive 3 (Royal Jelly)		
2 processor cores	Linux kernel 3.2.0	bash 4.2.25
2 GB RAM	Python 2.7.3	tshark 1.6.7
80 GB HD space	gcc 4.6.3	wget 1.13.4

Additionally, authbind is used to allow a regular user to run a server on privileged ports. This is needed for both HTTP and ISO-TSAP emulation. To set up authbind for port 80, for instance, a blank executable file must be created once at `/etc/authbind/byport/80`.

Next, the Windows XP VMs include many special vendor software applications such as the RSLogix suite. The Windows XP VM settings and relevant software versions for the EtherNet/IP experiment are listed in Table 2. The VM used for the ISO-TSAP experiment has the same virtual hardware and utility software, but the VM has STEP7 software version 5.5 installed instead of the RSLogix suite.

4.9.3 Experimental scripts.

Several scripts are used to run the experiments and aggregate collected data for analysis in R [R14]. The scripts developed are as follows:

Table 2. Windows XP VM configuration

Windows XP Service Pack 3		
2 processor cores	RSLogix V19.01.00	python 2.7.2
3 GB RAM	ControlFLASH	Java 1.7.0u60
60 GB HD space	RSLinux Classic 2.59.02	SikuliX 1.1.0

ScriptGenEdir.py

- Runs ScriptGenE.py using random combinations of Pcaps from a directory

diffPcapDir.py

- Uses diffPcaps.py to collect `diff` data on all combination of reference and experimental Pcaps in a directory.
- `diff` data is exported into text files for import into R statistical software.

enip-exp.sh

- Runs the EtherNet/IP experiment
- Uses *RobotServer* to control RSLinx remotely

wget-exp.sh

- Runs the HTTP experiment

wget-results.sh

- Computes the measurements for the HTTP experiment:
 - Pcap differences
 - Number identical / different HTML files
 - HTML directory sizes

RobotClient.py

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

kill_Robot.sh

- Helper script that shuts down the RobotServer

4.9.4 Graphical User Interface automation.

In addition to the scripts introduced in the prior subsection, there are several SikuliX scripts that are used to control RSLinx. SikuliX allows anything that can be seen on the screen of a computer to be automated [Hoc15], and it is the successor of the original Sikuli Script project developed in 2009 [YCM09]. SikuliX uses OpenCV image recognition software to identify GUI components and optionally Tesseract for text recognition. SikuliX can emulate mouse and keyboard events. What makes SikuliX really powerful, however, is that the actions it performs are scripted using Jython or JRuby (Java variants of Python and Ruby, respectively). This allows the scripts to be easily integrated with other software. Additionally, SikuliX includes an Integrated Development Environment (IDE) that aides in taking screenshots and using the API correctly.

Five SikuliX scripts are created that together manage the RSLinx drivers and run the RSWho task. Four additional scripts control STEP7. Finally, separate batch scripts restart RSLinx and the STEP7 node drivers. These are all controlled by RobotServer.py. The server has a limited set of tasks that it performs. Each request must be preceded by 'RUN'. The set of tasks are:

Driver 'STOP_DRIVER', 'START_DRIVER', 'RESTART_DRIVER',
'DEL_DRIVER', 'CONFIG_DRIVER'

RSlinx 'SHUTDOWN', 'START', 'RESTART'

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

Conversations 'RSWHO', 'STEP7BROWSE'



During the experiments the server is controlled by RobotClient.py, but it can just as easily be controlled by netcat. Figure 47 shows an example session controlled by netcat that clearly shows how the server interacts with a client. As can be seen in the figure, the server echoes back the command it runs and replies back with status after completion.

```
$> netcat 192.168.1.201 13370
run shutdown
RUNNING cmd 'sc stop rslinx'...
PASS
run shutdown
RUNNING cmd 'sc stop rslinx'...
FAIL
run start
RUNNING cmd 'sc start rslinx'...
PASS
run start
RUNNING cmd 'sc start rslinx'...
FAIL
run rswho -o -d 192.168.1.206
RUNNING cmd 'runsikulix.cmd -r RSWho.sikuli -- -o -d 192.168.1.206'...
PASS
DIE
Shutting down after you disconnect...
^C
```

Figure 47. Example RobotServer session controlled by netcat

Next, the RSWho script run in the EtherNet/IP experiment is set up so that the traffic it generates is controlled. The main issue is that the 'autobrowse' setting in RSWho will automatically browse the network and PLC components. Thus, the script turns that setting off. However, autobrowse is automatically reenabled when

the RSWho window regains focus. To prevent extraneous traffic generation, the RSLinx driver is deleted and reconfigured between runs. The RSWho command seen in Figure 47, for instance, configures a driver for IP address 192.168.1.206. With ‘autobrowse’ disabled, traffic is then generated every time the ‘refresh’ button is clicked in RSWho. The script looks for certain images to determine if the task run is a PASS or FAIL. Figure 48 shows a script snippet opened in the SikuliX IDE that does this. The image as shown in the figure that designates a PASS is an image of the backplane with the three PLC modules shown.

```
# Look for '?' or 'X' - signs of failure
if not exists( and not exists():
    # Refresh ENBT
    type(linxGateway, Key.DOWN + Key.DOWN + Key.RIGHT)
    type(Key.DOWN + Key.RIGHT)
    type(Key.DOWN)
    click(refresh)
    sleep(3)

if run_tests:
    run_module_tests(rswwho_window)


if exists(): # PICTURE OF SUCCESS :)
    result = "PASS"
else:
    Debug.user("FAIL - final modules cannot be found")
```

Figure 48. RSWho SikuliX script snippet

4.9.5 Configuring and running experiments.

Both experiments are run via the shell scripts introduced previously. Each experiment’s script begins by loading a corresponding configuration file (wget-exp.conf and

enip-exp.conf). These files control all of the options described previously. In addition, they define the output directories, IP addresses, ports, and more. If desired, singular configuration options can be overridden at runtime. For example, the following command changes the number of experimental captures to collect to six:

```
NUMPCAPS=6 ./enip-exp.sh &> ../logs/enip-exp.log
```

4.10 Methodology Summary

In summary, this chapter describes the methodology used to measure the accuracy, flexibility, and efficiency of the ScriptGenE Framework. This is accomplished by running three experiments that use different workloads and an increasing number of network traces to build and test emulators.

All three experiments use various scripts developed for this research in order to autonomously run application tasks, collect packet captures, build protocol trees, test emulator profiles, and process data. Application tasks are run repeatedly against an Allen-Bradley L61 PLC or Siemens S7-300 PLC to generate network traffic. A set of reference packet captures are randomly generated, and then random combinations of these captures are used with a set of build options to create various emulator profiles. Each emulator profile is then used with a set of replay options in order to emulate PLC responses during task sessions. This generates experimental captures.

The experiments are first evaluated by measuring how much time elapsed while building the trees and how many nodes per second can be processed during each protocol tree build. Second, byte-level variability is measured in the groups of reference captures, and this is compared to the differences in bytes between experimental and reference captures. Third, each task is evaluated as a PASS or FAIL based on its successful completion. The first experiment uses `wget` to generate HTTP traffic,

and downloaded HTML files are evaluated to score each task run as a PASS or FAIL. The other two experiments uses RSLinx or STEP7 to generate traffic, and the task runs are considered a PASS if PLC modules can be browsed in the GUI.

V. Results and Analysis

5.1 High-level Summary

Table 3 below shows the results summary for both experiments. As can be seen in the table, all of the P-trees can be built in 6.074 seconds or less. This is well below the 5 minute requirement. Next, the task pass rate shows that only one trace is needed to emulate HTTP well enough for `wget` to download the same files. Additionally, only five EtherNet/IP traces are required to correctly populate all PLC modules in RSLinx 100% of the time. It should be noted that all passing tasks for HTTP and ENIP have *zero* incorrect bytes. The ISO-TSAP runs all passed (even with only two traces), but some of the experimental traces have two different bytes that are potentially incorrect.

Table 3. High-level summary of results

Protocol	Captures	Average Time (sec)	Task Pass Rate	% Difference		
				95% conf interval		p-value
HTTP	1	5.549	100%	-0.0057	0.0044	0.8836
	2	6.074	100%	-0.0127	-0.0047	5.652e-05
ENIP	3	2.634	75%	0.0125	0.0135	0.0009
	4	2.799	93.75%	0.0117	0.0118	3.755e-05
	5	2.963	100%	0.0103	0.0102	5.992e-06
	6	3.168	100%	0.0168	0.0178	0.0259
ISO-TSAP	2	0.323	100%	0.0100	0.0100	< 2.2e-16
	3	0.404	100%	3.524e-06	0.0100	2.461e-11
	4	0.501	100%	0.0100	0.0100	< 2.2e-16
	5	0.643	100%	0.0100	0.0100	4.789e-14

The percent differences in bytes between passing experimental and reference group traces generally show that the variation in the traffic produced by the emulators is different than what is produced by the PLC. The one exception is the HTTP experimental run using one input capture per emulator. For the other data sets, the differences are always less than 0.018% in the 95% confidence intervals. For EtherNet/IP, this equates to less than one byte per session. The maximum differences in the ISO-TSAP traces are one byte per TCP connection.

