EXAMINING EFFECTIVENESS OF WEB-BASED INTERNET OF THINGS HONEYPOTS

THESIS

Lukas A. Stafira, 2d Lt, USAF

AFIT-ENG-MS-19-M-057

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-19-M-057

EXAMINING EFFECTIVENESS OF WEB-BASED INTERNET OF THINGS HONEYPOTS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Lukas A. Stafira, B.S.C.S.

2d Lt, USAF

March 2019

AFIT-ENG-MS-19-M-057

EXAMINING EFFECTIVENESS OF WEB-BASED INTERNET OF THINGS HONEYPOTS

THESIS

Lukas A. Stafira, B.S.C.S.
2d Lt, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
(Chairman)

Timothy H. Lacey, Ph.D., CISSP
(Member)

Stephen Dunlap, M.S., CISSP
(Member)

AFIT-ENG-MS-19-M-057

Abstract

The Internet of Things (IoT) is growing at an alarming rate. It is estimated that there will be over 25 billion IoT devices by 2020. The simplicity of their function usually means that IoT devices have low processing power, which prevent them from having intricate security features, leading to vulnerabilities. This makes IoT devices the prime target of attackers in the coming years. Honeypots are intentionally vulnerable machines that run programs which appear as a vulnerable device to a would-be attacker. They are placed on a network to entice and trap an attacker and then gather information on them, including place of origin and method of attack. Due to their prevalence and propensity for having vulnerabilities, IoT devices are a perfect candidate for honeypots placed on a network.

Honeyd is popular open-source software written by Niels Provos that creates low-interaction virtual honeypots. It is able to simulate everything at the network level, allow the user to create various Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) services, and allow Operating System (OS) simulation for scanning tools such as Nmap. This research seeks to determine if Honeyd is capable of producing convincing IoT honeypots.

Three IoT devices: a TITAThink camera, a Proliphix thermostat, and an ezOutlet2 power outlet, had their Hypertext-Transfer Protocol (HTTP) services simulated through Python scripts and integrated with Honeyd to create three IoT honeypots. These honeypots were then compared to the actual devices to determine how similar they were. The devices and honeypots are both queried in the exact same manner and have their response times, code, headers, and Nmap scan results compared to see how they differ.

Experimental results show there is a statistically-significant difference between the means for query response times and Nmap scan times; however, the code and headers were over 90% similar in 18/27 tests. The differences that were recorded were mostly due to limitations in the physical IoT devices. The Nmap scan results were successful at simulating the service and manufacturer scans, but Nmap was able to identify a difference in fingerprinted operating systems.

This thesis showcases how Honeyd is a useful program for creating IoT honeypots. The code in this research could be used to quickly deploy authentic IoT honeypots or it could be adapted to create different types of IoT honeypots. In addition, it could easily be adapted to create honeypots of other IoT devices that utilize HTTP. The ability to easily create these IoT honeypots would be useful to the defense department and members of the security industry interested in integrating IoT honeypots in their networks.

To Mom and Dad,

Thanks for your love and support.

Acknowledgements

It's a job that's never started that takes the longest to finish -J.R.R Tolkien

I want to thank Dr. Barry Mullins, my advisor, for his consistent advice and support throughout my time at AFIT and during my thesis research.

I would also like to thank everyone in the lab, for making those hours spent there entertaining, enjoyable, insightful, and at times thought-provoking.

Lukas A. Stafira

Table of Contents

Page

Abstract iv

Dedication vi

Acknowledgements vii

List of Figures xi

List of Tables xviii

I. Introduction 1

1.1 Background 1
1.2 Motivation 2
1.3 Research Goals 3
1.4 Approach 3
1.4.1 IoT Network 4
1.4.2 Honeyd Implementation 4
1.4.3 Comparison 5
1.4.4 Experimentation 5
1.5 Assumptions and Limitations 7
1.5.1 Assumptions 7
1.5.2 Limitations 7
1.6 Research Contributions 8
1.7 Thesis Overview 9

II. Background and Related Research 10

2.1 Overview 10
2.2 Background 10
2.2.1 Internet of Things (IoT) 10
2.2.2 Honeypots 19
2.2.3 Nmap 22
2.2.4 Honeyd 23
2.3 Related Research 27
2.3.1 IoT Honeypots 27
2.3.2 Honeyd 29
2.3.3 Uses for Honeypots 30
2.4 Chapter Summary 31

Page

III. Framework Design . . . 32

3.1 Overview . . . 32
3.2 Motivation and Application . . . 32
3.3 IoT System Under Test . . . 34
3.3.1 TITAThink Camera . . . 34
3.3.2 Proliphix Thermostat . . . 39
3.3.3 ezOutlet2 Power Switch . . . 43
3.4 Honeyd IoT System Framework . . . 45
3.4.1 Honeyd Configuration . . . 46
3.4.2 Web Server . . . 48
3.4.3 Proliphix Thermostat Honeypot . . . 55
3.4.4 ezOutlet2 Honeypot . . . 59

IV. Research Methodology . . . 64

4.1 Goals . . . 64
4.2 Approach . . . 64
4.2.1 Packet Timing and Content . . . 64
4.2.2 Nmap Scans . . . 68
4.3 System Boundaries . . . 69
4.4 Parameters and Factors . . . 70
4.4.1 Assumptions . . . 70
4.4.2 System Parameters . . . 71
4.4.3 Factors . . . 72
4.4.4 Metrics . . . 73
4.5 Methodology . . . 74
4.6 Apparatus . . . 76
4.7 Results . . . 77
4.8 Chapter Summary . . . 83

V. Results and Analysis . . . 84

5.1 Overview . . . 84
5.2 Metric 1 - Query Completion Time . . . 84
5.2.1 TITAThink Camera . . . 84
5.2.2 Proliphix Thermostat . . . 86
5.2.3 ezOutlet2 Power Outlet . . . 90
5.3 Metric 2 - Data Difference . . . 93
5.3.1 TITAThink Camera . . . 93
5.3.2 Proliphix Thermostat . . . 95
5.3.3 ezOutlet Power Outlet . . . 97
5.4 Metric 3 - Number of Packets . . . 99
5.4.1 TITAThink Camera . . . 99
5.4.2 Proliphix Thermostat . . . 100

Page

5.4.3 ezOutlet Power Outlet . . . 101
5.5 Metric 4 - Header Difference . . . 103
5.5.1 TITAThink Camera . . . 103
5.5.2 Proliphix Thermostat . . . 104
5.5.3 ezOutlet Power Outlet . . . 106
5.6 Metric 5 - Nmap Scan Time . . . 108
5.6.1 TITAThink Camera . . . 108
5.6.2 Proliphix Thermostat . . . 110
5.6.3 ezOutlet Power Outlet . . . 112
5.7 Metric 6 - Nmap Scan Difference . . . 115
5.7.1 TITAThink Camera . . . 115
5.7.2 Proliphix Thermostat . . . 116
5.7.3 ezOutlet Power Outlet . . . 116
5.8 Summary . . . 117

VI. Conclusions . . . 118

6.1 Introduction . . . 118
6.2 Research Conclusions . . . 118
6.2.1 Response Time . . . 118
6.2.2 Data Similarity . . . 119
6.2.3 Nmap Scans . . . 121
6.3 Research Significance . . . 122
6.4 Research Limitations . . . 122
6.5 Scalability . . . 124
6.6 Future Work . . . 124
6.7 Chapter Summary . . . 125

Appendix A. Honeyd Configuration Code . . . 126

Appendix B. TITAThink Camera Honeypot Code . . . 128

Appendix C. Proliphix Thermostat Honeypot Code . . . 137

Appendix D. ezOutlet2 Power Outlet Code . . . 144

Appendix E. Testing and Comparison Scripts . . . 150

Appendix F. Experimentation Data . . . 172

Appendix G. Statistical Tests . . . 179

Bibliography . . . 198

List of Figures

Figure Page

1. IoT architecture [1] 11
2. MAC headers for IEEE 802.11 and IEEE 802.11ah [2] 13
3. Example ZigBee network topology adapted from [3] 15
4. Honeypot deployed as an IDS adapted from [4] 22
5. Example Honeyd configuration file [5] 25
6. Example of an Nmap signature database file. This signature is for Windows XP SP3 or SP4 [6] 26
7. `test.sh` script packaged with Honeyd [7] 27
8. Model of experiment network configuration 35
9. Network of the physical IoT devices being tested 36
10. TITAThink TT520PW camera 36
11. TITAThink camera login page 37
12. TITAThink camera main page 38
13. Proliphix NT130h thermostat 40
14. Wiring of power and Ethernet for Proliphix thermostat 40
15. Wiring diagram of Proliphix thermostat 41
16. Proliphix thermostat main page 42
17. ezOutlet2 EZ-22b power outlet 44
18. ezOutlet2 EZ-22b Ethernet port 44
19. ezOutlet2 EZ-22b power outlet main page 45
20. Code excerpt from `iotHoneyd.conf` (Appendix A) showing how each device has a script run on TCP port 80 47
21. Example HTTP GET request [8] 49

Figure Page

22. The Python script has the capability to draw date and time on the camera image . . . 50

23. Wireshark capture of the TITAThink camera showing the redirects that happen when trying to access the main page . . . 51

24. Conditional statements showing how headers and files are sent based on the file requested . . . 52

25. Homepage of Honeyd camera honeypot when it is fully configured . . . 53

26. Wireshark capture of the TITAThink Camera showing access to the settings page is unauthorized . . . 54

27. Detected OS of a TCP SYN scan on the TITAThink camera . . . 55

28. Wireshark capture of the Proliphix thermostat show accessing the main page as well as an unauthorized page . . . 57

29. Homepage of Honeyd thermostat honeypot when it is fully configured . . . 58

30. Detected OS of a TCP SYN scan on the Proliphix thermostat . . . 59

31. Homepage of Honeyd outlet honeypot when it is fully configured . . . 60

32. Wireshark capture of the ezOutlet2 shown accessing the main page as well a nonexistent page . . . 61

33. Script for flipping and resetting the switch on the honeypot outlet . . . 62

34. ezOutlet2 Nmap OS Detection . . . 63

35. HTTP IoT Honeyd framework . . . 70

36. Example bar graph of average query response time . . . 79

37. Example bar graph showing code percent similarity for 1 user . . . 79

Figure Page

38. Example bar graph of TCP/IP and HTTP header percent similarity for the trials with 1 user 80

39. Example bar graph of number of TCP/IP packets 80

40. Example bar graph of average Nmap scan time 81

41. Example scatter plot of average query response time with trendline 81

42. Camera average query response time for device and honeypot 85

43. IoT camera average query response versus number of users 86

44. Camera honeypot average query response versus number of users 86

45. Thermostat average query response time for device and honeypot 89

46. IoT thermostat average query response versus number of users 89

47. Thermostat honeypot average query response versus number of users 90

48. Outlet average query response time for device and honeypot 92

49. IoT outlet average query response versus number of users 92

50. Outlet honeypot average query response versus number of users 93

51. Code similarity for camera with 1 user 94

52. Code similarity for camera with 10 simultaneous users 94

53. Code similarity for camera with 20 simultaneous users 95

54. Code similarity for thermostat with 1 user 96

55. Code similarity for thermostat with 5 simultaneous users 96

Figure Page

56. Code similarity for thermostat with 10 simultaneous users 97

57. Code similarity for outlet with 1 user 98

58. Code similarity for outlet with 10 simultaneous users 98

59. Code similarity for outlet with 20 simultaneous users 99

60. The number of TCP/IP packets sent by both the IoT camera and honeypot camera 100

61. The number of TCP/IP packets sent by both the IoT thermostat and honeypot thermostat 101

62. The number of TCP/IP packets sent by both the IoT outlet and honeypot outlet 102

63. Camera header similarity 1 user 103

64. Camera header similarity 10 users 104

65. Camera header similarity 20 users 104

66. Thermostat header similarity 1 user 105

67. Thermostat header similarity 5 users 106

68. Thermostat header similarity 10 users 106

69. Power outlet header similarity 1 user 107

70. Power outlet header similarity 10 users 108

71. Power outlet header similarity 20 users 108

72. Average Nmap scan time for camera 110

73. Average Nmap scan time for thermostat 112

74. Average Nmap scan time for power outlet 114

75. TITAThink camera query response time T-test 100 queries - 1 user 179

76. TITAThink camera query response time T-test 100 queries - 10 users 180

Figure Page

77. TITAThink camera query response time T-test 100 queries - 20 users 180

78. TITAThink Camera query response time T-test 500 queries - 1 user 181

79. TITAThink camera query response time T-test 500 queries - 10 users 181

80. TITAThink camera query response time T-test 500 queries - 20 users 182

81. TITAThink camera query response time T-test 1000 queries - 1 user 182

82. TITAThink camera query response time T-test 1000 queries - 10 users 183

83. TITAThink camera query response time T-test 1000 queries - 20 users 183

84. Proliphix thermostat query response time T-test 100 queries - 1 user 184

85. Proliphix thermostat query response time F-test and T-test 100 queries - 5 users 184

86. Proliphix thermostat query response time F-test and T-test 100 queries - 10 users 185

87. Proliphix thermostat query response time T-test 500 queries - 1 user 185

88. Proliphix thermostat query response time F-test and T-test 500 queries - 5 users 186

89. Proliphix thermostat query response time F-test and T-test 500 queries - 10 users 186

90. Proliphix thermostat query response time T-test 1000 queries - 1 user 187

91. Proliphix thermostat query response time F-test and T-test 1000 queries - 5 users 187

Figure Page

92. Proliphix thermostat query response time F-test and T-test 1000 queries - 10 users 188

93. ezOutlet2 query response time T-test 100 queries - 1 user 188

94. ezOutlet2 query response time T-test 100 queries - 10 users 189

95. ezOutlet2 query response time F-test and T-test 100 queries - 20 users 189

96. ezOutlet2 query response time T-test 500 queries - 1 user 190

97. ezOutlet2 query response time T-test 500 queries - 10 users 190

98. ezOutlet2 query response time T-test 500 queries - 20 users 191

99. ezOutlet2 query response time T-test 1000 queries - 1 user 191

100. ezOutlet2 query response time T-test 1000 queries - 10 users 192

101. ezOutlet2 query response time T-test 1000 queries - 20 users 192

102. TITAThink camera Nmap SYN times Mann-Whitney U test 193

103. TITAThink camera Nmap UDP times Mann-Whitney U test 193

104. TITAThink camera Nmap FIN times Mann-Whitney U test 194

105. Proliphix thermostat Nmap SYN times Mann-Whitney U test 194

106. Proliphix thermostat Nmap UDP times Mann-Whitney U test 195

107. Proliphix thermostat Nmap FIN times Mann-Whitney U test 195

108. ezOutlet2 Nmap SYN times Mann-Whitney U test 196

Figure Page

109. ezOutlet2 Nmap UDP times Mann-Whitney U test 196

110. ezOutlet2 Nmap FIN times T-test 197

List of Tables

Table Page

1. Summary of vulnerabilities and attacks found in wearable devices [9] 19
2. TITAThink camera and honeypot Nmap scan time Anderson-Darling results and final p-value 110
3. Proliphix thermostat and honeypot Nmap scan time Anderson-Darling results and final p-value 112
4. ezOutlet2 power outlet and honeypot Nmap scan time Anderson-Darling results and final p-value 115
5. TITAThink camera average query response time 172
6. Proliphix thermostat average query response time 172
7. ezOutlet2 average query response time 173
8. TITAThink camera HTML percent similarity 173
9. Proliphix thermostat HTML percent similarity 174
10. ezOutlet2 HTML percent similarity 174
11. TITAThink camera header information 175
12. Proliphix thermostat header information 175
13. ezOutlet2 header information 176
14. Nmap scan times for TITAThink camera and camera honeypot 176
15. Nmap scan times for Proliphix thermostat and thermostat honeypot 177
16. Nmap scan times for ezOutlet2 and outlet honeypot 178

EXAMINING EFFECTIVENESS OF WEB-BASED
INTERNET OF THINGS HONEYPOTS

I. Introduction

1.1 Background

The Internet of Things (IoT) is growing at an alarming rate. It is estimated that there will be over 25 billion IoT devices by 2020 [10]. These devices are always operating, gathering information taken from outside stimuli and transmitting it over the Internet while waiting for commands from the user. The simplicity of their function usually means that IoT devices have low processing power, which prevents them from having intricate security features, leading to vulnerabilities. Also, many times, vendors do not bother to install more intricate security measures or have good processes in place to update a device after a vulnerability has been found and fixed [10]. This makes IoT devices an ideal avenue for a would-be attacker to exploit.

Honeypots are intentionally vulnerable machines that run programs which appear as a vulnerable device to a would-be attacker. They are placed on a network to trap and entice an attacker and then gather information on them, including place of origin and method of attack. They usually only offer certain services and are by no means used by regular users on the network. Many honeypots run on their own custom software, but there are also honeypot managers that can be used to create multiple honeypots that run various services among other helpful features. Honeypots are useful tools for security professionals to place on their networks as a way to divert attackers from mission critical devices. Attackers can be slowed down

or locked out by interacting with a honeypot. Honeyd is a popular honeypot manager that can create many low-interaction virtual honeypots [11]. Instead of emulating an entire operating system, Honeyd simulates everything at the network level as well as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) services.

1.2 Motivation

With 25 billion IoT devices entering networks across the world [10], there will be an increasing number of malicious attackers who try to use them as a means of access into a network. Creating honeypots that simulate these services would be a perfect way to trap an attacker and possibly gather information about them. Moreover, devices with vulnerabilities are a perfect pairing for intentionally vulnerable honeypots.

Many IoT devices, such as Nest and Ring, now have mobile apps which is the main method for interacting with the user [12] [13]. However, some of these devices still have a web interface as well. Some of the IoT devices used in this thesis have mobile apps and web interfaces. If the web interfaces are ignored by the user, they remain with blank or default passwords. These web-based IoT devices could be more vulnerable than ever before. For this reason, it would be worthwhile to see how effective honeypots could be that utilize the Hyptertext Transfer Protocol (HTTP).

While Honeyd is an older program, it is still one of the most widely used honeypot managers today [4]. It provides several capabilities with its Operating System (OS) and network emulation. The scripting capability it provides can be used to create very convincing honeypots. Even web applications can be easily scripted by using the requests made to Honeyd as a pseudo-webserver. The challenge comes from seeing if the way IoT devices present their information on the web can be presented to a user through Honeyd in the exact same manner.

1.3 Research Goals

The goal of this research is to see if Honeyd can be used to create convincing honeypots that simulate IoT devices. The IoT devices interact with the user through a web interface with HTTP. The convincibility of the honeypot is directly related to how similar the honeypot is to the IoT device it is actually simulating. In theory, a honeypot that looks and functions exactly as a real IoT device is indistinguishable to an attacker. The hypothesis is that Honeyd is able to successfully create a near copy of the real IoT devices. This is determined through a combination of code, header, access time, and Nmap scan comparisons.

1.4 Approach

To determine how effective a honeypot is, it is modeled after a real IoT device and compared. To accomplish this task, this research is broken into four parts:

1. Create a network of IoT devices and gather information about them
2. Construct the honeypots and services to simulate the devices in Honeyd
3. Compare the IoT devices and honeypots, and make changes to the honeypots to increase their authenticity
4. Run tests on both the IoT devices and honeypots to obtain their response times, data, headers, and Nmap scan results for comparison.

creating a network of IoT devices and gathering information about them, constructing the honeypots and services to simulate the devices in Honeyd, comparing them to make additional changes to increase their authenticity, and experimenting on them.

1.4.1 IoT Network.

While there are countless IoT devices on the market today, this research is focused on ones with web interfaces. Selected devices include an IoT camera, thermostat, and power outlet. The camera is a TITAThink TT520PW. The thermostat is a Proliphix NT130h. The power switch is an ezOutlet2 EZ-22b. All these devices are place on the same Local Area Network (LAN), not connected to the Internet, and observed through a web browser. Their files are extracted by saving the page from the web browser to make localized copies of their web page. Using these and a combination of the actual Hypertext Markup Language (HTML) code on the web page, these files can be transferred to Honeyd to create a honeypot.

1.4.2 Honeyd Implementation.

Honeyd provides the ability to run scripts on service ports to give an attacker the illusion of a fully-functioning system. Using a bash script, Honeyd is able to function as a pseudo-webserver, taking requests from a client and sending correct HTTP response messages along with the corresponding data. However, this is only half of the necessary function. Some IoT devices, such as the camera and thermostat, have functions they perform and present to the user. A camera shows a camera feed, and a thermostat shows the current temperature. For these services, Python scripts are written to simulate a live camera feed as well as update the temperature. This way, when a user tries to access one of these pages, the honeypot gives the appearance of a dynamic IoT device as opposed to a static web page characteristic of some IoT honeypots.

1.4.3 Comparison.

Once Honeyd is properly configured and the honeypots are running, some observations are made. Access time is an important attribute to consider, as some IoT devices run on low-powered hardware and may take longer to access than the honeypot. In this case, delays may need to be built into the honeypot code to bring them more in line with the IoT device they are trying to simulate. If it is the other way around, more computing resources may be needed to speed up the honeypot. Also, Nmap is a common tool used as an automated network scanner. Both the IoT device and honeypot are scanned repeatedly to fine tune the detected operating systems and open services. Finally, some of the devices have unique ways of presenting a page to the user, such as redirect commands via HTTP. These can be discovered using Wireshark. Using Wireshark to observe the transmission of data from the IoT device to the user, the exact HTTP response codes sent can be observed and implemented into the Honeyd web server.

1.4.4 Experimentation.

In this research, experimentation has four main components that are used to compare the IoT device and honeypot: access time, HTML code, network headers, and Nmap scan.

1. Access Time: The main area of interest is how fast the IoT device responds when a user tries to access the main web page. While there are other pages on each device, this research focuses on how fast the honeypot responds to a request of the main page. In theory, timing should be similar for the other pages. This provides a good baseline to compare the access time between the IoT device and the honeypot. Requests are made from a differing number of simultaneous users as well. This increased load should increase the response

time from the IoT device and honeypot in the same way.

2. HTML Code: Since Honeyd has the ability to act as a pseudo-webserver, there should be no discrepancy between the files requested from the IoT device or honeypot. Each IoT device has multiple pages that can be accessed. The tests query different pages including a page that requires authorization, if the IoT device has it, and a page that does not exist. These pages are compared to determine the number of differences between the IoT device and honeypot. Items that are expected to change, such as date, time, temperature, Internet Protocol (IP) addresses, etc, are not factored into the differences.

3. Network Headers: When each page is accessed, the HTTP headers are stored in a file, and there is another program listening for all of the incoming TCP and IP headers. The tests compare the IoT device and honeypot to ensure they contain the same fields and that those fields have the correct values. The TCP and IP packet header fields are typically deterministic, but HTTP header fields can be different depending on the device. Therefore, only the TCP/IP header values need to be compared, but it is necessary to compare both the HTTP header fields and values.

4. Nmap Scan: The Nmap scan provides various information about the operating system and services. The tests are run, on the IoT device and the honeypot, to determine that the services are the same and the similarity of the operating systems.

1.5 Assumptions and Limitations

1.5.1 Assumptions.

In evaluating the devices and honeypots in this research, the following assumptions are made:

- Each connection attempt ends with a success. If a connection fails, an entirely empty web page is used for comparison purposes.
- The same commands are used on both devices.
- Date and time of access are different for each attempt.
- Non-standard commands are not attempted against any device.
- All interactions take place on the same network with the same network hardware.
- All interactions begin from the same location with the same hardware. Each interaction uses the same machine so processor specifications are not a factor.
- The same version of all software is used.
- There are no outside interaction with the devices other than the user in the scenario.

1.5.2 Limitations.

1.5.2.1 Singular Task.

This research is only focused on the HTTP web application of these IoT devices. Some of these devices have other services available, but they are not explored in this research. These services are only enabled in the Honeyd configuration file, making them observable by Nmap.

1.5.2.2 Nmap Version.

The version of Honeyd used in this research is 1.5c. This version uses an older Nmap fingerprint database format. Because of this, the OS scanning on the honeypots with the latest version of Nmap may not produce ideal results. The Honeyd honeypots do not have the option to choose some of the operating systems that Nmap can detect in its latest version. This research is mostly focused on how convincing IoT honeypots are with the last version of Honeyd that Provos produced.

1.5.2.3 Computer Resources.

The honeypot is running on a glsvm running Ubuntu 12.04.5, and the resources are constrained to allow running other virtual machines without slowing down the function of the laptop on which everything is running. Therefore, if a honeypot is responding slower than one of the physical devices, a more powerful machine hosting the virtual machines would be required. In this research, honeypots are able to be slowed down, but not sped up.

1.6 Research Contributions

The research in this thesis tries to recreate as close as possible the appearance and function of actual IoT devices to determine whether or not Honeyd is a suitable framework within which to simulate these devices. It assesses how well Honeyd is capable of simulating web IoT devices that have static pages of information, such as the Proliphix thermostat and ezOutlet2, and how it may not be optimal for simulating web IoT devices that have a dynamically-updating page, such as the TITAThink camera. This research could be used by defense agencies and companies who seek to delay or examine attackers on their own networks by making convincing honeypots of IoT devices, one of the most prevalent devices that make up the Internet today,

quickly and with limited resources.

1.7 Thesis Overview

Chapter II contains an overview of the Internet of Things, honeypots, and the other pieces of software related to this research as well as some background research on IoT honeypots. Chapter III provides a description of the Honeyd-based system to simulate the three IoT devices as well as the physical system they are tested against. Chapter IV details the design of the experiments with the corresponding results in Chapter V. Chapter VI outlines the conclusions drawn from this research as well as potential areas to explore in future research.

II. Background and Related Research

2.1 Overview

This chapter details relevant background information regarding Internet of Things (IoT) and honeypots. Section 2.2.1 discusses the background, protocols, and security of the IoT, and Section 2.2.2 discusses honeypots. Sections 2.2.3 and 2.2.4 discuss the tools used for this research: Nmap and Honeyd. Finally, Section 2.3 covers current research that has been done in combining the IoT and honeypots into security tools.

2.2 Background

2.2.1 Internet of Things (IoT).

The IoT is a blanket term used to describe the interconnection of various devices and sensors through the Internet. IoT devices take in outside stimuli, react to it, and communicate with one another through the Internet [14]. There are IoT sensors, appliances, and cameras just to name a few. These devices are usually always running, meaning they are constantly performing their functions and interacting with each other through the Internet.

One example of a device is an IoT thermostat that has the ability to control the temperature of a home dynamically. It can be changed by the user through the Internet, or it can be programmed to change at certain times. It is always running, waiting on a command to change the temperature. Overall, its function is quite simple. The simplicity of the function usually means that IoT devices have low processing power, which can lead to vulnerabilities.

Suo et al. have divided the IoT functionality into 4 layers (Figure 1): Application, Support, Network, and Perceptual [1]. The Perceptual layer deals with data gathering. Physical sensors such as temperature, GPS, and cameras collect data for

processing on the device. The Network layer is responsible for the reliable transfer of this data. It mostly passes through the Internet but also through other transportation mediums such as Bluetooth. The Support layer deals with the computation of the data on the IoT device. This can be performed on the device itself, but due to the low processing power of many IoT devices, it is frequently done remotely through cloud computing. Finally, the Application layer provides services to the user, such as the graphical user interface that displays the data from the sensors, and allows manipulation of the device by the user. This model is a useful way to highlight the different portions of what makes up an IoT device.