5.2 Experiment #1 - HTTP

5.2.1 Build efficiency and qualities.

Tables 4 and 5 summarize how long it took to build the P-trees. With 95% confidence, it can be expected that each tree will build in less than 6.1 seconds. Tables 6 and 7 show how many nodes per second can be processed while building protocol trees for the HTTP experiment. Most processing time while building protocol trees is generally spent clustering and aligning, but Macroclustering is disabled in this experiment. Thus, the P-trees are built very quickly. The average RRP size is about 2.4KB over 60 RRP. For the trees built from one Pcap, no edges are combined, but environmental links are found. For the trees built from two Pcaps, exactly half of the edges are combined during the initial tree consolidation. This means that each node in the tree has two equally weighted responses to randomly choose from.

Finally, the automatically selected default error messages are tiny image files (either ‘spacer.gif’ or ‘border.gif’). A 404 error page is present in the traces in response to requests for robots.txt, but the error page is stylized and is much larger than the tiny image files. In pilot testing, the current heuristic picked correct HTTP error messages by default, and in other cases it picks a short redirect page. Future enhancements to the default error message heuristic may be needed.

Table 4. HTTP protocol tree build time (sec)

traces	mean	std dev	min	Q1	median	Q3	max
1	5.549	0.054	5.480	5.510	5.540	5.570	5.780
2	6.074	0.083	5.970	6.028	6.055	6.093	6.350

Table 5. HTTP protocol tree build time confidence intervals

traces	95% conf. interval	
	lower	upper
1	5.535	5.563
2	6.053	6.096

Table 6. HTTP protocol tree throughput (nodes/sec)

traces	mean	std dev	min	Q1	median	Q3	max
1	10.813	0.103	10.376	10.770	10.831	10.881	10.950
2	19.762	0.264	18.906	19.693	19.816	19.924	20.101

Table 7. HTTP protocol tree throughput confidence intervals

traces	95% conf. interval	
	lower	upper
1	10.787	10.840
2	19.693	19.830

5.2.2 Successful task completions.

As shown previously, 100% of the `wget` tasks resulted in a PASS. However, initially this does not appear to be the case. Figure 49 shows that some of the HTML directories generated by the one-trace emulators are smaller in content than those downloaded from the PLC. The reason for this is that 5 of the 60 reference captures are missing at least one packet. Thus, the five emulator profiles that use those input traces omit the data in their responses. The same issue occurs in the two-trace emulators because 1 of the 60 reference captures is missing a packet. Since the issue is a problem with `tshark` and not the SUT, the tasks are scored as a PASS.

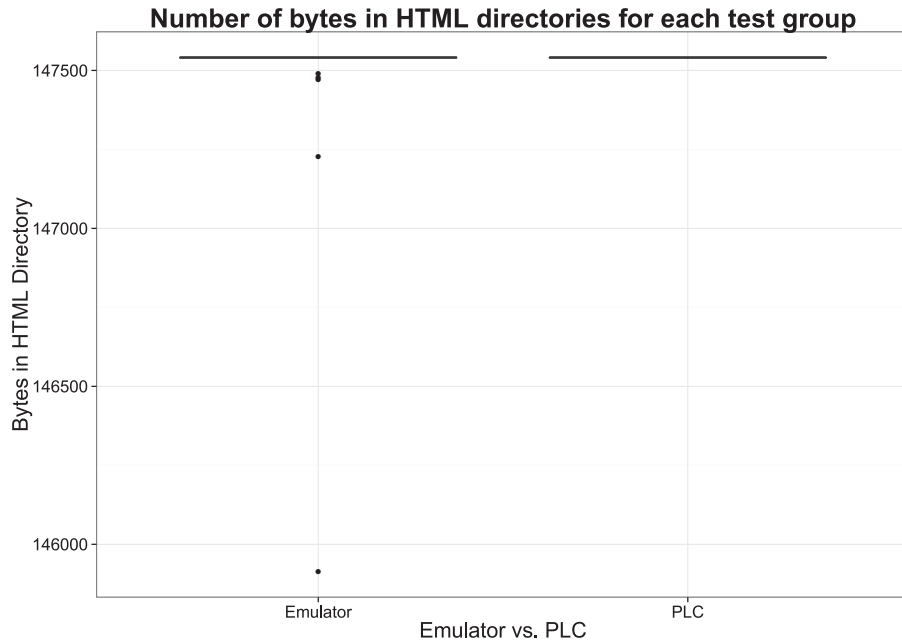


Figure 49. Number of bytes in HTML directories for each test group

It is unknown why `tshark` dropped packets. It is plausible that too many intensive processes were running on the physical host at the time, causing `tshark` or the VMware network driver to have difficulty keeping up. It is recommended that future work use separate physical machines instead of VMs.

Finally, the file diff analysis shows that the emulators generally have one more different file than the reference directories. This is due to a difference in environmental information. Specially, the root web page contains the hostname, which is the same as the server IP address in this experiment.

5.2.3 Byte-level variability.

In fact, environmental links play a big role in the HTTP tasks. Figure 50 shows that there is consistently at least one byte that is different in all of the client messages seen during the `wget` tasks. All of the GET requests contain the hostname. Thus, at least one byte (192.168.2.200 versus 192.168.2.206) is expected to be different for every client message. Additionally, all client messages after the first three use a referrer field which also contains the server IP address.

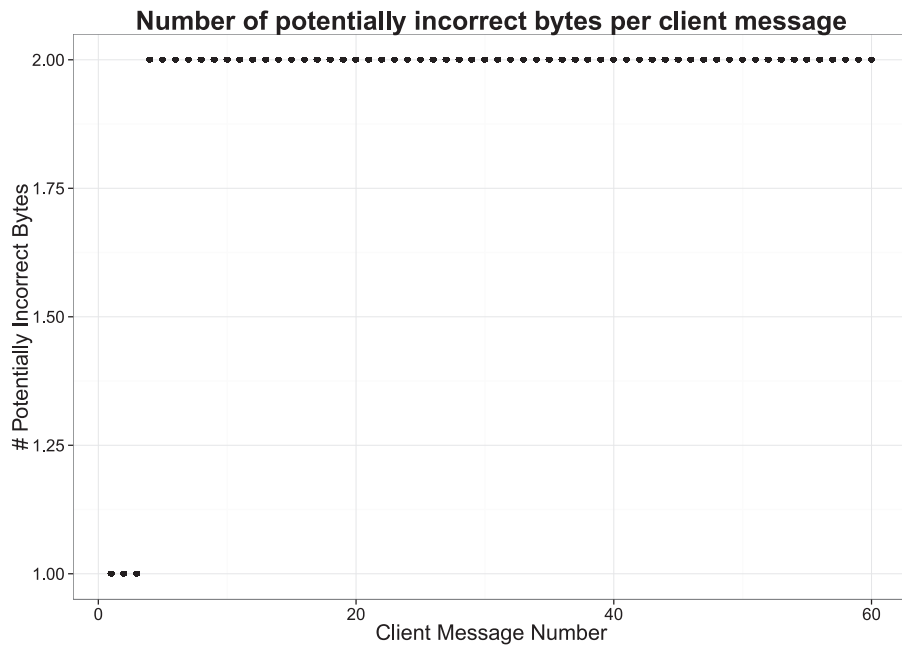


Figure 50. Number of potentially incorrect bytes per client message found in experimental traces - all attributed to environmental links

Next, Figure 51 shows that three of the server messages generated by the emulators have different bytes. As previously mentioned, the root web page (first message) contains the hostname. The message is actually a HTML redirect page, and it contains two copies of the hostname. The other two messages contain the server IP address. Overall, all of the byte differences in the client and server messages are due to differences in environmental link values. Thus, there are no incorrect bytes generated by the emulators.

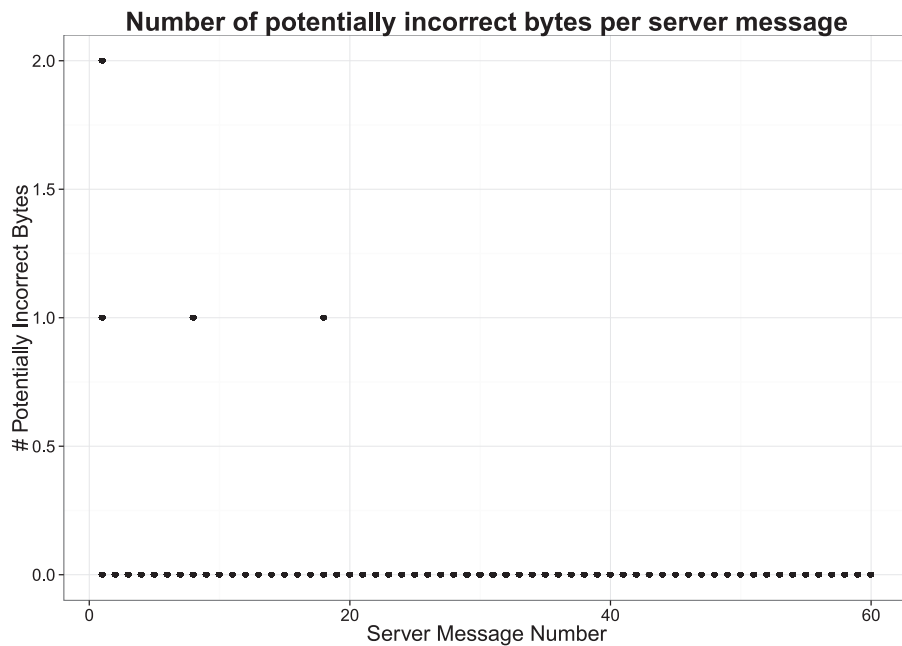


Figure 51. Number of potentially incorrect bytes per server message found in experimental traces - all attributed to environmental links

Figures 52 and 53, along with corresponding Tables 8 and 9, show the overall percent different bytes between the HTTP captures as compared to the reference capture group. The boxplots for the one-trace test in Figure 52 are nearly identical, and this is confirmed by the p-value of 0.8836 from the Mann-Whitney-Wilcoxon test. The null hypothesis is accepted, and the two groups are considered identical populations. Only bytes for environmental links are changed.

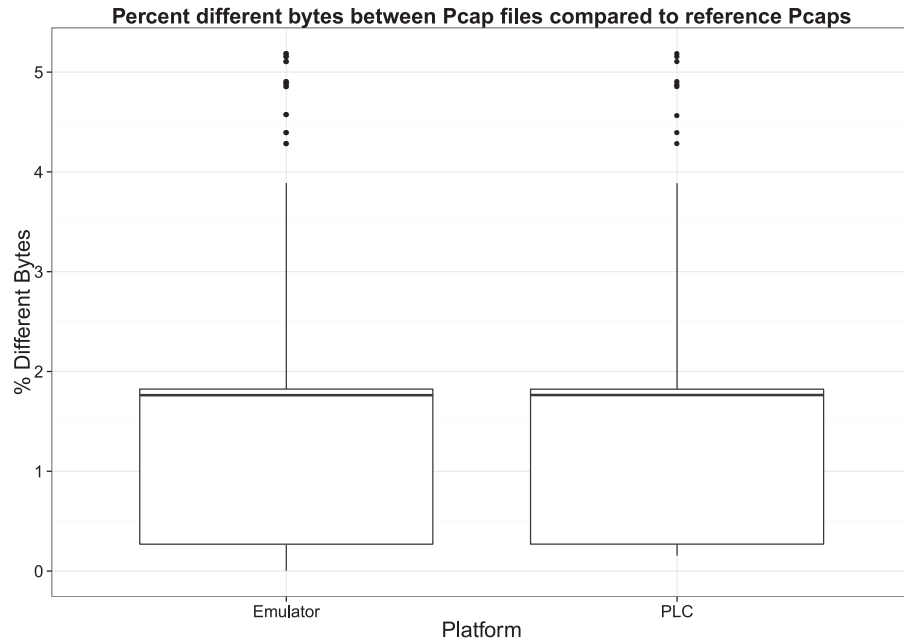


Figure 52. Percent different bytes between Pcap files compared to reference Pcaps - One Pcap per HTTP emulator profile

Table 8. Statistics for percent different bytes between Pcap files compared to reference Pcaps - One Pcap per HTTP emulator profile

Platform	mean	std dev	min	Q1	median	Q3	max
Emulator	1.350	1.053	0.002	0.269	1.762	1.823	5.184
PLC	1.370	1.047	0.154	0.270	1.764	1.823	5.182

Table 9. Statistics for percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per HTTP emulator profile

Platform	mean	std dev	min	Q1	median	Q3	max
Emulator	1.029	0.870	0.040	0.257	0.293	1.706	3.472
PLC	1.138	0.900	0.152	0.264	1.644	1.716	3.457

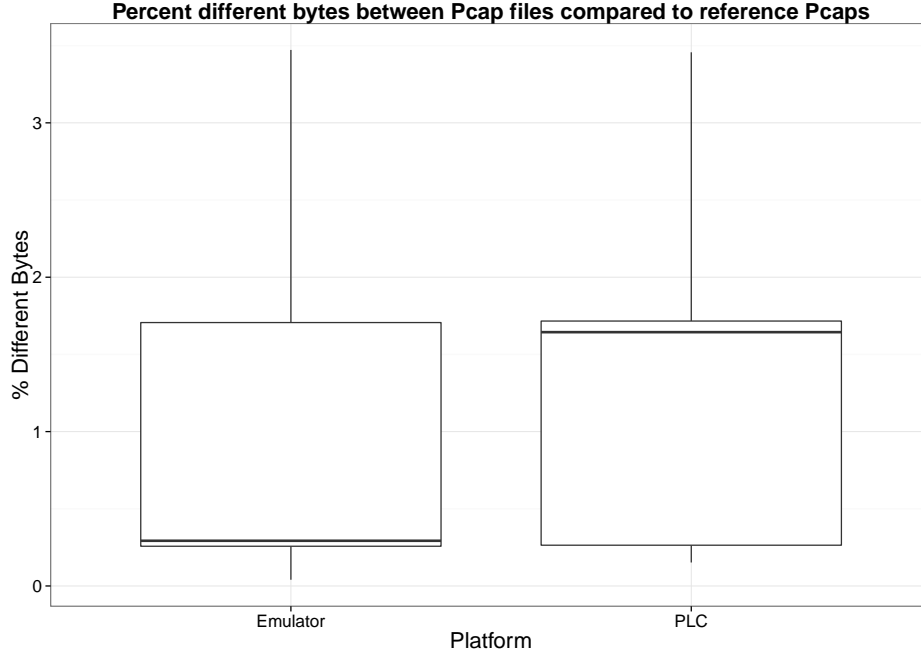


Figure 53. Percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per HTTP emulator profile

However, the emulators built with two traces are not statistically equivalent to the reference group. In fact, as can be seen in Figure 53, the emulators generally generate *less* variability than the PLC. This is due to the fact that the emulators randomly choose between two server messages to respond to every client request. This inherently spreads differences out across the session, since either source trace can be used for a given response. Pilot testing with more input traces confirms that this trend continues with more traces. However, sometimes equivalent server messages between traces lead to a particular response being more heavily weighted.

5.3 Experiment #2 - EtherNet/IP

5.3.1 Build efficiency and qualities.

Tables 10 and 11 summarize how long the P-trees take to build. The time required continues to go up as more traces are added, but it can be expected with

95% confidence that a tree built with six traces or less can be built in less than 3.22 seconds.

Table 10. EtherNet/IP protocol tree build time (sec)

traces	mean	std dev	min	Q1	median	Q3	max
3	2.634	0.144	2.520	2.550	2.565	2.620	2.980
4	2.799	0.138	2.590	2.730	2.785	2.813	3.260
5	2.963	0.175	2.820	2.858	2.890	2.978	3.650
6	3.168	0.141	2.920	3.098	3.145	3.260	3.540

Table 11. EtherNet/IP protocol tree build time confidence intervals

traces	95% conf. interval	
	lower	upper
3	2.583	2.686
4	2.750	2.849
5	2.899	3.026
6	3.117	3.219

Tables 12 and 13 show the throughput achieved while building protocol trees for the RSLinx task. Unlike the HTTP experiment, most of the processing time is spent clustering/aligning edges. Only the first two edges are initially consolidated. These correspond to the initial data handshake and register session commands between RSLinx and the PLC. The final trees consist of one branch since all branches from input traces merge together.

Table 12. EtherNet/IP protocol tree throughput (nodes/sec)

traces	mean	std dev	min	Q1	median	Q3	max
3	39.970	2.032	35.270	40.039	40.925	41.218	41.648
4	50.117	2.311	42.972	49.838	50.321	51.236	54.147
5	59.245	3.142	47.917	58.704	60.556	61.289	61.969
6	66.416	2.912	59.335	64.484	66.837	67.817	71.864

Table 13. EtherNet/IP protocol tree throughput confidence intervals

traces	95% conf. interval	
	lower	upper
3	39.970	39.237
4	49.283	50.950
5	58.112	60.378
6	65.367	67.466

5.3.2 Successful task completions.

Unfortunately, some of the merged edges for the P-trees have erroneous gaps in the aligned sequences. Figure 54 shows a particularly problematic example that is fairly common in P-trees built with two or three input traces. The figure shows that the gaps inserted during alignment result in a fixed region that should not be there. This typically causes an unknown transition to occur during replay unless the client request happens to coincidentally match the erroneous fixed region. The cluster shown in Figure 54 belongs to one of the three-trace emulators, and its test is a PASS in spite of the unknown transitions caused by the gaps. The backtracking algorithm enables the emulator to recover and continue with the rest of the conversation correctly.

The default error message for this experiment, on the other hand, does not appear to help “entice” the client to continue the conversation. Pilot testing shows that repeating the previous message can satisfy what RSLinx/RSLogix is looking for, but only in cases where the sender context (discussed next) is not in effect. Changing the default error messages to use intra-protocol links may resolve this issue.

Overall, the biggest barrier to successful task completion in this experiment is having enough variety in the input traces in order to replay *sender context* correctly. As shown in Figure 55, sender context is an eight byte field within the structure of an EtherNet/IP header (the numbers along the top of the figure correspond to bytes). The values are generated by the client (e.g., RSLinx), and they behave like a counter. Any server response that does not match the client’s sender context is effectively ignored by the client. Because the sender context acts like a counter, some bytes are never varied during a session. As a result, random input traces often incorrectly infer fixed regions that cause issues during replay.

The sampling issue can be overcome by using more input traces. The results show that a minimum of five input traces are required in order to achieve a 100% PASS

rate for this experiment. Other experimental setups may require a different number of traces. It should also be noted that longer traces typically enable easier recovery from unknown transitions since the backtracking algorithm has more transitions to pick from.