The number of IoT devices is growing every day. It is estimated that there will be over 25 billion IoT devices by 2020 [10]. Not only does the limited processing power lend to their insecurity, but some of their protocols do as well. The main protocol used by these devices is the Internet Protocol (IP); however, there are also other protocols that add additional layers of functionality and insecurity.

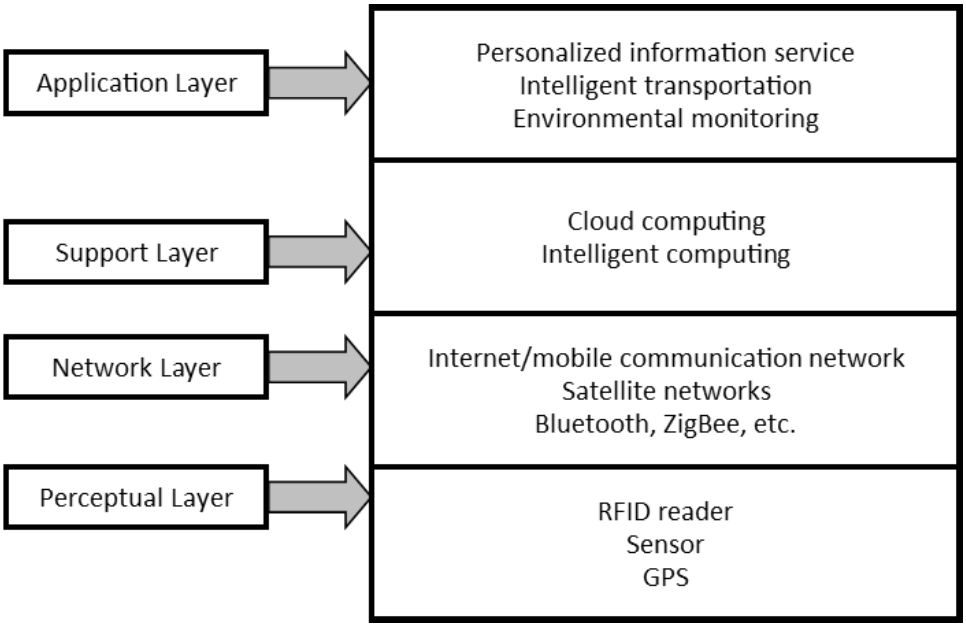


Figure 1. IoT architecture [1]

2.2.1.1 IoT Protocols.

Most protocols in use by IoT devices are different than ones that have been the standard for other Internet-connected devices. They add additional functionalities and abilities for users to interact with their device. Sometimes the devices transfer personal information that is collected or stored within the device. Some of these protocols include Bluetooth, ZigBee, and WirelessHART. These protocols can add a new dimension into fingerprinting IoT devices as well as introducing more security vulnerabilities.

The majority of IoT devices are wireless and use exclusively wireless protocols. They usually connect to the Internet through IEEE 802.11, also known as WiFi. However, some IoT devices have such a small amount of processing power, that WiFi is not feasible given the required overhead. Some devices utilize the protocol IEEE 802.11ah (hereafter referred to as 802.11ah), which is a lower-overhead version of the standard IEEE 802.11 [14]. Figure 2 shows the headers of both protocols. 802.11ah is ideal for IoT devices that sacrifice processing power to be compact and may need to conserve battery life. Some features of 802.11ah that assist IoT devices include the reduction of the Media Access Control (MAC) frame from 30 bytes to 12 bytes. It also replaces the 14-byte ACK frame for a tiny signal called a preamble that is used instead. There is a synchronization frame that only allows stations with valid channel information to transmit after reserving the channel medium. Finally, there is also efficient bidirectional packet exchange that reduces power consumption by allowing IoT devices to go to sleep as soon as they finish their communication [14]. While IoT devices are often resource constrained, protocols such as these improve their functionality and efficiency.

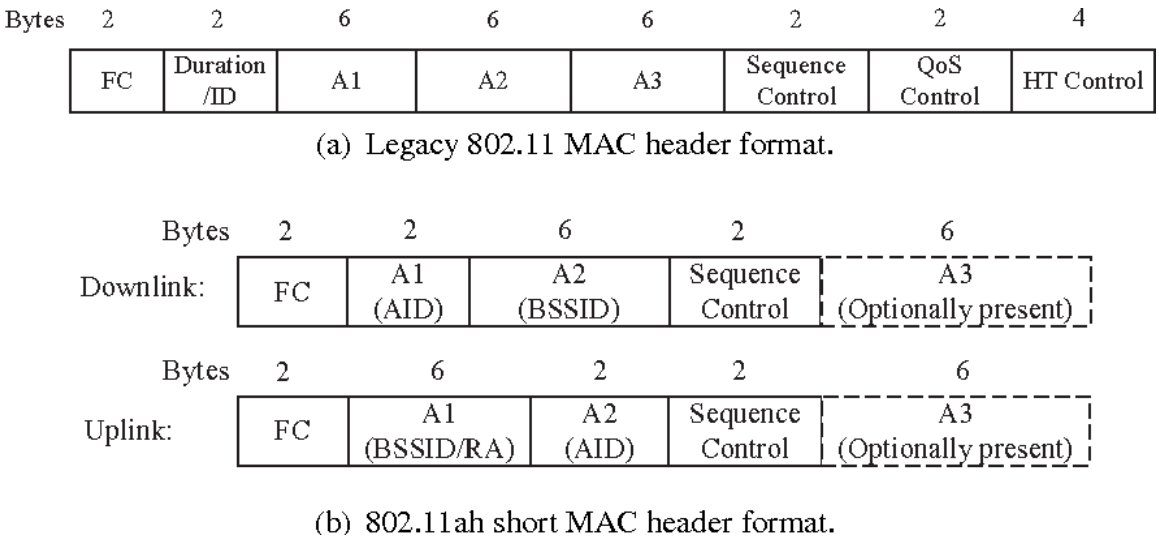


Figure 2. MAC headers for IEEE 802.11 and IEEE 802.11ah [2]

Bluetooth is another common protocol in IoT devices. It is a standard for short range, low power, low cost wireless communication that uses radio technology [15]. One common feature of IoT devices is the lack of user interface on the physical device. There is no mouse or keyboard to modify how the device functions or how it presents information to the user. Some devices do have a small touchscreen interface, but even these devices usually still have a Bluetooth connection to a user's smart phone. Bluetooth requires 100 mW of power, which is too large of a power draw for some IoT devices [16]. They require low power due to their small size and 24/7 operation. Bluetooth Low Energy (BLE) is often the solution. BLE is very similar to standard Bluetooth; the only difference is that latency is fifteen times quicker and transmissions only reach 10% as far due to low power with a minimum power output of 0.01 mW and a maximum of 10 mW [14][17]. It uses a master/slave architecture where slaves advertise on special channels and masters sense this and connect to the device. Nodes stay awake only when communicating and sleep otherwise to conserve power, which is beneficial for IoT devices [14].

While Bluetooth is a common protocol in mobile computing and IoT devices, ZigBee is a protocol that is similar to BLE and is almost exclusively used in IoT devices. ZigBee is a protocol based on IEEE 802.15.4, low-rate Wireless Personal Area Network (WPAN), for communication between devices within 10 meters [18]. While Bluetooth has a master that connects to a slave, ZigBee uses a central coordinator and all the devices connect to the coordinator [14]. There are two types of devices in a ZigBee network: a Full-Function Device (FFD) and a Reduced-Function Device (RFD). A FFD can act as a coordinator or a device, while a RFD is reserved for low-powered simple devices [18]. As shown in Figure 3, the central FFD acts as the coordinator, and more FFDs acting as routers or devices connect to it, and some RFDs connect to these FFD routers. This creates a star topology where devices communicate to the coordinator and the coordinator passes the message along to other devices on the network. It does not provide any security features unless the ZigBee Pro protocol is used. ZigBee Pro has more features such as symmetric key exchange, scalability with stochastic address assignment, and improved performance with a more computationally efficient many-to-one routing mechanism [14]. This security is provided through the more powerful central coordinator, taking some of the burden off the low-powered IoT device.

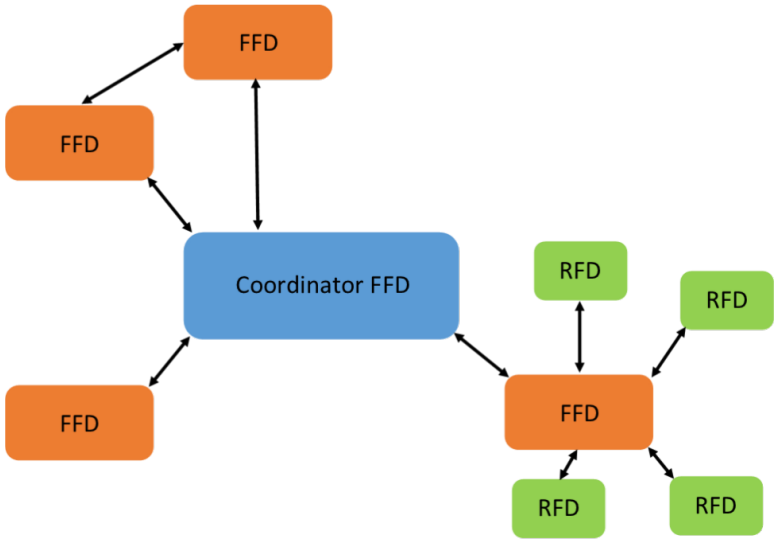


Figure 3. Example ZigBee network topology adapted from [3]

The main takeaway from these IoT protocols is in the requirements of these devices. All of these protocols are specifically designed to operate with the low processing power and short range communication of IoT devices. This is the major distinction of IoT devices compared to desktop computers. Another impact of the design of these devices is the lack of security within this protocol. These protocols make compromises on security to compensate for lower processing capabilities.

2.2.1.2 IoT Security.

Security managers point to the increasing number of IoT devices as a cause for concern within network security. As previously stated, the lack of processing power does not allow for security features such as encryption on the devices themselves or within their protocols [1]. This is driven mainly by the vendors' priorities which are on ease of use and getting their products to market quickly. The IoT architecture is not equipped to handle security updates, and manufacturers do not provide a mechanism to provide security updates [10]. Another reason IoT devices have security

as an afterthought is that everyday consumers do not care as much about security as the private sector or the military. Some argue that if the Department of Defense (DoD) sought to implement IoT devices, the industry would be enticed to put more research and development into the security of IoT devices to obtain military contracts [19]. A study by Hewlett-Packard found that 70% of IoT devices were vulnerable to attack [20]. Many of the IoT devices that are used today by consumers have these documented vulnerabilities.

Suo et al. highlight some of the most important aspects today within IoT security: identification, confidentiality, integrity, and non-repudiation [1]. They also list the four layers of the IoT (Figure 1) and why some of the aspects are challenging. In the Application layer, sharing is one of the characteristic features which creates a challenge for data privacy, access control, and disclosure of information. The Support layer is the intelligent computing (sometimes cloud) platform for the Application layer. Its challenge is preventing malicious information from making its way into data processing. The next layer is the Network layer, which deals with reliable data transfer between the Perceptual layer and the Support layer. The Network layer is susceptible to man-in-the-middle and counterfeit attacks. The final layer is the Perceptual layer which contains the sensors for capturing the physical world and translating it into the computing system. The challenge here is that the lack of computing power in sensors does not leave room for security features like encryption. This is one of the main challenges found with IoT security, as many of the protocols do not include security features.

Patton et al. use Shodan, a web search engine for IoT devices, and a database of default passwords to see if they could find IoT devices for which they could log into easily [21]. They scanned for Supervisory Control and Data Acquisition (SCADA) devices, IoT cameras, and printers to see which ones were vulnerable. They found that

4% of the SCADA devices were vulnerable, 40% of traffic cameras were vulnerable, and 41% of the printers were vulnerable. This was all using default login credentials. In this case, the vulnerabilities are the fault of the user. This is due to the perception that an insecure IoT device has no bearing on overall network security. While logging into a printer may seem innocuous, updating it with custom firmware can give the attacker full control over the printer and the ability to attack the network to which it is attached.

The BLE protocol in use by many IoT devices can also be a vulnerability. Using a tool called Ubertooth, Ryan was able to sniff BLE traffic as well as inject packets and break the encryption of BLE [22]. Because the protocol is intended for low-powered devices, the protocol is simplistic, which aids in the creation of an eavesdropping tool. Ubertooth is a USB dongle that can monitor a single BLE channel at any given moment. It has a partial sniffer for Bluetooth, but since BLE is a simpler protocol, the packets can be processed entirely on the dongle, making sniffing even easier. Ryan documents exactly how to sniff this protocol using this tool and provides a proof of concept injection attack.

Some devices using the popular IoT protocol BLE are Nest Cameras [23], FitBit [24], Tile Item Locators [25], and smart light bulbs [26]. The BLE protocol has fewer security and privacy features and consumes less energy compared to regular Bluetooth. For example, the pairing process of the FitBit and smartphone through BLE is not encrypted. To combat this, developers implementing BLE make the devices frequently change their private address with each pairing in order to avoid tracking. FitBit however does not do this, and a malicious attacker could obtain the MAC address and BLE credentials through simple Bluetooth sniffing [24]. Using the process laid out by Ryan, much information could be gleaned from sniffing this protocol on FitBit and other IoT devices [22].

The low processing power of smart wearable devices, primarily due to the small form factor and the lower computational requirements, limits the complicated security mechanisms that can be implemented by developers in these devices. Ching and Singh examine some wearable computing devices including Google Glass, FitBit, and Samsung smart watches as well as some security vulnerabilities discovered (Table 1) [9]. Many of these vulnerabilities are present due to protocols like BLE, that can be easily sniffed with certain tools. For example, they cite research from Bitdefender that had a proof of concept hack where they could access the six-digit PIN code and Bluetooth traffic between a Samsung smart watch and Google Nexus phone through brute force by using any open source Bluetooth sniffing tool. Strong authentication was not used [9]. These wearable devices are part of the IoT and the same protocols, including their vulnerabilities, are found in other stationary IoT devices that are finding their way into homes and the workplace.