Overall, it is observed that the number of required traces to achieve desired variability can be determined by first quantifying the number of bytes that vary for important fields (e.g., sender context) during normal operations. Once this number is known, trees can be iteratively built in successively greater numbers of random input traces until the number of bytes required vary. For instance, in this experiment it is observed that the sender context field generally varies at most by 5 out of 8 bytes for all combinations of reference traces. Thus, any protocol trees that also have mutating regions representing 5 out of 8 sender context bytes generally are successful during replay. Protocol trees with 4 out of 8 bytes varying are sometimes successful, and those with less are rarely successful. This process can be automated by creatively parsing the tree build logs and is left for future work. It should be noted, however, that the proposed process requires knowing which fields are important, which is not protocol-agnostic. The process also assumes a consistent workload.

5.3.3 Byte-level variability.

Next, like the HTTP experiment, environmental links do play a role for this experiment. However, the selected conversation only uses the server IP address in one client message and one server message, so the role is limited.

Figures 56 through 59 show boxplots that compare the overall byte-level variability between the EtherNet/IP emulators and the reference groups. The corresponding table is Table 14. As can be seen in the figures, the variability is very similar, but not the same. It is hypothesized that the current link proposal consolidation algorithm

(shown in Algorithm 4) has a role in the loss of variability. Given two equally weighted unique proposals that have an equal number of links, one is randomly selected (the “first” one). This could potentially result in the same spreading of byte differences as described for HTML in Section 5.2.3. Future work could revise the algorithm to append the additional proposals in a list in order to preserve the information.

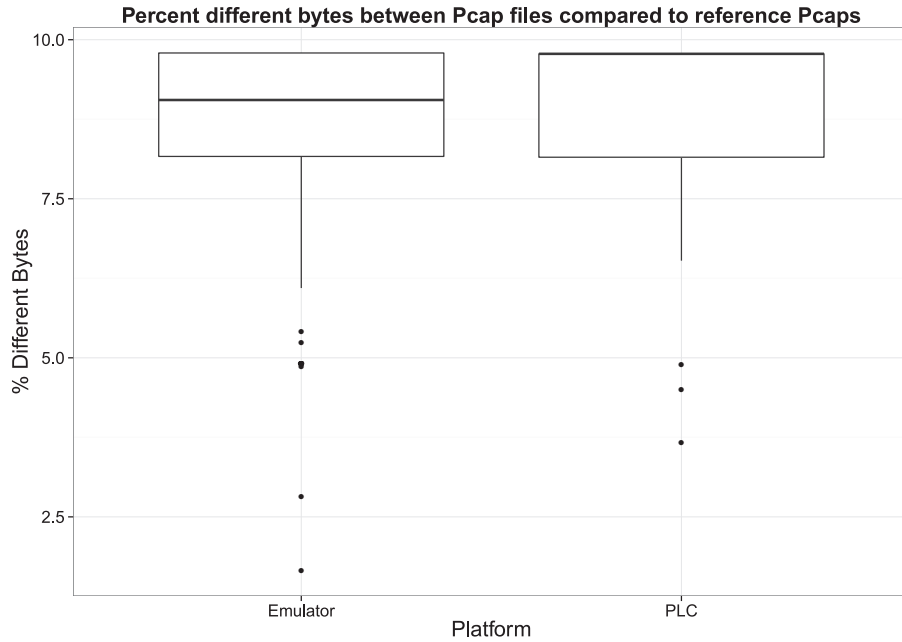


Figure 56. Percent different bytes between Pcap files compared to reference Pcaps - Three Pcaps per EtherNet/IP emulator profile

Next, Table 14 shows that the six-capture tests resulted in higher byte-level variability in both the reference and experimental groups. Viewing the byte differences on the message level reveals that these traces generally had one additional byte that was different than previous tests. The different byte corresponds to the server-set session handle (another counter), which coincidentally happened to count high enough to change a more significant byte.

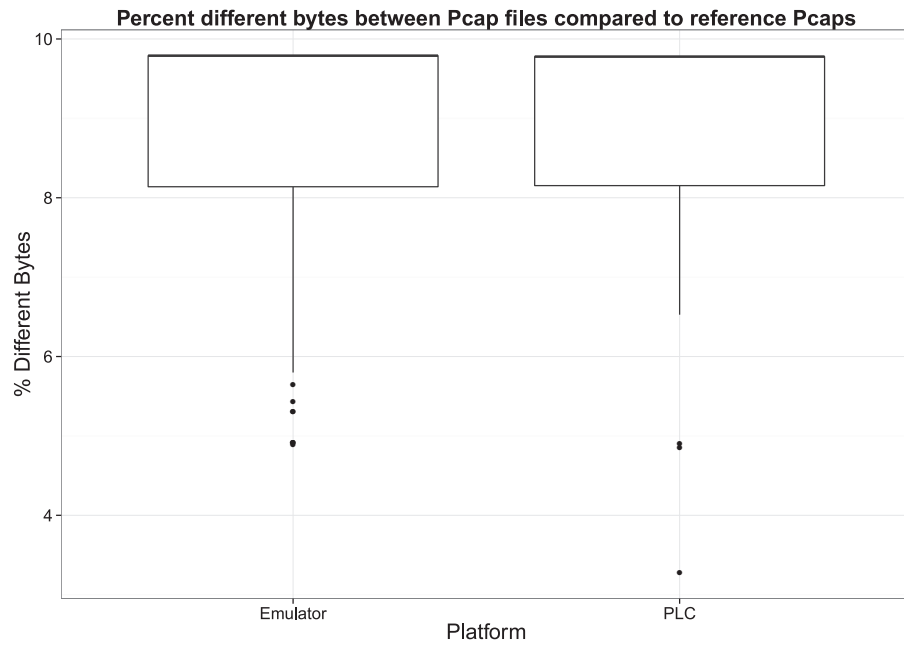


Figure 57. Percent different bytes between Pcap files compared to reference Pcaps - Four Pcaps per EtherNet/IP emulator profile

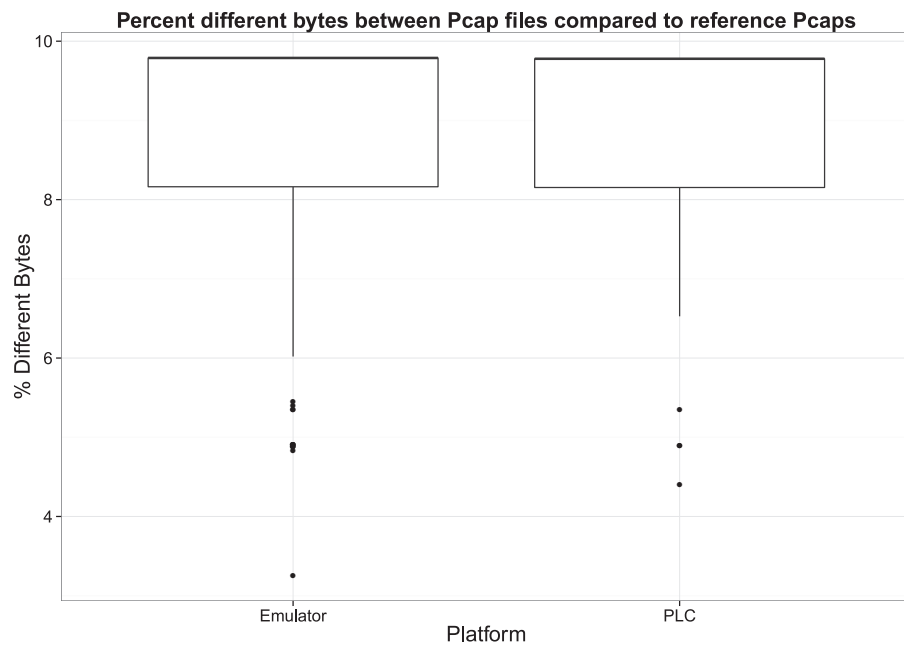


Figure 58. Percent different bytes between Pcap files compared to reference Pcaps - Five Pcaps per EtherNet/IP emulator profile

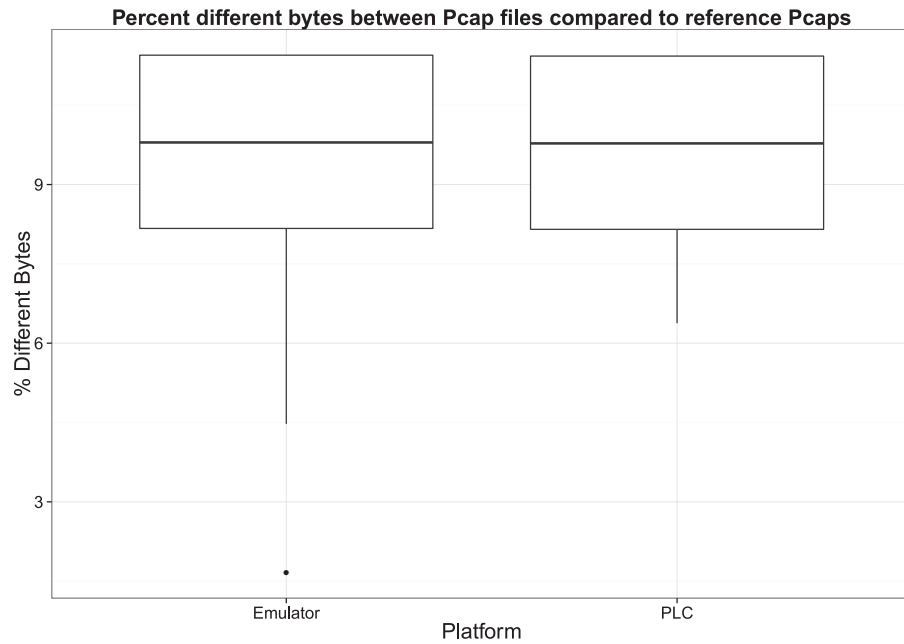


Figure 59. Percent different bytes between Pcap files compared to reference Pcaps - Six Pcaps per EtherNet/IP emulator profile