Table 1. Summary of vulnerabilities and attacks found in wearable devices [9]

Wearable Devices	Security Vulnerabilities	Attacks
Google Glass	Insecure PIN system or authentication in place	The gesture-based authentication scheme easily to be recorded by people nearby
	Privacy: pictures and videos can be recorded without user's consent and unauthorized eye movement tracking.	Eavesdropping and spyware
	It relies on QR codes for WiFi setup	QR photobombing malware
	Insecure network and hostile environment	WiFi hijacking, man-in-the-middle attacks such as session hijacking or sniffing.
Fitbit Devices	Lack of authentication	Data injection attack, Denial of Service (DoS), and battery drain hacks
	Leaky BLE technology	It can be easily tracked
	Privacy: Users location or places visited can be tracked	Phishing
Samsung Smartwatch	Authentication mechanism not secure enough	Brute force attack

2.2.2 Honeypots.

The number of IoT devices is growing drastically, and it is important to understand exactly what types of devices are considered part of the IoT. The increase in the number of these devices is becoming a security concern within computer networks. Honeypots could be the perfect pairing for devices that lack security. When trying to secure a network, a common course of action would be to remove all of the known vulnerabilities on the network. Honeypots are intentionally-vulnerable machines placed on a network that have the appearance of a working computer system complete with an operating system, services, and even a network [4]. In actuality, they are a sealed compartment used to lure in and contain an attacker. A tremendous amount of information can be learned about the attacker because a log of their actions is recorded by the honeypot including access attempts, keystrokes, files accessed, files modified,

and processes executed [27]. This information can be used to capture and later analyze the type of malware an attacker was using. It also allows security experts to determine methodologies of their attackers when trying to infiltrate a network. Honeypots can even be used as an alternative to an Intrusion Detection System (IDS). One advantage of Honeypot devices over other IDSs is that they are able to interact directly with an attacker at the application level [5]. This means the attacker has a convincing device that they assume to be the real thing.

There are two main types of honeypots: physical and virtual [5]. A physical honeypot is a physical device on the network with software to handle the honeypot logging capabilities. A virtual honeypot is a device that simulates one or many devices through software such as virtualization. They are more economical because one device can create many honeypots.

The value of a honeypot comes from the value of the information gathered [28]. This is why it is so important for a honeypot to be a convincing target for an attacker. One of the ways they can be implemented is as a high-interaction honeypot. These are actual computer systems such as a Commercial Off-the-Shelf (COTS) computer, router, or switch [5]. High-interaction honeypots are usually implemented as a physical honeypot (i.e., actual computing systems), but they can be implemented as a virtual honeypot as well. Regardless of implementation, high-interaction honeypots should not produce additional network traffic other than the traffic of the COTS device it is contained on. This makes it a extremely convincing honeypot, because it is an actual computer system. Every interaction with a high-interaction honeypot would be suspect, because no legitimate user would have a reason to interact with it. Therefore, all network traffic is logged and analyzed [28]. Several honeypots can be placed together to create a simulated network of devices, called a honeynet. This is more difficult with high-interaction honeypots because as physical machines they take

up physical space. Fully virtualized machines take up computing resources instead of space. This means high-interaction honeypots lack scalability.

Low-interaction honeypots are very scalable. Low-interaction honeypots simulate services, the network stack, and other aspects of a real machine allowing for only limited interaction with an attacker [5]. The interaction does not need to provide complete functionality of a simulated service or protocol. For example, a Hyptertext Transfer Protocol (HTTP) server may only serve requests for a certain URL and only implement a portion of the HTTP protocol. It only needs to be convincing enough to fool an attacker into believing that it is a real machine. Because of the low functionality requirements, it is much easier to create many low-interaction honeypots which can be tailored to a specific function. Low-interaction honeypots may seem more limited, but they are useful for gathering information at a higher level, such as network activity [5]. That along with their ease of implementation make them very powerful tools.

Another difference between low-interaction and high-interaction is the risk that they pose. Because low-interaction honeypots are not full production machines, an attacker can only perform the functions allowed by the honeypot developer. A high-interaction honeypot, that has the full capabilities of a production machine, could provide a new avenue for attack on the network, if implemented improperly. For example, if a honeypot runs a full operating system and is used as a jail to keep an attacker in, they could potentially find a way out of the cage and use the honeypot to launch attacks against the network [4]. Fortunately, low-interaction honeypots do not have much risk associated with them because of the limited functions available to a potential attacker.

One common use for a honeypot is as an IDS. A traditional IDS detects attacks against the network based on the signature of the attack [29]. Certain exploits have

specific actions they take against a system to execute; an IDS alerts the network administrator when any of these actions are taken. A honeypot used as an IDS sits on the network as a regular machine as shown in Figure 4. However, no legitimate user would have a reason to interact with it, or even know that it is there. If anyone does interact with it, there is a high probability an intruder is on the network. Honeypots also produce fewer false positives, since interacting with a honeypot would not be normal activity [4]. A regular IDS might flag legitimate work, such as the network administrator viewing the Unix password file, /etc/shadow. These false positives create a wave of alerts that are eventually ignored by network administrators.

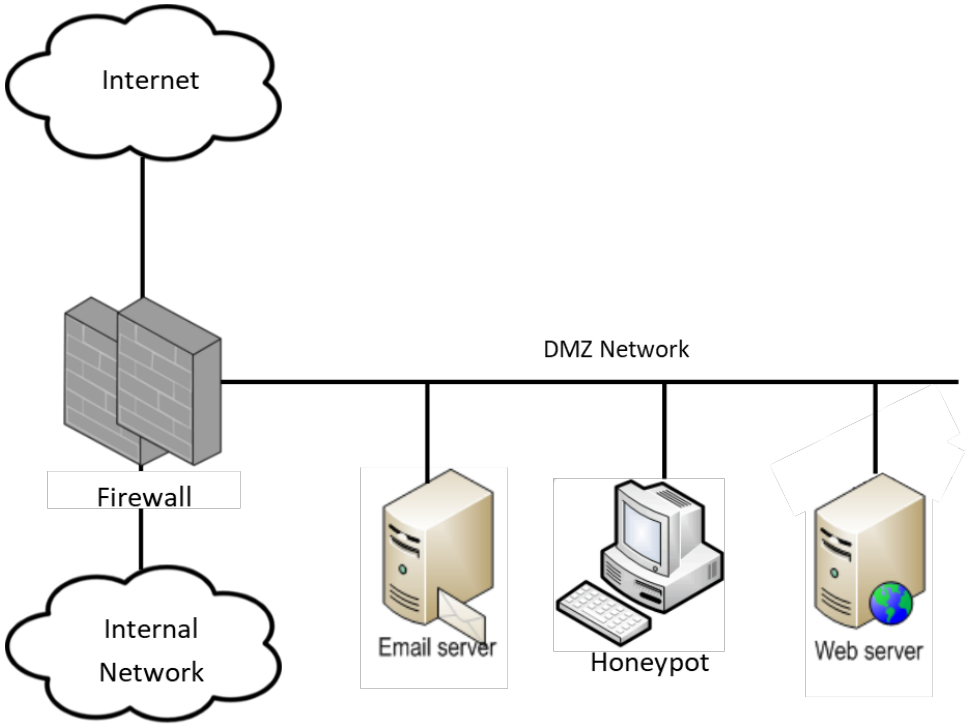


Figure 4. Honeypot deployed as an IDS adapted from [4]

2.2.3 Nmap.

Nmap is an open source tool that can generate packets to scan enterprise networks to identify live hosts, their services available, and operating systems [30]. It is widely

used in cyber operations to gather information on targets and has a wide variety of features. It can automatically scan just a single address or an entire subnet. It identifies the services open on different ports and the operating system on the machine.

The operating system is determined using fingerprinting. Nmap sends a series of TCP and UDP packets and performs tests on the responses including: TCP Initial Sequence Number (ISN) sampling, TCP options support and ordering, IP Identification (ID) sampling, and the initial window size check [31]. This information is then compared to a database to see if it matches any other known fingerprints. It provides the vendor name, underlying Operating System (OS), OS generation, and device type [31]. This functionality can be utilized by Honeyd to create more realistic virtual honeypots.

2.2.4 Honeyd.

Honeyd is open-source software written by Niels Provos that creates low-interaction virtual honeypots [11]. Instead of simulating an entire operating system, Honeyd simulates everything at the network level and simulates TCP and UDP services [11]. IoT devices usually only perform a single simple function, which makes Honeyd an ideal program for creating honeypots based on these devices. A common theme among IoT devices is a simple service that a user interacts with, which makes them a good candidate for a low-interaction honeypot.

Honeyd uses a configuration file to store all of the defining information about a honeypot. Figure 5 shows an example Honeyd configuration file. The route portion of the code provides an example of how Honeyd allows the user to create a honeypot router by providing routing information to the various honeypots. The configuration file creates two honeypots: routerone and netbsd. The simulated operating system is set with the `set <honeypotName> personality` command along with a fingerprint

from Honeyd's `nmap-os-db` file. This fingerprint is then used to simulate the device's TCP and UDP behavior on the network stack [28]. Default TCP/UDP actions can be set with `set <honeypotName> default tcp/udp action` command. Default TCP and UDP actions that can be taken include: Reset, Open, Block, or Script [4]. Reset means that Honeyd will send a TCP Reset (RST) or Internet Control Message Protocol (ICMP) unreachable signal for specified TCP and UDP ports. Open means that Honeyd will send an Acknowledgement (ACK) for all TCP data received, but no service will be simulated. Block means that Honeyd will ignore connections on specified ports. Finally, using `add <honeypotName tcp/udp port <port number> <scriptName>`, a specific script is bound to the specified port. Usually a user sets a default action for all ports then creates specific scripts for individual ports. Common languages for these scripts include shell scripting, C, Python, or Perl. These scripts are how Honeyd simulates the application level and how an attacker will primarily interact with the honeypot. Whenever a user tries to access this port, the script is executed. This is how the honeypot interacts with a user. The user sets any IP or MAC address to the virtual honeypot using the `bind` command or allow Honeyd to assign them automatically using the `dhcp` command. One configuration file can create many virtual honeypots all on the same machine.

```
route entry 10.0.0.1
route 10.0.0.1 link 10.0.0.0/24
route 10.0.0.1 add net 10.1.0.0/16 10.1.0.1 latency 55ms loss 0.1
route 10.0.0.1 add net 10.2.0.0/16 10.2.0.1 latency 20ms loss 0.1
route 10.1.0.1 link 10.1.0.0/24
route 10.2.0.1 link 10.2.0.0/24

create routerone
set routerone personality "Cisco 7206 running IOS 11.1(24)"
set routerone default tcp action reset
add routerone tcp port 23 "scripts/router-telnet.pl"

create netbsd
set netbsd personality "NetBSD 1.5.2 running on a Commodore Amiga (68040 processor)"
set netbsd default tcp action reset
add netbsd tcp port 22 proxy $ipsrc:22
add netbsd tcp port 80 "sh scripts/web.sh"

bind 10.0.0.1 routerone
bind 10.1.0.2 netbsd
```

Figure 5. Example Honeyd configuration file [5]

Figure 6 shows an example Nmap fingerprint, which comes from the database of OS fingerprints used by Nmap. The code in this file is the same format as the fingerprint used by Nmap. Nmap performs a series of tests on a device and an output similar to Figure 6 is generated. Nmap tries to match this output to fingerprints in the database. The closest match it finds is the OS it detects.

```
# Microsoft Windows 2000 Server [Winver: Version 5.0 (Build 2195: Service Pack 4)]
# Windows XP Professional Version 2002 Service Pack 3
Fingerprint Microsoft Windows 2000 Server SP4 or Windows XP Professional SP3
Class Microsoft | Windows | 2000 | general purpose
CPE cpe:/o:microsoft:windows_2000::sp4:server
Class Microsoft | Windows | XP | general purpose
CPE cpe:/o:microsoft:windows_xp::sp3:professional
SEQ(SP=FE-108%GCD=1-6%ISR=102-110%TI=I%II=I%SS=S%TS=0)
OPS(O1=M5B4NW0NNT00NNS%O2=M5B4NW0NNT00NNS%O3=M5B4NW0NNT00%O4=M5B4NW0NNT00NNS%O5=M5B4NW0NNT00NNS%O6=M5B4NNT00NNS)
WIN(W1=4470%W2=41A0%W3=4100%W4=40E8%W5=40E8%W6=402E)
ECN(R=Y%DF=Y%T=7B-85%TG=80%W=4470%O=M5B4NW0NNS%CC=N%Q=)
T1(R=Y%DF=Y%T=7B-85%TG=80%S=O%A=S+%F=AS%RD=0%Q=)
T2(R=N)
T3(R=N)
T4(R=N)
T5(R=Y%DF=N%T=7B-85%TG=80%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
T6(R=N)
T7(R=N)
U1(DF=N%T=7B-85%TG=80%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)
IE(DFI=S%T=7B-85%TG=80%CD=Z)
```

Figure 6. Example of an Nmap signature database file. This signature is for Windows XP SP3 or SP4 [6]

One advantage of Honeyd is how quickly it can create new honeypots. In theory, a network admin could easily fill an IP space with virtual honeypots. This would not only add obscurity to a network, but any interaction with these machines from the outside would surely be malicious since they do nothing on their own. IoT honeypots could blend in with normal IoT devices already on a network.

The power of Honeyd is truly shown by the ability of the user to add scripts on specific ports to simulate services. Users have the ability to use any scripting language they desire to simulate many services. Honeyd comes with a sample shell script (Figure 7) that supplies an Secure Socket Shell (SSH) banner then logs and echos the entered text [4].

```
DATE=`date`
echo "$DATE: Started From $1 Port $2" >> /tmp/log
echo SSH-1.5-2.40
while read name
do
        echo "$name" >> /tmp/log
        echo "$name"
done
```

Figure 7. `test.sh` **script packaged with Honeyd [7]**

The challenge presented by Honeyd is making these scripts convincing. As with all low-interaction honeypots, how convincing the honeypot is depends entirely on how realistic it appears to an attacker. Honeyd does a good job of simulating the network stack in a convincing way, but if the service is unconvincing, an attacker will see through the facade.

2.3 Related Research

2.3.1 IoT Honeypots.

Some research has been done to try and create varying types of IoT honeypots that perform different functionls. The following research involves custom honeypots. The researchers created their own systems to simulate IoT services and log connections. They did not use Honeyd, rather other honeypot programs.

Pa Pa et al. developed a system called IoTPOT and IoTBOX [32]. IoTPOT is a honeypot system that simulates Telnet services that are used by various IoT devices. They found various types of IoT devices using this protocol that had been compromised including Digital Video Recorders (DVR), IP Cameras, Wireless Routers, TV Receivers, etc. They focused on presenting realistic experiences to an attacker including accurate welcome messages and support for multiple Central Processing

Unit (CPU) architectures. They also developed an IoT malware analysis environment called IoTBOX, which allows the analysis of malware on eight different CPU architectures. It works hand in hand with IoTPOT to interact with the attackers and document their attack methodologies. They discovered different attack patterns used by attackers against Telnet-based IoT devices. They also claimed that Honeyd would not have been capable of replicating their system's functionality. This research only focused on Telnet-based attacks, but they hope to do more research on SSH attacks and expand IoTBOX to handle different architectures used on IoT devices.

HoneyIo4 is a virtual honeypot that simulates four IoT devices: camera, printer, video game console, and cash register machine [33]. It does not simulate their full functionality but does simulate their protocols to trick network scanners like Nmap. This was written with Python and Scapy, not a program like Honeyd. This work could be expanded upon by trying to get more realistic network interaction through a program like Honeyd. The OS fingerprinting of IoT devices is an important part of that; however, an actual honeypot would be able to interact with a user that tries to connect to it. Honeyd does not have downloadable profiles for many smart devices because it is an older program, so this research could be very helpful in simulating the protocols of smart devices within Honeyd and then implementing an actual service to interact with an attacker.

Krishnaprasad created his own IoT honeypot that could simulate IoT services, including Telnet, SSH, HTTP, and Customer-Premises Equipment Wide Area Network Management Protocol (CWMP) [34]. The inspiration for his research came from the creators of IoTPot with the goal of creating a lightweight honeypot to handle all protocols used by IoT devices. He did not use a honeypot manager like Honeyd, rather a variety of packages and programs to handle the different functions of his honeypot. It was broken into two parts, the frontend and the backend. The frontend is a

low-interaction responder, and the backend is a high interaciton virtual environment. The frontend deals with the interaction between an attacker and the IoT device. It has various programs that handle Telnet, SSH, HTTP, and CWMP requests. The program uses different Python libraries that are linked together with Docker. Krishnaprasad claims that the frontend is flexible enough to support any kind of backend as long as the frontend protocols are supported. When the honeypot was deployed, it received many attacks, with Telnet being the most commonly attacked protocol. Since Telnet is so widely used in IoT devices, an IoT honeypot would be useful to capture the multitude of attacks targeting Telnet. Future research includes creating a more robust backend or turning it into a network, so there would be multiple devices for which an attacker can interact.

2.3.2 Honeyd.

Honeyd is an older program, but research has still been done to create IoT honeypots using it. Freeman and Woodward created a first generation smartphone honeypot that simulates a Windows Mobile 5 device [35]. They used Honeyd to simulate the OS in order to detect smartphone worms. The only problem is that Honeyd only supports the first generation Nmap fingerprint for operating systems, and currently there is no update to implement this feature. In the future, it would be important to implement different phone operating systems like iOS or Android. When trying to create honeypots for other smart devices this limitation in Honeyd may pose an issue requiring additional research to rectify.

Kreibich and Crowcroft used honeypots to create a system that generates signatures for malicious network traffic automatically [36]. The honeypots automatically analyze traffic on the honeypot and then generate the signature that describes the characteristic elements of the attack. The researchers discuss how honeypots usually

exist in a passive role, but they argue that their system, Honeycomb, is the first step in making honeypots more active in network security. They use Honeyd to implement their system, which means that any system simulated using this software could potentially make good use of their work, including Internet of Things devices.

2.3.3 Uses for Honeypots.

Others have hypothesized about ways that honeypots could be used within network security. Honeypots are normally used to catch external hackers, but Spitzner proposes that honeypots could be used to help catch insider threats [37]. He suggests things like: Honeynets, which could store fake sensitive information in order to observe who on the network is accessing it or Honeytokens, fake credentials that would alert security if an insider tries to use them to access a system. The difference between these types of honeypots and others is that they have to be placed on the internal network, and work has to be done to try and trick the insider threat into interacting with the honeypot. They likely will try to be sneaky and not simply scan the entire network for open ports. One example is sending a fake email to an executive with Honeytokens in them. A Honeytoken is a login credential that provides a notification whenever someone tries to use them to log into a system. If someone tries to log into a system with the Honeytokens then someone else must have access to that executive's email account. The attacker can then be sought out in the organization. The author provides an interesting use for honeypots that go beyond conventional thinking. There may be ways that IoT honeypots could be implemented in a similar manner.

Zhang et al. implemented a cyber security defensive system that distributed passive and active network sensors to capture suspicious information associated with cyber threats [38]. It also included a dynamic firewall that allows hosts to update

their detection information to block attacks. They implement honeypots as passive scanners on the network, while simulating regular services and provided information when attackers tried to access those services. The firewalls were dynamically configured based on the information of malicious traffic trying to connect to the honeypots. They used Windows XP, Windows 2003, Avaya, and Solaris as the simulated services on these boxes. The honeypots were only simulating regular operating systems and standard computer services on a network. They did not look into smart devices on the network or how they could also be leveraged to provide that information. In theory, Windows XP is a very vulnerable service in the same way that IoT devices are. IoT devices could be used as passive network scanners in a similar way.

2.4 Chapter Summary

This chapter examines the basics of IoT and its protocols. It also provides some of the security challenges and innate vulnerabilities with IoT. These vulnerabilities make them ideal targets for an attacker. Honeypots are intentionally vulnerable machines on the network that can be used to track and log an attacker. Combining IoT devices and honeypots create a perfect environment for luring attackers and gathering information on them.

Much of the research that has been done focuses on aspects of creating an IoT honeypot, such as simulating their protocols or simulating the network stack. Little has been done to create a honeypot that completely replicates the services of a specific IoT device. This replica device should be convincing to an attacker by showing a legitimate looking service and having convincing network traffic. The addition of vulnerable IoT devices to networks is continuing to grow. Adding IoT honeypots as well would blend them in with the others, becoming a powerful tool for network security.

III. Framework Design

3.1 Overview

This chapter details the system where IoT devices are simulated by honeypots and how they are compared to the real device. The honeypots are created using Honeyd and a simple web server that provides the web page of the IoT device when queried. A series of Python scripts make web requests to the honeypot and real device and then compare the results to determine how similar they are. Other Python scripts perform various scans using Nmap and compare those results as well. The motivation for these processes are provided in Section 3.2. The physical IoT devices under test are outlined in Section 3.3. The design parameters for the IoT honeypots are detailed in Section 3.4.

3.2 Motivation and Application

One of the most common protocols used by IoT devices is HTTP; however, much of the research dealing with IoT honeypots has focused on the Telnet protocol [32] or with simulating proper TCP responses that they receive [34]. A honeypot that provides a full web service to interact with in the same way as a physical IoT device would be very worthwhile.

If a honeypot has any chance of fooling an attacker, the service it provides needs to be as identical as possible to a real device. This is even more true if the device is something the attacker is very familiar with. The main item that an IoT device provides to a user is data. Therefore, a honeypot simulating an IoT device should produce nearly identical data in the same amount of time as the device it is replicating. The system outlined in this thesis tries to provide nearly identical Hypertext Markup Language (HTML) code as actual IoT devices. Time delays are also incorporated to

make them more similar to the actual IoT device. For this reason, the tests compare the HTML code of the honeypot and the IoT device to determine similarity.

Along with the data, another attribute of HTTP, TCP, and UDP packets sent on a network is that they all have headers that provide information on the packet that contains the data. HTTP headers in particular are interesting because different devices have different fields in their HTTP headers. Honeyd generates the TCP and IP headers sent by the honeypots in this system, and the system also creates HTTP headers that are identical to the IoT devices they are simulating. This function is important because packet headers are one of the key items an attacker inspects to see if the device they are interacting with is a honeypot. Therefore, it is important to include this in the system outlined in this thesis.

The testing against the system is inspired by different levels of users. The most basic user accessing a honeypot is an average computer user, a Level 1 user. They might come upon the honeypot by happenstance or have malicious desires, but not much technical knowledge. These users look primarily to the HTML code and access time of the device. Therefore, tests are devised to compare actual devices to honeypots in this way for fooling basic users. The next level of user is a motivated and educated attacker, a Level 2 user. They might look a bit deeper into a honeypot and examine the packets being sent to examine the network headers. Therefore, tests are devised for comparing the headers of actual devices to honeypots to examine how effective they might be at fooling this type of attacker. Finally, Level 3 users are automated web crawlers that search for IoT devices on the internet. These tools, such as Shodan [21], run different tests and scripts against public facing Internet devices in a manner similar to Nmap. Therefore, Nmap scans are done against the actual device and the honeypot and compared to test the ability of the honeypots to appear as an actual IoT device placed on the public internet. The level of user does not even necessarily

indicate the skill level of that particular user. It is merely an indicator of exactly what a user might be examining.

3.3 IoT System Under Test

This section describes the setup and function of the IoT devices that the honeypots are based upon in this system. While some of them have wireless capability, they are all connected via wired Ethernet for the purposes of the experiments to keep testing parameters as constant as possible for testing. The camera is a TITAThink TT520PW. The thermostat is a Proliphix NT130h. The power switch is an ezOutlet2 EZ-22b. Figure 8 shows a diagram of how the devices in this experiment are connected. The three IoT devices and the Laptop containing the VMs that host Honeyd and the requesting client are all connected to the router. Figure 9 shows the IoT devices connected to the router. The items in the photo from left to right are: the camera, the router, the thermostat, and the power outlet.

3.3.1 TITAThink Camera.

TITAThink produces various types of cameras [39]. This particular model is a pinhole camera, meant to be deployed in a concealed manner; however, the software is the same for all of their cameras. TITAThink still produces this model camera in 2019 [39]. The actual camera unit is very small in size, attached via a cable to a black box that contains the main computer, power, Ethernet, and wireless antenna. The device has a capability of deploying in a wired or wireless configuration, but wired Ethernet was chosen to try and keep the network connection variables as constant as possible between this device and the honeypot. Figure 10 shows the camera connected to the network. The camera module is connected to the power and Ethernet interface that connects the device to a power outlet and the router.