Table 14. Statistics for percent different bytes between EtherNet/IP emulator Pcap files compared to reference Pcaps

Captures	Platform	mean	std dev	min	Q1	median	Q3	max
3	Emulator	8.628	1.439	1.664	8.166	9.053	9.792	9.792
	PLC	8.808	1.335	3.670	8.153	9.778	9.778	9.778
4	Emulator	8.724	1.350	4.888	8.139	9.790	9.790	9.790
	PLC	8.765	1.410	3.276	8.153	9.778	9.778	9.778
5	Emulator	8.893	1.308	3.260	8.162	9.788	9.788	9.788
	PLC	9.059	1.356	4.409	8.153	9.778	9.778	9.778
6	Emulator	9.671	1.610	1.668	8.170	9.796	11.446	11.446
	PLC	9.803	1.600	6.379	8.153	9.778	11.429	11.429

5.4 Experiment #3 - ISO-TSAP

5.4.1 Build efficiency and qualities.

Tables 15 and 16 summarize how long the P-trees take to build for this experiment. In contrast to the other experiments, the trees all build in less than one second. The reason these trees build much faster than the others is because there is a significantly greater portion of fixed regions in the input traces for this experiment.

Table 15. ISO-TSAP protocol tree build time (sec)

traces	mean	std dev	min	Q1	median	Q3	max
2	0.323	0.007	0.310	0.320	0.320	0.330	0.340
3	0.404	0.007	0.400	0.400	0.400	0.410	0.420
4	0.501	0.007	0.490	0.500	0.500	0.500	0.520
5	0.643	0.014	0.630	0.630	0.640	0.650	0.680

Table 16. ISO-TSAP protocol tree build time confidence intervals

traces	95% conf. interval	
	lower	upper
2	0.321	0.326
3	0.401	0.406
4	0.499	0.504
5	0.638	0.649

Tables 17 and 18 show the build throughput statistics. It is interesting that the throughput actually decreased using 5 input traces versus 4. This could be an anomaly related to the amount of concurrent processing going on within the experimental setup. Additionally, this experiment builds trees with a much higher throughput

than the others. This is partially due to the greater proportion of fixed regions, but it is also due to the fact that approximately half of the client messages are only 7 bytes long.

Table 17. ISO-TSAP protocol tree throughput (nodes/sec)

traces	mean	std dev	min	Q1	median	Q3	max
2	489.587	10.479	465.448	483.977	494.335	496.212	502.447
3	586.523	8.947	562.582	583.831	589.212	591.667	596.609
4	630.640	8.189	605.564	628.993	632.154	635.343	644.167
5	614.605	13.522	580.078	605.919	617.677	625.450	631.110

Table 18. ISO-TSAP protocol tree throughput confidence intervals

traces	95% conf. interval	
	lower	upper
2	485.808	493.365
3	583.298	589.749
4	627.687	633.592
5	609.730	619.480

Finally, the protocol traces observed in this experiment are unique from the other experiments in that they consist of two connections of substantial length (49 and 30 RRs, respectively). Figure 60 show the structure of the initial tree on the left that is generalized into to the smaller final tree on the right. The initial tree was created using two input traces each containing two connections. As can be seen in Figure 60, ScriptGenE’s clustering algorithms yield a final tree with the distinct paths correctly maintained. This is significant since the clustering algorithms are not sufficient to cluster the stateless HTTP traces. The first three RRs in this experiment are

correctly combined as shown at top of final tree. These initial messages appear to establish each connection.

5.4.2 Successful task completions.

As previously shown in Section 5.1, all of the task runs for this experiment completed correctly. However, the SikuliX scripts occasionally falsely reported a FAIL due to the modules being obfuscated by another window. Future work can increase the robustness of the SikuliX scripts.

5.4.3 Byte-level variability.

Table 19 shows that the experimental traces consistently have more variability than the reference traces. In fact, the traces generated by the PLC maintain a fixed amount of variability (0.058%).

Table 19. Statistics for percent different bytes between ISO-TSAP emulator Pcap files compared to reference Pcaps

Captures	Platform	mean	std dev	min	Q1	median	Q3	max
2	Emulator	0.068	0.008	0.029	0.068	0.068	0.077	0.077
	PLC	0.058	0.000	0.058	0.058	0.058	0.058	0.058
3	Emulator	0.063	0.008	0.019	0.058	0.068	0.068	0.077
	PLC	0.058	0.000	0.058	0.058	0.058	0.058	0.058
4	Emulator	0.065	0.009	0.019	0.058	0.068	0.068	0.077
	PLC	0.058	0.000	0.058	0.058	0.058	0.058	0.058
5	Emulator	0.065	0.009	0.019	0.058	0.068	0.068	0.077
	PLC	0.058	0.000	0.058	0.058	0.058	0.058	0.058

The differences in variability are more apparent in Figures 61 and 62. Figure 61 shows the differences in variability when using two input traces, while Figure 62

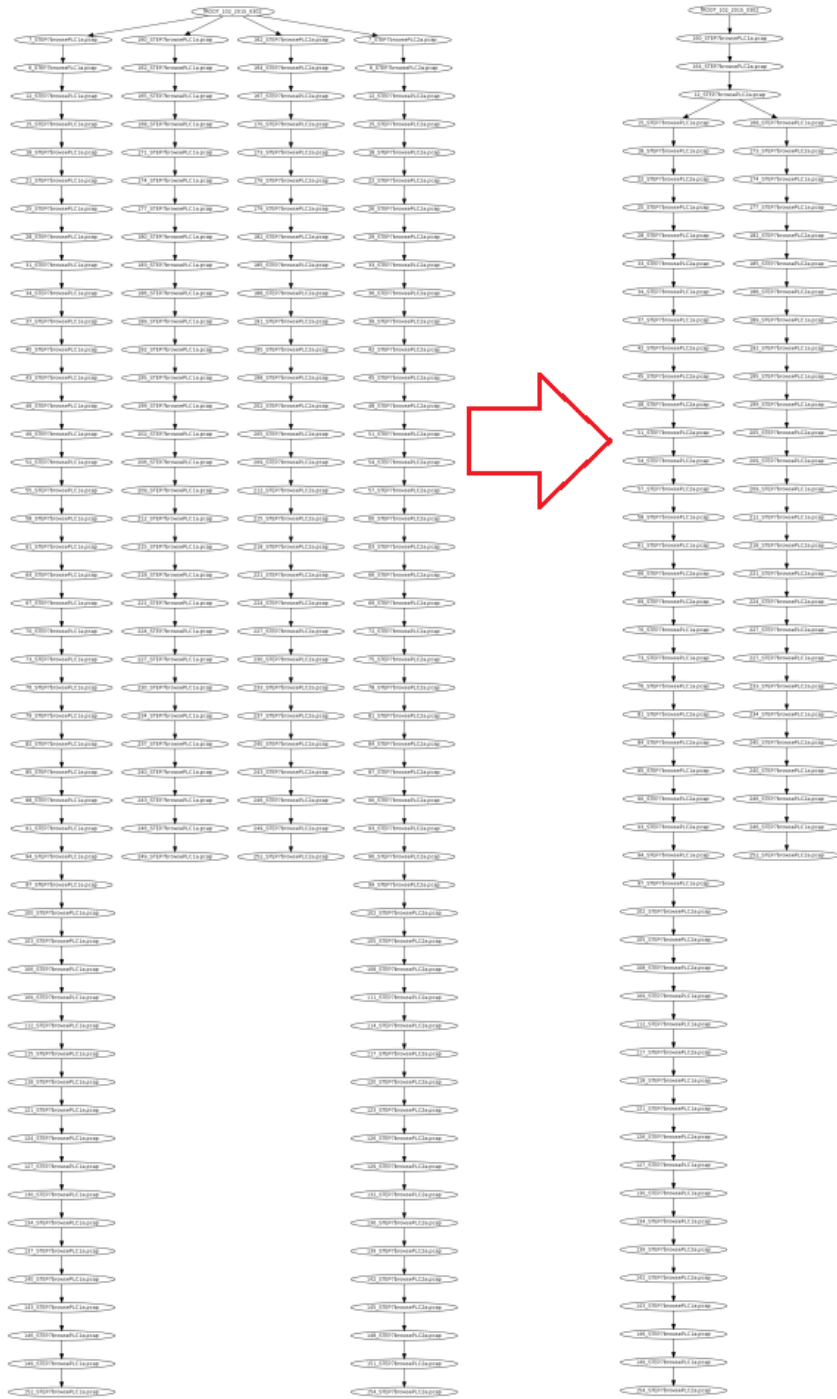


Figure 60. Initial and final trees for STEP7 browse task using two input traces

shows the differences in variability when using three input traces. Charts equivalent to Figure 62 are generated for four and five input traces, respectively, and are not shown. Comparing Figures 61 and 62, it can be seen that the experimental are more similar to the reference traces when using at least three input traces. Thus, even though STEP7's output was the same for all tests, using three input traces may be preferable over using only two input traces. Since many of the messages are fixed and can be trivially matched, it is hypothesized that random server response choices may affect the achieved variability.