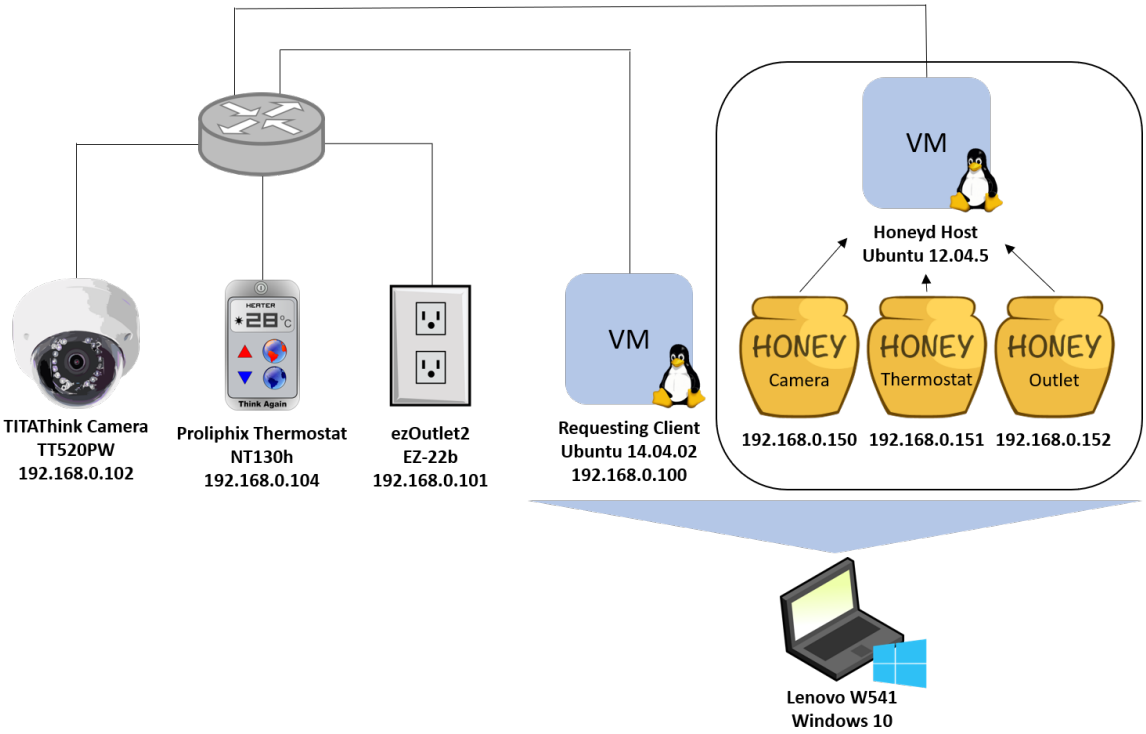


Figure 8. Model of experiment network configuration

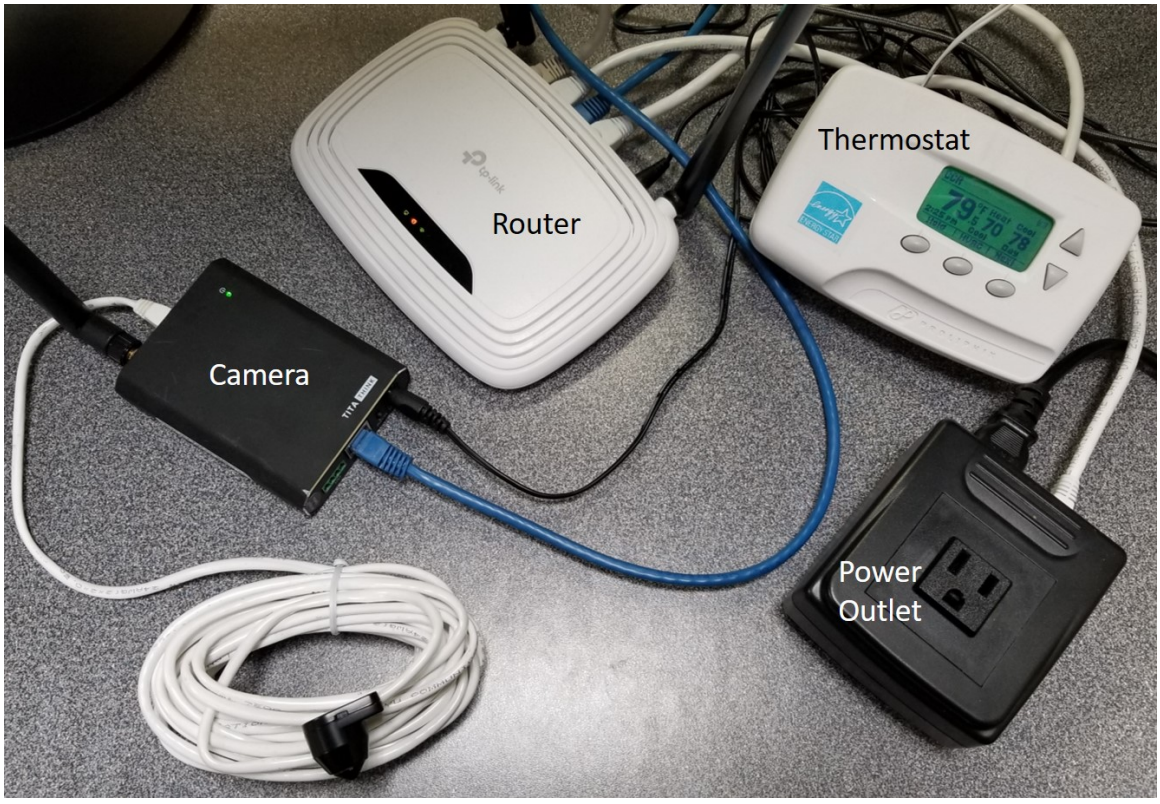


Figure 9. Network of the physical IoT devices being tested

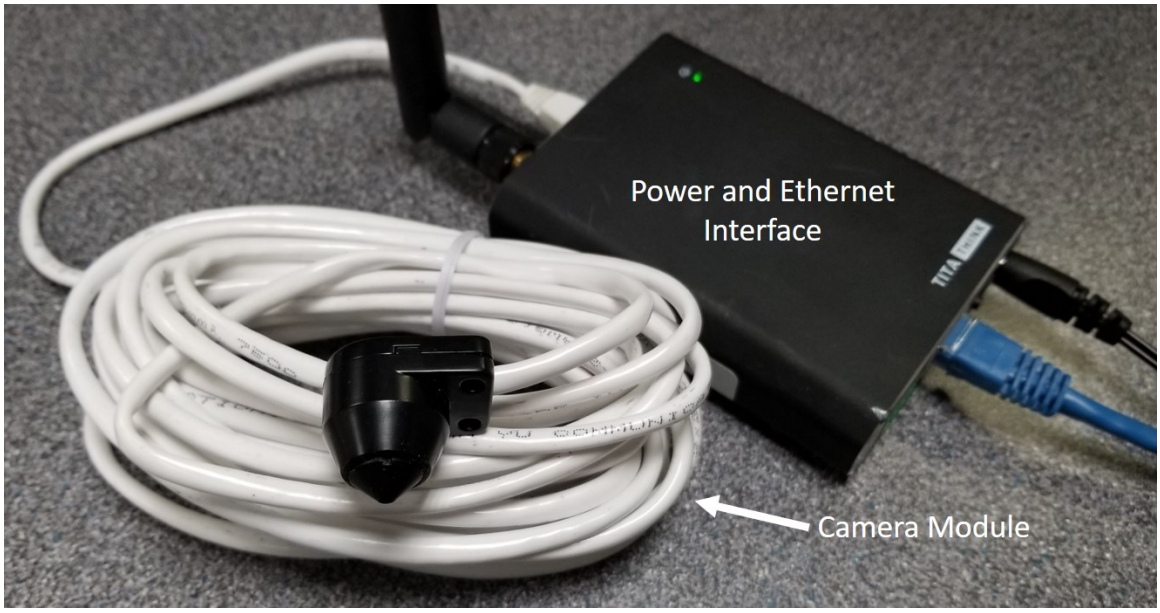


Figure 10. TITAThink TT520PW camera

Once the device is powered on and connected to the network, it can be accessed from a web browser by entering the IP address of the device. The user is first prompted with a login page (Figure 11). From here they can view the camera by clicking Enter or change the camera settings by clicking Setting. The camera can be configured to have the camera feed locked out with a password, but many IoT devices on the Internet are not configured this way. Many users do not include simple security functions like this in the name of ease of use or negligence. This device is configured without a password for the camera feed, but the settings page must be locked with a password.

Figure 11. TITAThink camera login page

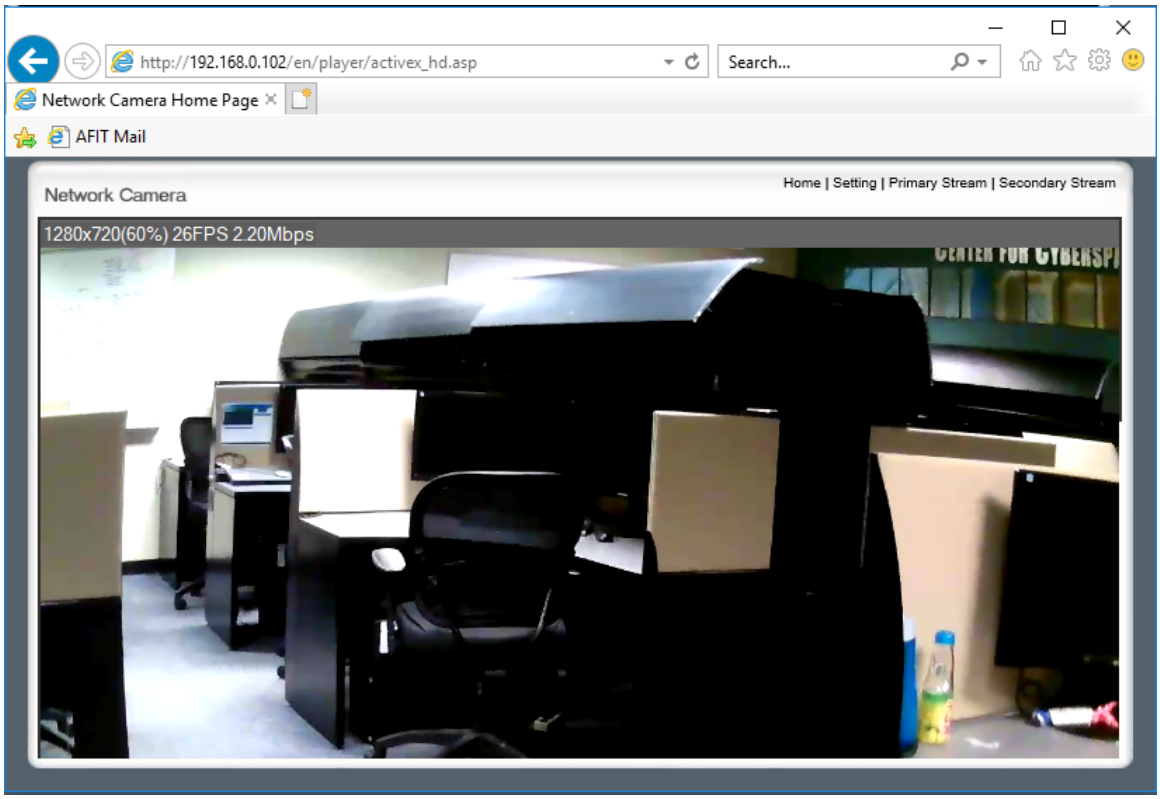


Figure 12. TITAThink camera main page

The main camera feed consists of a page with some navigation buttons at the top (Figure 12). Home takes the user back to the login page. Setting takes the user to the settings page, provided they have a password. Primary stream keeps the user on the page they are currently on while secondary stream takes the user to the same stream as on the current page, only smaller. This second stream may have to do with the ability to have multiple camera feeds from other TITAThink cameras through the same web interface. Upon examining the HTML code, most of the menu is comprised of various images. There are also some files that contain formatting information such as `top.css`. These photos and files are downloaded for the honeypot creation. The camera feed is a live view of the subject with very little delay. This could prove to be problematic for a honeypot that has no actual camera to pull a live feed from; this

is addressed in Section 3.4. If a non-authorized page or a page that does not exist is requested, the user is asked for login credentials. If they fail to provide them, they are sent to a page telling them the name of the file that they are unauthorized to see.

3.3.2 Proliphix Thermostat.

The Proliphix thermostat looks similar to many other standard non-IoT thermostats (Figure 13). It is an older device that was manufactured in 2007 [40]. It has a screen with 5 physical buttons allowing the user to interact with the device. It does not appear to be Internet enabled except upon examining the inside wiring. There are wire diagrams on how to wire the device for Ethernet. One end of an Ethernet cable needs to be cut and the wires stripped so that each wire can be placed into the six sockets as follows: (1) Blue-Blue/White, (2) Brown-Brown/White, (3) Orange/White, (4) Orange, (5) Green/White, and (6) Green (Figures 14 and 15). Once the device is wired and powered on, it gains an IP address through Dynamic Host Configuration Protocol (DHCP) [40]. This can be viewed in the Network Status Screen. One of the problems with implementing this device in a test environment is that thermostats are usually wired on a wall with leads that provide it power as well as the ability to turn the heating and HVAC systems on and off. This was remedied with a 24V AC power adapter. It had two leads, one placed in socket RC and the other in socket C on the thermostat (Figures 14 and 15). This made it possible to provide the thermostat with power from an AC wall outlet for the test environment.