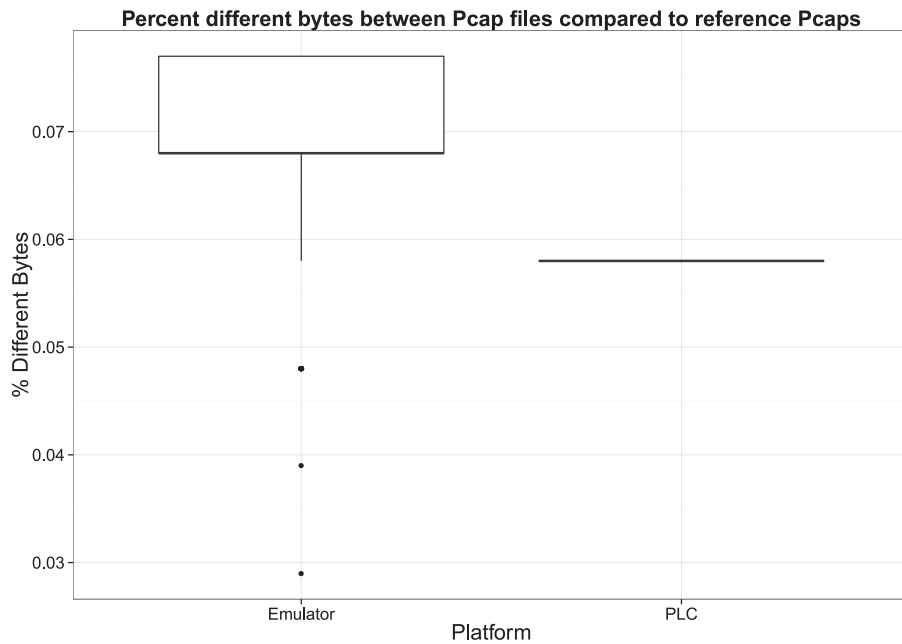


Figure 61. Percent different bytes between Pcap files compared to reference Pcaps - Two Pcaps per ISO-TSAP emulator profile

Additionally, it should be noted that no environmental links are detected, but they could exist. The experimental traces are different from the reference traces by one byte per TCP connection. The significance of these bytes is unknown.

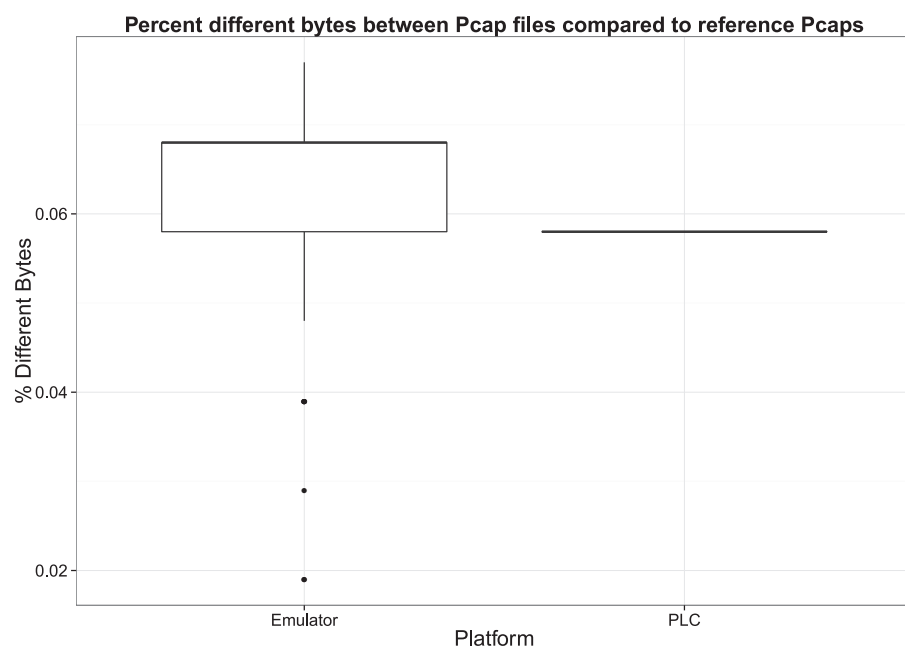


Figure 62. Percent different bytes between Pcap files compared to reference Pcaps - Three Pcaps per ISO-TSAP emulator profile

VI. Conclusions

6.1 Introduction

This final chapter summarizes the research conclusions. Section 6.2 presents the conclusions obtained from experimentation. Section 6.3 discusses the research impacts. Finally, Section 6.4 provides recommendations for future related research.

6.2 Research Conclusions

This research is successful in achieving its goals of developing a proof of concept framework for automatically configuring PLC emulators. The design parameters from Section 3.2 are translated into experimental goals detailed in Section 4.1. Overall, the experiments show that the ScriptGenE Framework satisfies the goals of accuracy, flexibility, and efficiency.

6.2.1 Accuracy.

Results show that with as little as one input trace for HTTP that an Allen-Bradley PLC webserver can be accurately emulated to respond correctly to `wget` requests attempting to mirror the site. Additionally, with as little as five input EtherNet/IP traces, an accurate emulator can be created that is identified by RSLinx as a PLC with modules. For the ISO-TSAP experiment, a minimum of two traces are required to allow STEP7 to browse for an emulator. However, the ISO-TSAP emulators produce potentially one incorrect byte per TCP connection. The other protocol emulators achieve zero incorrect bytes. Overall, the emulators produce traffic that differs in variability from the reference capture group by less than 0.018% with 95% confidence. This equates to less than a one byte difference per EtherNet/IP session on average. It

is hypothesized that the variability can be improved upon by limiting random choices to one source trace profile per session.

6.2.2 Flexibility.

Next, the experiments show that the framework is flexible enough to automatically emulate a variety of protocols without any protocol-specific knowledge. The tested protocols, HTTP, EtherNet/IP, and ISO-TSAP, represent differing text and binary protocols. Additionally, the framework is found to be flexible during replay. By using the novel backtracking algorithm, the replay engine is able to recover from unknown transitions and complete tasks conversations.

6.2.3 Efficiency.

Next, the developed framework builds protocol trees and emulates devices automatically. This is proved by the fully automated experimental methodology. Everything is automated from creating input traces, building protocol trees, replaying traffic, and data collection and aggregation. The input traces required for 100% acceptance rate by standard interrogating tools also meets the requirement of less than seven traces. Finally, the protocol trees are built very fast. Each of the P-trees created in this research's experiments can each be built in less than 6.1 seconds.

6.3 Significance of Research

6.3.1 Contributions.

This research provides numerous contributions. For one, it represents the first successful attempt at automatically developing emulators for EtherNet/IP devices. Unlike similar works designed for IT networks, ScriptGenE requires less traces, can efficiently mimic long polling sessions through looping, and recover from unknown

transitions. Both looping and recovery use the novel backtracking algorithm presented in this research. ScriptGenE also discovers environmental and protocol dependency links, while innovative filtering algorithms inspired by ScriptGen are used to remove trivial links. Additionally, the research methodology presented provides an effective means to evaluate emulator accuracy for unknown protocols. Finally, a thorough literature survey is provided that ties the related fields of honeypots and protocol reverse engineering together. This is done in order to assess the various techniques and technologies that can be used for effective protocol-agnostic emulation.

6.3.2 Applications.

The applications of the above contributions are numerous. In addition to providing stand alone honeypots, ScriptGenE can be used to enhance hybrid network sensors in order to provide enhanced intrusion detection capabilities, capture and contain malware, and develop auto-learning firewalls or intrusion prevention sensors. Additionally, the tools can be used for client and server fuzzing. Finally, ScriptGenE provides tools that are useful for network security training and future research.

6.4 Future Work

6.4.1 Overview of recommendations.

Recommendations for future work fall into three main categories. The first category relates to additional testing and test enhancements. The second category is for enhancing current algorithms for improved clustering, linking, and replay behavior. The final category discussed is for integrating ScriptGenE into hybrid network sensor applications.

6.4.2 Testing.

Future research should consider performing additional testing on the framework. First, there are many more EtherNet/IP tasks that can be tested such as program upload, program download, go online, and firmware update. SikuliX scripts can be employed again to develop automated tests. Additionally, more PLC configurations should be considered in testing. These can include devices from other vendors that use different protocols.

Testing may also be enhanced by using tcpdump instead of tshark for capturing. Using different machine configurations may also prevent loss of packets. Additionally, ScriptGenE's ability to detect missing packets can be enhanced. Finally, as discussed in Section 5.3.2, enhanced tree quality indicators may be able to be determined based upon observed number of mutating regions versus the variability observed in reference traces.

6.4.3 Enhancing current ScriptGenE algorithms.

Next, the current implementation of ScriptGenE can be improved upon in many ways. The primary areas for improvement are clustering, linking, and replay behavior.

6.4.3.1 Improving clustering.

One technique that has been proven to significantly improve clustering is micro-clustering based upon region-wide mutation rates [LMD05]. Additionally, testing should be done to determine the maximum message and/or cluster size that is still useful to Macrocluster server messages. Once this is determined, Macroclustering server messages can be automatically turned off when the threshold is exceeded. Finally, trivial links may be able to be used for discovering delimiters. For example, pilot testing shows that typical trivial links for HTTP are '\r\n', which also hap-

pen to denote the end of a line. The delimiters can then be used for enhanced field identification. Thus, trivial links may have a legitimate use.

6.4.3.2 Improving linking.

Several improvements can be made to the link algorithms. Intra-protocol links, for instance, currently only recognize relationships between adjacent messages. However, often the relationships span through the majority of messages throughout a session. These *global links* could be identified and used to enhance the current link identification schemes. This is particularly important for branches that diverge on their own since unclustered edges do not obtain links. Thus, global links can infer links in other messages as needed. Determining different kinds of byte relationships (such as a counter) can also be explored. Finally, environmental links can be improved upon by adding additional link types (e.g., length fields), expanding the representations allowed for the current types, and auto-detecting host names from DNS packets. Filtering trivial environmental links of less than four bytes may also be worth considering. “80”, for example, would make a rather short port link and may coincidentally appear in data.

6.4.3.3 Improving replay.

Additionally, replay can be enhanced in many ways. For instance, it is often observed that clients will continue to send messages even after an erroneous server response is sent. Thus, after sending a default error message, the next needed response is in the next state rather than past states. As a result, it is proposed that a *look-ahead* feature be added to the backtracking algorithm for increased efficiency. Additionally, new options could limit how many nodes will be backtracked, or an option could disable the BFS feature.

As discussed previously, replay can also be improved by limiting random response choices. Once the first random selection is made, future random response considerations can attempt to use the same source trace. The same feature can also apply to multiple link proposals.

A third area of improving replay is to research the usefulness of using links in default error messages. New heuristics may also improve their utility.

Finally, replay can be greatly enhanced by allowing more than one concurrent session. This would open more possibilities for testing as well.

6.4.4 Hybrid network sensor integration.

Allowing for more than one connection is also a prerequisite for discovering and handling inter-protocol dependencies [LDM06]. This and other improvements would allow for tight integration with hybrid network sensors.

6.4.4.1 Application layer integration.

One such needed improvement is the ability for ScriptGenE to be incrementally updated (i.e., update a current tree using new input traces). Previous research has used this to integrate honeypots with proxies such as those described in Section 2.3.5. ScriptGenE also includes initial capabilities for using missed handshake sessions that can be fully implemented. The primary idea is to take each branch and match it up with sequence of messages along the tree. An algorithm similar to identifying links in data chunks (Section 3.6.2.6) can be employed at the message level to accomplish this. Incremental updates can then be done to the tree to merge the new data.

6.4.4.2 Transport layer integration.

In addition to the application layer improvements described previously, ScriptGenE could also be integrated with additional software for transport layer emulation such as those described in Section 2.3.2. Honeyd is a popular option, and it has been shown that automatically configuring it using network sniffers is effective [VM14]. Collaboration with the Idaho National Labs research team may prove highly beneficial for integrating ScriptGenE’s flexible application layer replay with their hybrid IDS sensor network [VML14].

6.4.5 Miscellaneous.

Finally, there are a few miscellaneous improvements that can be made such as handling IPv6 and UDP traffic. Timing issues may also be considered. Finally, a tree visualization and manipulation tool may be desirable (particularly for demonstration purposes). The Python Environment for Tree Exploration may be a good library to consider.

6.5 Chapter Summary

In summary, this final chapter presents the research conclusions and impacts. Additionally, many ideas for future work were discussed that, along with previous research, build upon the foundational ScriptGenE Framework presented in this research.

Bibliography

- [ANV11] João Antunes, Nuno Neves, and Paulo Verissimo. Reverse engineering of protocols from network traces. In *Reverse Engineering, 18th Working Conference on*, WCRE'11, pages 169–178, October 2011.
- [Bed04a] Marshall Beddoe. Network protocol analysis using bioinformatics algorithms. Technical report, 2004. Retrieved 7 October, 2014 from <http://www.4tphi.net/~awalters/PI/pi.pdf>.
- [Bed04b] Marshall Beddoe. The protocol informatics project, 2004. Retrieved 7 October, 2014 from <http://www.4tphi.net/~awalters/PI/PI.html>.
- [Ber06] Pavel Berkhin. A survey of clustering data mining techniques. In *Grouping Multidimensional Data*, pages 25–71. Springer, 2006.
- [Ber13] Robin G. Berthier. Honeybrid: Hybrid honeypot framework, 2013. Retrieved 23 September, 2014 from <http://honeybrid.sourceforge.net>.
- [Ber35] Robin G. Berthier. *Advanced Honeypot Architecture for Network Threats Quantification*. PhD thesis, University of Maryland, College Park, MD, USA, 2009 (304924035). Retrieved 23 September, 2014 from http://drum.lib.umd.edu/bitstream/1903/9204/1/Berthier_umd_0117E_10310.pdf.
- [BGH13] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Netzob, 2013. Retrieved 27 October, 2014 from <http://www.netzob.org>.
- [BGH14] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Information, Computer, and Communications Security, Proceedings of the 9th ACM Symposium on*, pages 51–62. ACM, 2014.
- [BJM⁺14] 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 October, 2014 from <http://crysys.hu/publications/files/BuzaJMFH2014smartgridsec.pdf>.
- [BK13] Paul Baecher and Markus Koetter. Dionaea, 2013. Retrieved 9 March, 2015 from <http://dionaea.carnivore.it>.
- [BKH⁺06] 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.

- [Bod19] Roland C. Bodenheimer. 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).
- [Bro01] 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.
- [CCG⁺11] 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.
- [CKW07] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 199–212, 2007. Retrieved 16 October, 2014 from http://static.usenix.org/event/sec07/tech/full_papers/cui/cui.pdf.
- [CPC⁺08] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Computer and Communications Security, Proceedings of the 15th ACM Conference on*, CCS’08, pages 391–402, New York, NY, USA, 2008. ACM.
- [CPKS09] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Computer and Communications Security, Proceedings of the 16th ACM Conference on*, CCS’09, pages 621–634, New York, NY, USA, 2009. ACM.
- [CPW06] 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 21 October, 2014 from <http://www.icir.org/vern/papers/gq-techreport.pdf>.
- [CPWK06] 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 18 September, 2014 from <http://www.internetsociety.org/doc/protocol-independent-adaptive-replay-application-dialog>.
- [CS13] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451–474, 2013.

- [CTC04] 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 26 October, 2014 from <http://seclab.cs.sunysb.edu/alok/papers/icdcit04.pdf>.
- [CWKK09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *Security and Privacy, 30th IEEE Symposium on*, pages 110–125. IEEE, 2009.
- [CYLS07] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Computer and Communications Security, Proceedings of the 14th ACM Conference on*, CCS'07, pages 317–329, New York, NY, USA, 2007. ACM.
- [DBDS05] 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.
- [Dun13] Stephen J. Dunlap. Timing-based side channel analysis for anomaly detection in the industrial control system environment. Master's thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, June 2013.
- [Edw13] Richard Edwards. UPGMA worked example, 2013. Retrieved 5 February, 2015 from <http://www.southampton.ac.uk/~re1u06/teaching/upgma>.
- [Fal10] Nicolas Falliere. Exploring Stuxnet's PLC infection process. Technical report, Symantec Official Blog, 2010. Retrieved 8 February, 2015 from <http://www.symantec.com/connect/blogs/exploring-stuxnet-s-plc-infection-process>.
- [FB13] 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 16 October, 2014 from http://www.thinkmind.org/download.php?articleid=sec_v6_n12_2013_3.
- [FGM⁺99] 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 4 February, 2015 from <http://tools.ietf.org/html/rfc2616>.

- [Fin90] 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 (ADA602990).
- [GLB12] 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.
- [HCA⁺09] 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.
- [HH13] 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.
- [Hie04] 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/utis/getfile/collection/etd/id/516/.../517.pdf>.
- [Hig13] Kelly J. Higgins. ‘Project SHINE illuminates sad state of SCADA/ICS security on the net. Dark Reading blog entry, 2013. Retrieved 8 February, 2015 from <http://www.darkreading.com/vulnerabilities---threats/project-shine-illuminates-sad-state-of-scada-ics-security-on-the-net/d/d-id/1140691>.
- [HNH06] 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.
- [Hoc15] Raimund Hocke. SikuliX powered by RaiMan, 2015. Retrieved 9 March, 2015 from <http://www.sikulix.com>.
- [hon09] Honeywall, 2009. Retrieved 2 October, 2014 from <https://projects.honeynet.org/honeywall>.
- [HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Python in Science Conference, Proceedings of the 7th, SciPy’08*, pages 11–15, Pasadena, CA USA, August 2008.

- [Jar82] 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 (ADA582482).
- [JX04] Xuxian Jiang and Dongyan Xu. BAIT-TRAP: a catering honeypot framework. Technical report, Purdue University, 2004. Retrieved 26 October, 2014 from <http://friends.cs.purdue.edu/pubs/BaitTrap.pdf>.
- [JXW06] 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.
- [KGKR12] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. Learning stateful models for network honeypots. In *Security and Artificial Intelligence, Proceedings of the 5th ACM Workshop on*, pages 37–48. ACM, 2012.
- [KKR11] Tammo Krueger, Nicole Krämer, and Konrad Rieck. ASAP: Automatic Semantics-aware Analysis of Network Payloads. In *Privacy and Security Issues in Data Mining and Machine Learning, Proceedings of the International ECML/PKDD Conference on*, PSDML'10, pages 50–63, Berlin, Heidelberg, 2011. Springer-Verlag.
- [Kon14] Ioannis Koniaris. Honeydrive 3 honeypot bundle distro. BruteForce Lab's Blog, 2014. Retrieved 14 December, 2014 from <http://bruteforce.gr/honeydrive>.
- [KOY⁺12] 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*, Cyber-Security'12, pages 165–172. IEEE, December 2012.
- [KR13] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison-Wesley Publishing Company, Boston, MA, USA, 6th edition, 2013.
- [KSAMA04] Iyad Kuwatly, Malek Sraï, 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.
- [KT06] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley Publishing Company, Boston, MA, USA, 2006.