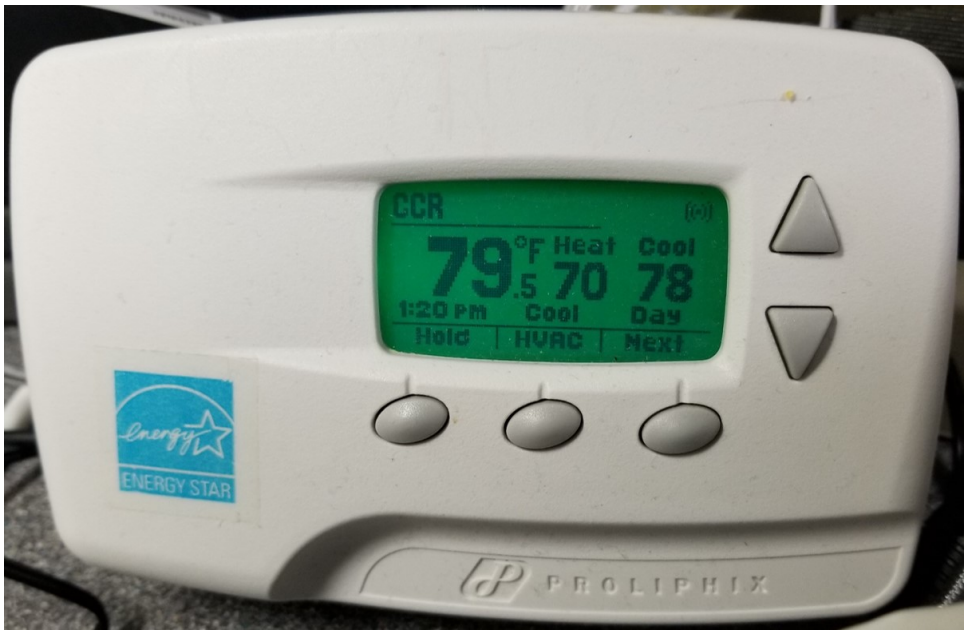


Figure 13. Proliphix NT130h thermostat

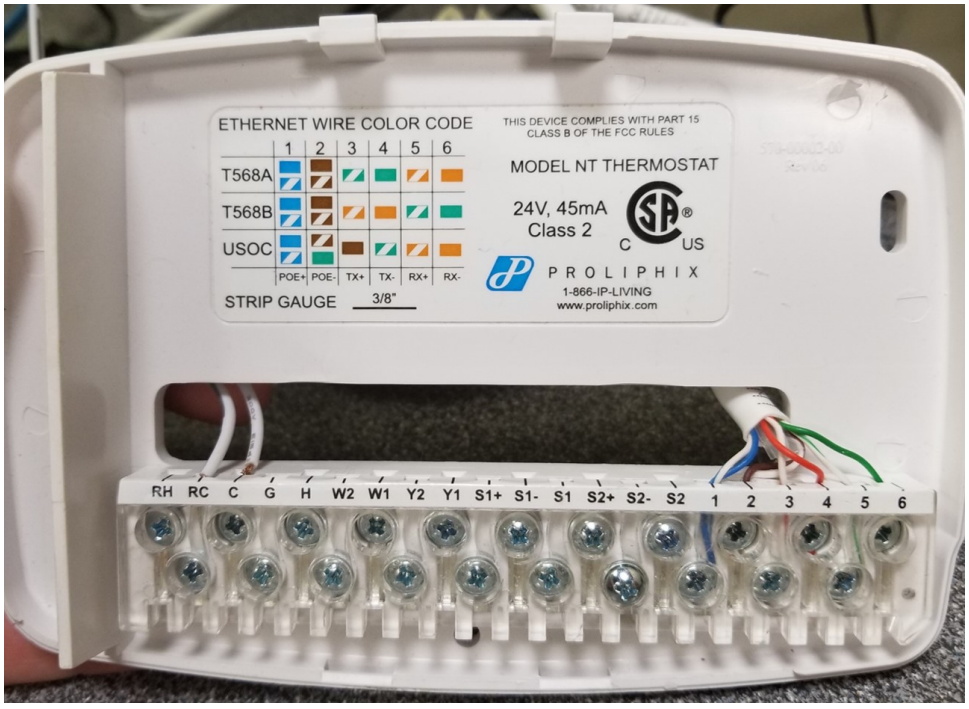


Figure 14. Wiring of power and Ethernet for Proliphix thermostat

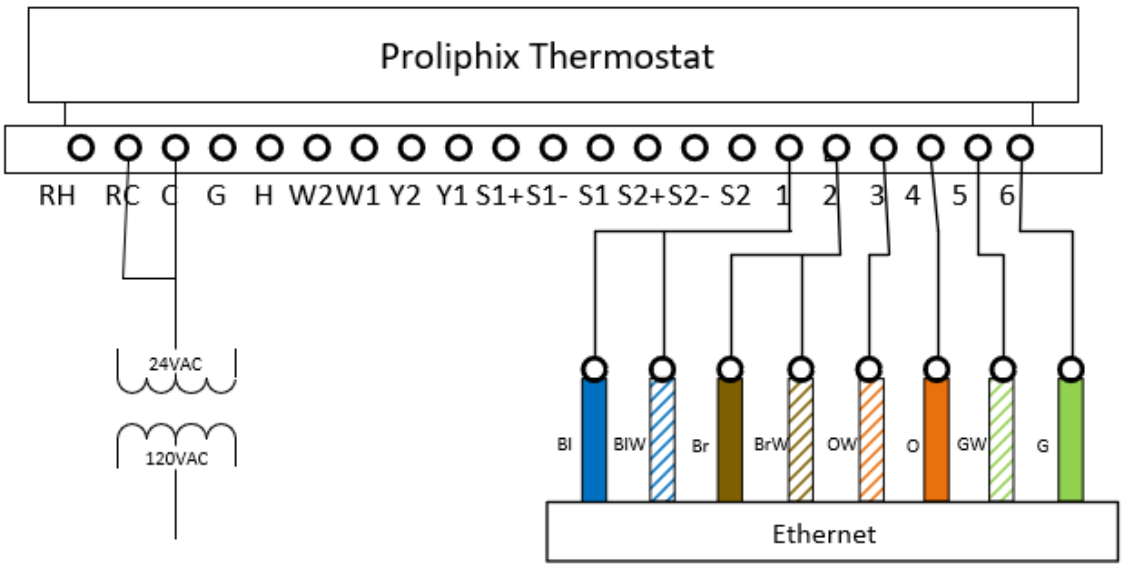


Figure 15. Wiring diagram of Proliphix thermostat

Once the thermostat had an IP address, it could be viewed in the web browser with no password. As shown in Figure 16, the main page provides all of the temperature and time information about the thermostat. All of the data that is displayed on the page is written directly in the HTML. There is not a references in the HTML that calls a script to obtain the time, date, or temperature from the server. Only two buttons are available to the user for navigation. Status just refreshes the page, allowing the user to see the same information as on the homepage. Login allows the user to view the device settings after entering the correct password. The settings page has to be kept behind a password, which is fine for the purposes of this experiment. Trying to navigate to any other page, even nonexistent ones, requires authentication from the user.

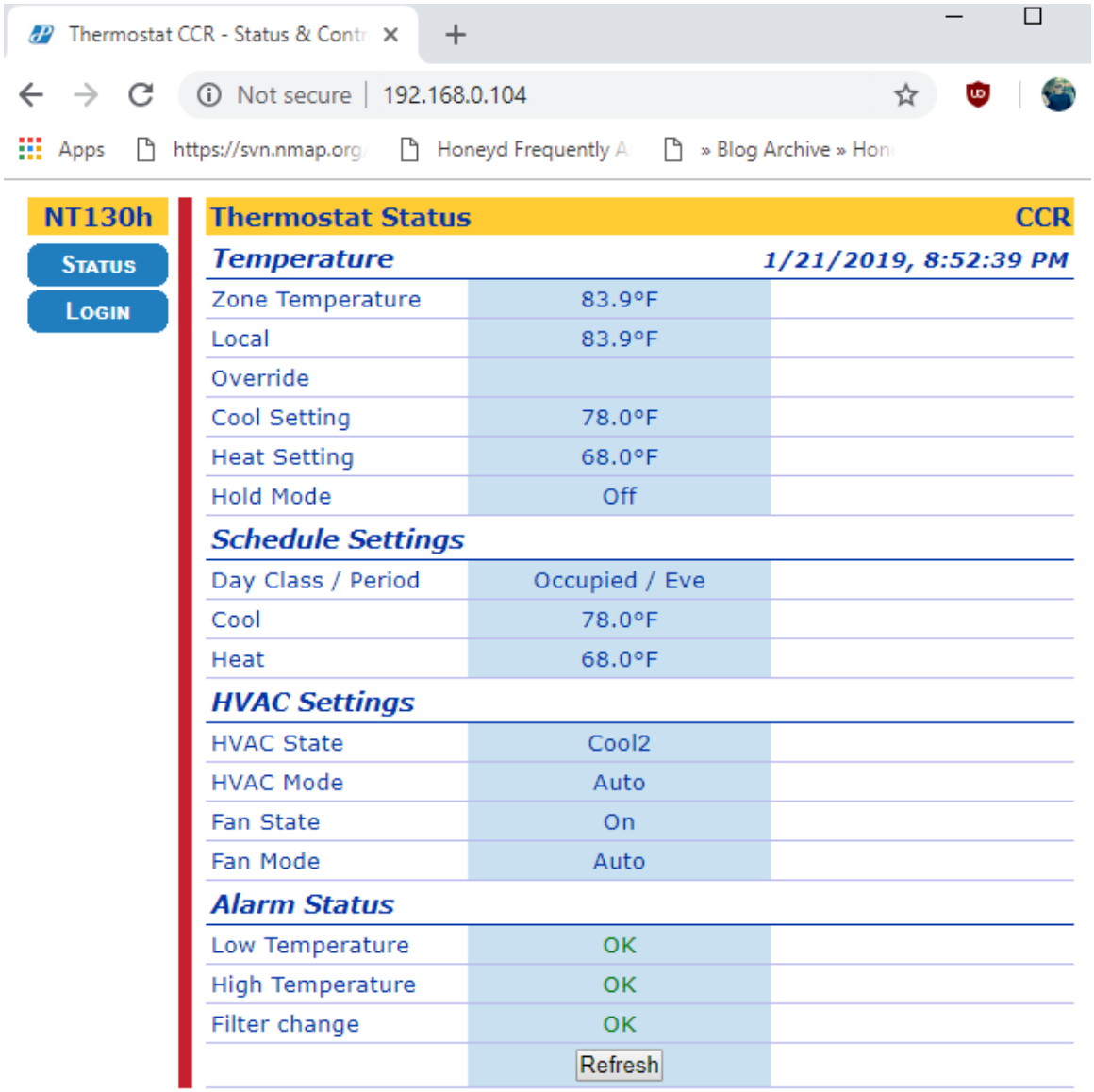


Figure 16. Proliphix thermostat main page

As with many IoT devices, this thermostat has various sensors and displays information to the user through the web interface. As shown in Figure 16 the main information shown is the name of the device, the current temperature in the room, the status of the HVAC (Heat/Cool), and schedule settings. This data changes based on the thermostat settings as well as the outside stimuli, such as temperature, that

the thermostat observes. All of the items are static on the page, but update upon refreshing the page. This is perfect functionality for a honeypot.

3.3.3 ezOutlet2 Power Switch.

The ezOutlet2 is a small box with a single power outlet as well as a slot for a power cable and Ethernet port (Figures 17 and 18). This device is still being produced by Mega System Technologies in 2019 [41]. The power cable port provides pass through for the main power outlet, and the main power outlet can be turned on and off via software. The device is not wireless and must be wired into the network with an Ethernet cable. The device is assigned an IP address by DHCP and then a user is able to navigate to a web interface through the assigned IP address.

As shown in Figure 19, the web interface provides a split web page, the left page consists of a navigation menu while the right side displays status information. The pages include: Status, Network, Settings, Schedule, Ping Address, and Save/Restore. The Status page provides the majority of the information on the device including: device name, network information, date and time, firmware version, and the status of the power outlet (On/Off). The Network page provides the same network information as the Status page but allows the user to change the Domain Name System (DNS) servers, hostname, and other information. The Settings page provides the user the ability to change the username and password as well as the outlet power cycling settings. The Schedule page allows the user to cycle power to the outlet at certain times of the day and week. The Ping Address page allows the device to ping an IP address, to confirm it is on the network or connected to the Internet. The Save/Restore page allows the user to reset the device and save the settings.

Figure 17. ezOutlet2 EZ-22b power outlet

Figure 18. ezOutlet2 EZ-22b Ethernet port

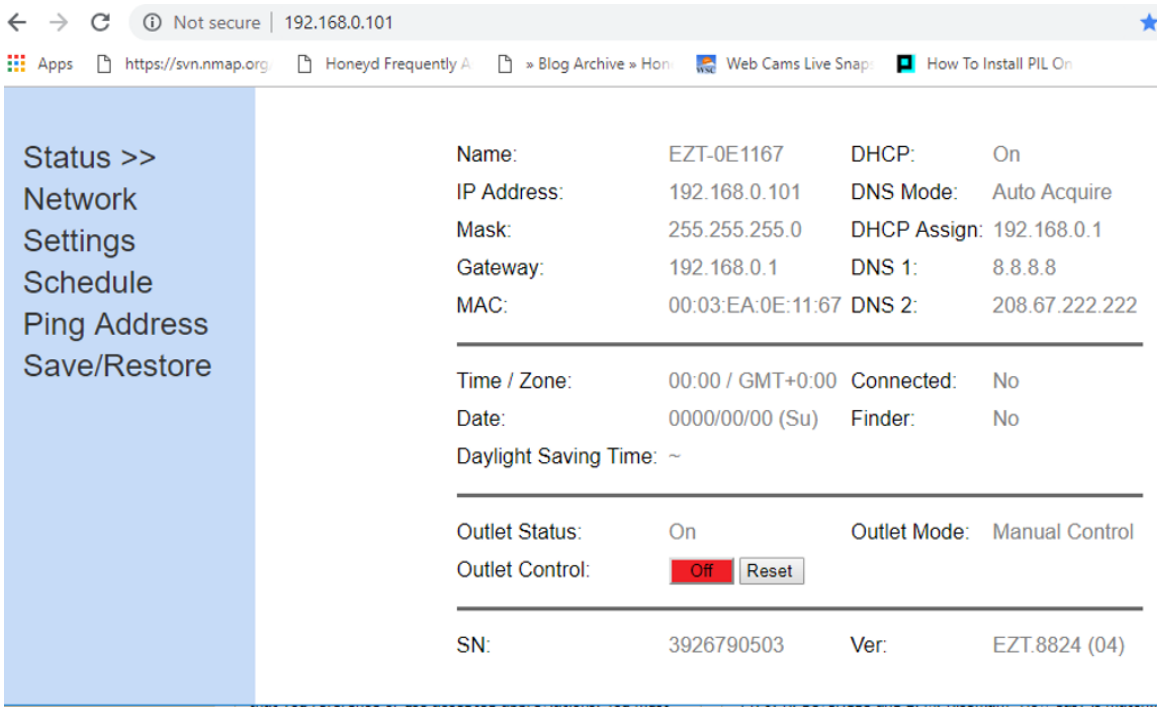


Figure 19. ezOutlet2 EZ-22b power outlet main page

The main function of this device is the ability to turn the power outlet on and off. Whenever the 'Outlet Control' button is clicked on the Status page, the button changes color and the state of the outlet changes. Upon examining the HTML code, this is done through a script, `invert.cgi`. For a honeypot, this is the main functionality that should be simulated. Since the main function of an IoT power outlet is the ability to turn the switch on and off, an attacker would likely be looking for this function to be properly working when examining this device.

3.4 Honeyd IoT System Framework

This section describes the scripts and files required for the functioning of the IoT honeypots in Honeyd. This system is made up of three honeypots. All of these honeypots are modeled after the physical devices that are highlighted in Section 3.3.

Honeyd is run on a virtual machine running Ubuntu 12.04.5; this configuration runs all the honeypots since Honeyd can control many virtual honeypots. The Ubuntu 12 VM is given 1 processor core and 4 GB of RAM. The host workstation for the VMWare Workstation 14 Pro virtual machines is a Lenovo W541 laptop with 32GB or RAM and an Intel i7 processor clocked at 2.9 GHz running Windows 10. The laptop also has Wireshark 2.6.3 running on it for scanning the IoT devices to construct the honeypots.