- [KWK⁺11] 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.
- [LBZH13] Patrick LaRoche, Aimee Burrows, and A. Nur Zincir-Heywood. How far an evolutionary approach can go for protocol state analysis and discovery. In *Evolutionary Computation, IEEE Congress on, CEC'13*, pages 3228–3235. IEEE, 2013.
- [LC11] 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.
- [LD08] 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.
- [LDM06] Corrado Leita, Marc Dacier, and Frederic Massicotte. Automatic handling of protocol dependencies and reaction to 0-day attacks with Script-Gen 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.
- [Lei12] Corrado Leita. Challenges in critical infrastructure security. Cybersecurity in CPS workshop. Technical report, Symantec, 2012. Retrieved 20 October, 2014 from http://csrc.nist.gov/news_events/cps-workshop/slides/presentation-6_leita-dacier.pdf.
- [Lev11] 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>.
- [LJXZ08] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed System Security Symposium*, volume 8, pages 1–15, February 2008. Retrieved 23 October, 2014 from <http://www.internetsociety.org/doc/automatic-protocol-format-reverse-engineering-through-context-aware-monitored-execution>.

- [LMD05] 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.
- [MB09] 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.
- [McM00] Lucille R. McMinn. External verification of SCADA system embedded controller firmware. Master’s thesis, Air Force Institute of Technology, Wright-Patterson AFB OH, USA, March 2012 (ADA557800).
- [NBFS06] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Re-player: Automatic protocol replay by binary analysis. In *Computer and Communications Security, Proceedings of the 13th ACM Conference on*, CCS ’06, pages 311–321, New York, NY, USA, 2006. ACM.
- [NTT83] Masatoshi Nei, Fumio Tajima, and Yoshio Tatenno. Accuracy of estimated phylogenetic trees from molecular data. *Journal of Molecular Evolution*, 19(2):153–170, 1983.
- [NW70] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.
- [PH07] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Pearson Education, Boston, MA, USA, first edition, 2007.
- [PHDW12] Fan Pan, Zheng Hong, Youxiang Du, and Lifa Wu. Efficient protocol reverse method based on network trace analysis. *International Journal of Digital Content Technology and its Applications*, 6(20):201–210, 2012. Retrieved 16 October, 2014 from <http://www.aicit.org/JDCTA/ppl/JDCTA1921PPL.pdf>.
- [Pro13] Niels Provos. Honeyd (version 1.6d), 2013. Retrieved 13 December, 2014 from <https://github.com/DataSoft/Honeyd>.
- [R14] R: The R Project for statistical computing, 2014. Retrieved 9 March, 2015 from <http://www.r-project.org>.
- [Rab89] Lawrence Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

- [RCHJ14] 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.
- [RGM09] 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.
- [Rou87] Peter J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.
- [San12] Ruben Santamarta. Project Basecamp – attacking ControlLogix. In *5th SCADA Security Scientific Symposium*, Miami Beach, FL, USA, January 2012. Digital Bond. Retrieved 10 January, 2015 from http://reversemode.com/downloads/logix_report_basecamp.pdf.
- [SD05] Suvrit Sra and Inderjit S. Dhillon. Generalized nonnegative matrix approximations with Bregman divergences. In *Advances in Neural Information Processing Systems 18*, NIPS'05, pages 283–290. Vancouver, Canada, December 2005. Retrieved 7 February, 2015 from <http://papers.nips.cc/paper/2757-generalized-nonnegative-matrix-approximations-with-bregman-divergences.pdf>.
- [SFS11] 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 7 February, 2015 from <http://csrc.nist.gov/publications/nistpubs/800-82/SP800-82-final.pdf>.
- [SHZ10] Chengyu Song, Brian Hay, and Jianwei Zhuge. Know your tools: Qebek – conceal the monitoring. Technical report, The Honeynet Project, October 2010. Retrieved 5 February, 2015 from http://www.honeynet.org/sites/default/files/files/KYT-Qebek-final_v1.pdf.
- [SMM⁺08] Sam Small, Joshua Mason, Fabian Monroe, Niels Provos, and Adam Stubblefield. To catch a predator: A natural language approach for eliciting malicious payloads. In *USENIX Security Symposium*, pages 171–184, 2008. Retrieved 20 October, 2014 from http://static.usenix.org/events/sec08/tech/full_papers/small/small.pdf.

- [STO⁺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.
- [SW81] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
- [TPB99] Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method. *Communication, Control, and Computing, 37th Annual Allerton Conference on*, 1999. Retrieved 7 February, 2015 from <http://www.cs.huji.ac.il/labs/learning/Papers/allerton.pdf>.
- [vdHvW05] Jeffrey van der Hoeven and Hilde van Wijngaarden. Modular emulation as a long-term preservation strategy for digital objects. In *5th International Web Archiving Workshop, IWAW'05*, Vienna, Austria, 2005. Retrieved 2 February, 2015 from <http://iwaw.europarchive.org/05/papers/iwaw05-hoeven.pdf>.
- [Ves15] Johnny Vestergaard. Beeswarm, 2015. Retrieved 5 February, 2015 from <http://www.beeswarm-ids.org>.
- [VM14] 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.
- [VMC⁺05] Michael Vrabie, 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.
- [VML14] 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.
- [WBC10] Sean Whalen, Matt Bishop, and James P. Crutchfield. Hidden Markov Models for automated protocol learning. In *Security and Privacy in Communication Networks*, SecureComm'10, pages 415–428. Springer Berlin Heidelberg, 2010. Retrieved 16 October, 2014 from <http://nob.cs.ucdavis.edu/bishop/papers/2010-securecomm/markov.pdf>.

- [WCKK08] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In *Network and Distributed System Security Symposium*, volume 8, February 2008. Retrieved 2 February, 2015 from <http://www.internetsociety.org/doc/automatic-network-protocol-analysis-paper>.
- [Whi13] White House Office of the Press Secretary. Presidential Policy Directive 21. *Critical Infrastructure Security and Resilience*, 2013. Retrieved 8 February, 2015 from <http://www.whitehouse.gov/the-press-office/2013/02/12/presidential-policy-directive-critical-infrastructure-security-and-resil>.
- [Wil13a] Kyle Wilhoit. The SCADA that didn't cry wolf. Technical report, Trend Micro, Inc., 2013. Retrieved 7 February, 2015 from <http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-the-scada-that-didnt-cry-wolf.pdf>.
- [Wil13b] Kyle Wilhoit. Who's really attacking your ICS equipment? Technical report, Trend Micro, Inc., 2013. Retrieved 7 February, 2015 from <http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-whos-really-attacking-your-ics-equipment.pdf>.
- [Wil89] 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).
- [WJ94] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.
- [WJC⁺09] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *Research in Computer Security, Proceedings of the 14th European Symposium on, ESORICS'09*, pages 200–215, Berlin, Heidelberg, 2009. Springer-Verlag.
- [WSED11] Gérard Wagener, Radu State, Thomas Engel, and Alexandre Dulaunoy. Adaptive and self-configurable honeypots. In *Integrated Network Management, 2011 IFIP/IEEE International Symposium on*, pages 345–352. IEEE, May 2011.
- [WYS⁺12] Yipeng Wang, Xiaochun Yun, Muhammad Zubair Shafiq, Liyan Wang, Alex X Liu, Zhibin Zhang, Danfeng Yao, Yongzheng Zhang, and Li Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Network Protocols, 20th IEEE International Conference on, ICNP'12*, pages 1–10. IEEE, October 2012.

- [WZY⁺11] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Applied Cryptography and Network Security, Proceedings of the 9th International Conference on*, ACNS'11, pages 1–18, Berlin, Heidelberg, 2011. Springer-Verlag.
- [YCM09] 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

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

1. REPORT DATE (DD-MM-YYYY) 26-03-2015			2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Sept 2013 — Mar 2015	
4. TITLE AND SUBTITLE Automatic Configuration of Programmable Logic Controller Emulators					5a. CONTRACT NUMBER	
					5b. GRANT NUMBER	
					5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Warner, Phillip, C., Capt, USAF					5d. PROJECT NUMBER 15G216C	
					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-15-M-024	
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 presents a scalable solution to automatically configure programmable logic controller emulators using network traces. The accuracy, flexibility, and efficiency of the proposed framework, ScriptGenE, is tested in three fully automated experiments. Results from the experiments show that ScriptGenE can accurately emulate a PLC's webserver with only one input trace. Additionally, only five input EtherNet/IP traces are required to create an emulator that is identified by RSLinx as a PLC with modules. A minimum of two input traces are required to create a Siemens PLC emulator that can be browsed by STEP7. Additionally, the emulators produce traffic that differs in variability from the reference capture group by less than 0.018% with 95% confidence. Overall, this research provides numerous contributions including the first successful automatically configured application layer honeypot for EtherNet/IP. ScriptGenE requires less input traces than previous works. Additionally, a novel backtracking algorithm is implemented that handles unknown transitions and allows for looping in ICS polling sessions.						
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	199	19b. TELEPHONE NUMBER (include area code) (937) 255-3636 x7979 Barry.Mullins@afit.edu	