3.4.1 Honeyd Configuration.

As previously discussed in Chapter II, Honeyd has a singular configuration file containing all parameters affecting the honeypots function. The system file, `iotHoneyd.conf` (Appendix A), contains configuration information for the three honeypots: camera, thermostat, and power switch. Honeyd is started with the command in the file `startHoney.sh` (Appendix A). The similarity between all of these devices is that they have TCP port 80 available, which is the main interface for interacting with the device.

```
create titacamera
#set titacamera personality "Linux Kernel 2.4.18 - 2.5.70 (X86)"
set titacamera personality "Linux 2.3.28-33"
set titacamera default tcp action reset
add titacamera tcp port 80 "TitaCamera/camera_web.sh"
add titacamera tcp port 554 open
add titacamera tcp port 49152 open
add titacamera udp port 443 filtered
add titacamera udp port 990 filtered
add titacamera udp port 1900 filtered
add titacamera udp port 1901 filtered
add titacamera udp port 3702 open
add titacamera udp port 16896 filtered
add titacamera udp port 18676 filtered
add titacamera udp port 19956 filtered
add titacamera udp port 22986 filtered
add titacamera udp port 30697 filtered
add titacamera udp port 32772 filtered
add titacamera udp port 32777 filtered

create proliphixthermostat
set proliphixthermostat personality "D-Link Print Server"
set proliphixthermostat default tcp action reset
#Regular Thermostat Interface when Internet is available
#add proliphixthermostat tcp port 80 "ProliphixThermostat/thermostat_web.sh"
#Thermostat Interface when no internet is available
add proliphixthermostat tcp port 80 "ProliphixThermostat/thermostat_web_nointernet.sh"

create ezoutlet
set ezoutlet personality "IBM OS/2 Warp 4.0"
set ezoutlet default tcp action reset
add ezoutlet tcp port 80 "ezOutlet/outlet_web.sh"
```

Figure 20. Code excerpt from `iotHoneyd.conf` (Appendix A) showing how each device has a script run on TCP port 80

As shown in Figure 20, each device has TCP port 80 listening with its own shell script governing its function. Whenever something tries to make a TCP connection, the shell script begins executing and provides the HTTP header along with other scripts that alter the data of the device and then send it out. These shell scripts are highlighted again later in this section. Also in the configuration file is the assignment of IP and MAC addresses. Honeyd is able to assign the IP address dynamically through DHCP, but this is commented out to allow manual IP address assignment. The MAC addresses are assigned to be exactly the same as the physical device that the honeypot is simulating.

3.4.2 Web Server.

Traditionally, IoT devices that utilize HTTP have a file structure to provide different aspects of the web interface. This can be JavaScript code, images, or other applications. Honeyd is extremely capable at simulating network traffic for any of the honeypots it creates, regardless of the service. It is able to gather the incoming TCP packets and parse through the files requested by the client. Then the shell script can provide the various files to the client as requested. In this way, it acts as a pseudo web server. The code for the bash scripts that act as a web server comes from one of the built-in Honeyd scripts `iis.sh` written by Niels Provos. It has, however, been edited extensively to serve different pages with varying HTTP response messages as well as the ability to log additional data from an attacker.

Each honeypot shell script starts with a loop that is listening for requests from Honeyd. Then, the `grep` command is used to isolate the important part of the message, the HTTP GET request. Each HTTP GET request is stored in a file `requests.txt` to see what kind of requests an attacker might be making. An attacker may be trying to exploit the device with a certain request, and the honeypot captures it.

The format of an HTTP GET request, highlighted in Figure 21, shows the file requested by the client (/doc/test.html). This is extracted and then through a series of conditional statements, the correct file is printed to standard output along with the correct HTTP header. It begins with the correct HTTP code, usually code 200 OK. The web page is usually printed with the `cat` command or with one of the `htmlprint.py` Python scripts used by each honeypot (Appendix B, C, & D). If an individual line needs to be updated, such as the correct date or time, a Python program can print the page line by line, inserting the updated line as necessary. These methods are used for printing web pages because the IoT devices have Unix-based

operating systems where each line of code ends with a carriage return and line feed. Making changes to the web page and saving it would remove the carriage returns. If this was done, the honeypot data would not appear as authentic when compared directly to the actual device. Honeyd takes this printed data and sends it as a TCP response packet to the client. Pictures can also be sent using the `cat` command.

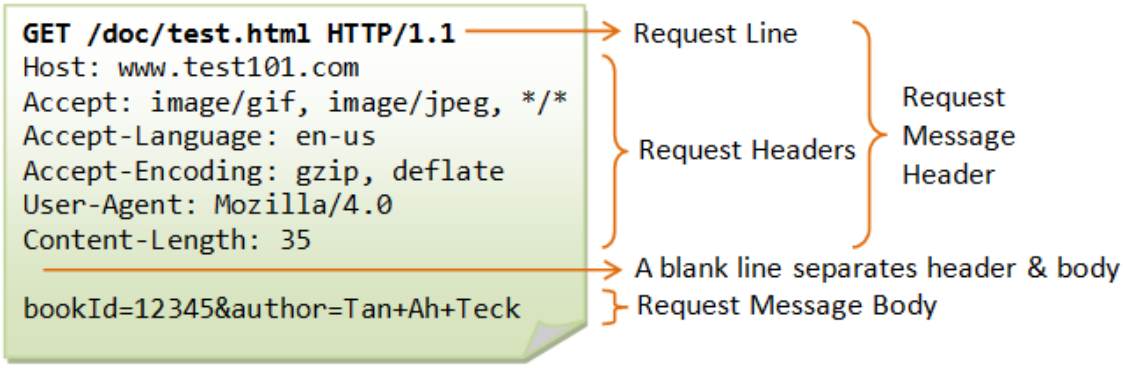


Figure 21. Example HTTP GET request [8]

3.4.2.1 TITAThink Camera Honeypot.

Examination of the TITAThink camera device showed a series of images that make up the menu as well as a flash application that shows the camera feed. These files were extracted by saving the web page of the original device. The flash application in the original device has a live camera feed that is presented to the user. However, there is no way to incorporate this system with Honeyd. Which is acceptable since the honeypot does not have an actual camera from which to pull live footage.

To counter this, a Python script, `camera.py` (Appendix B), was written to try and mimic the appearance of a live camera feed through updating pictures. One part of this script takes the current time and then compare that time to a folder of images of the same location at different times of day called "cameraFeeds". Each photo's file name indicates which time of day the photo was taken. The script looks at which

photo is closest to the current time of day and copies it to the image in the main directory that is sent by Honeyd. Currently, the system has 24 photos, one for each hour of the day; however, the system is capable of an unlimited number of photos. In theory, someone who implements this system could have an image for every minute of the day, or more frequent. The more images there are, the more convincing the camera would appear to be.

The TITAThink camera has no date and time overlay on the actual camera feedas found in many other IP cameras. This function is added in the `camera.py` script and disabled by default. This code could, however, be adapted to another IoT Camera honeypot that does require this functionality. In Figure 22, camera feed A is the template photo and camera feed B contains the date, time, and the location name.

Figure 22. The Python script has the capability to draw date and time on the camera image

The bash script associated with the camera honeypot, `camera_web.sh` (Appendix B) provides all the necessary files needed to run the honeypot. Wireshark is used to view the packet responses when accessing the TITAThink camera. One interesting issue is that when the root page of the camera is accessed, three redirections take place before taking the user to the login page. Figure 23 shows the Wireshark scan highlighting the redirects. The scan shows that upon accessing the camera, there is an HTTP code 302 Redirect, then a regular HTML page with HTTP code 200

OK that redirects again, then another HTTP code 302 Redirect to the main login page. A portion of how the bash script accounts for this interaction through a list of conditionals is highlighted in Figure 24. The `elif` statements are looking for requests to particular pages. If `/default.asp` is requested, it sends the HTTP 200 OK code and prints the `redirect.html` file using `htmlprint.py` (Appendix B). This file redirects to `/form/default`. When the conditionals see a request for this file, the second HTTP code 302 Redirect is sent to redirect to `/en/login.asp`. The rest of the script consists of different conditionals for the various other pages and images on the camera. Another item that is included in the HTTP headers is the current date and time. To include this in the HTTP headers sent to the user through `camera_web.sh`, the script `timedate.py` (Appendix B) is run every time the bash script sends a response to ensure the most current date and time is included in the proper format. Figure 25 shows the home page of the camera honeypot when it is fully configured and running.

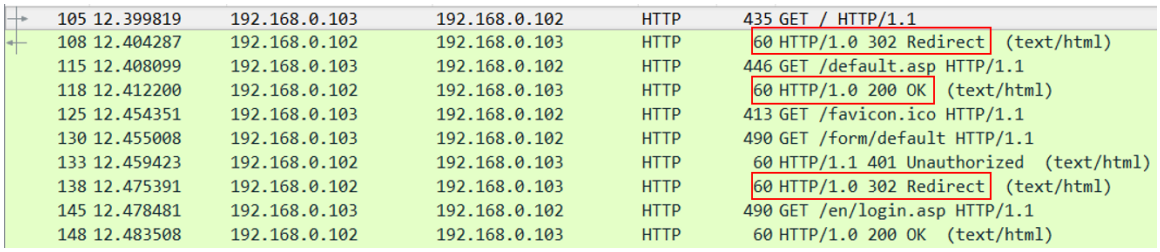


Figure 23. Wireshark capture of the TITAThink camera showing the redirects that happen when trying to access the main page

```
_   _
        elif [ '/default.asp' = "$y" ] ; then
                REQUEST=$NEWREQUEST
                DATE=$(python timedate.py)
                cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
python htmlprint.py redirect.html
        elif [ '/form/default' = "$y" ] ; then
                REQUEST=$NEWREQUEST
                DATE=$(python timedate.py)
                cat << _eof_
HTTP/1.0 302 Redirect
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache
Location: http://192.168.0.150/en/login.asp

<html><head></head><body>
                This document has moved to a new <a href="http://192.168.0.150/en/
login.asp">location</a>.
                Please update your documents to reflect the new location.
</body></html>
```

Figure 24. Conditional statements showing how headers and files are sent based on the file requested

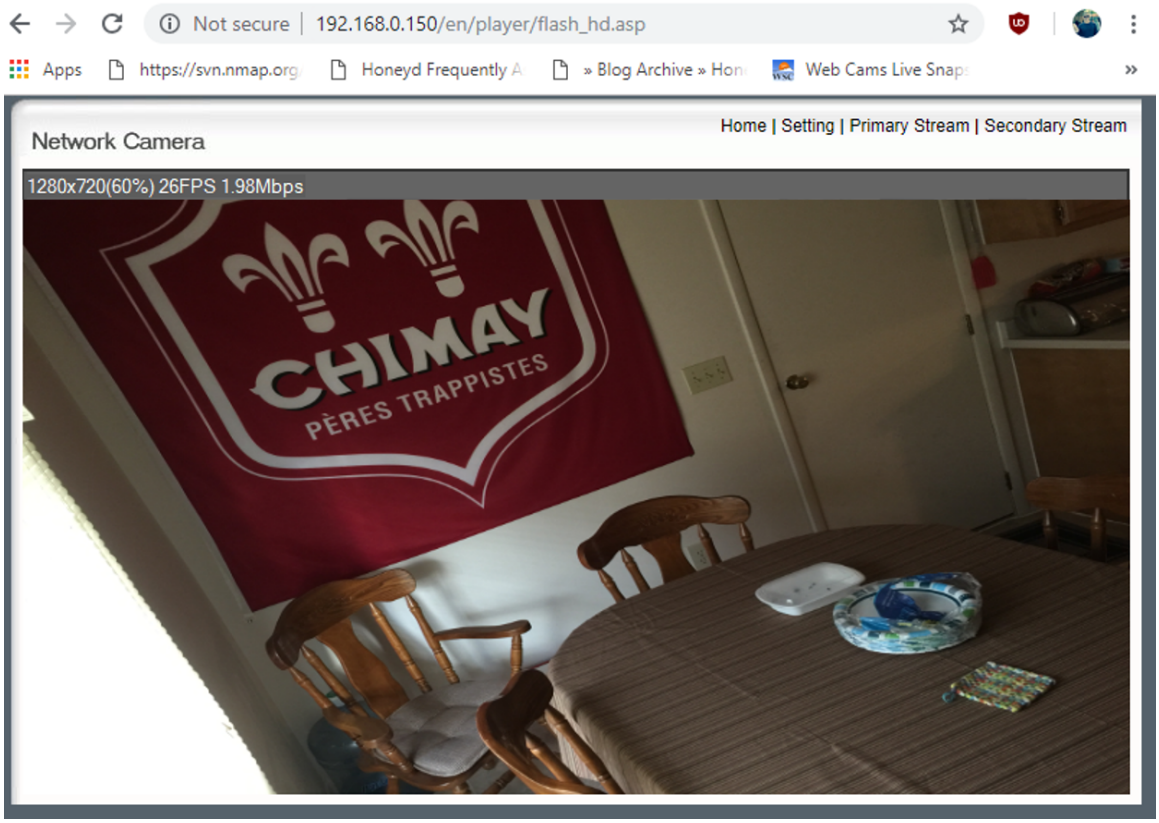


Figure 25. Homepage of Honeyd camera honeypot when it is fully configured

Whenever a page that does not exist is requested, the Python script `error.py` (Appendix B) is used in conjunction with the bash script to create the error HTML page.

Another aspect of the TITAThink camera is that the camera feed can be viewed without a password, but the settings page requires a password to be viewed. From the Wireshark capture in Figure 26, HTTP code 401 Unauthorized is sent to the client. According to the HTTP status code registry, code 401 means the client's web browser asks the user for a username and password and then sends that information to the requesting agent [42]. Since this device only uses HTTP and not HTTPS, the data sent is not encrypted. The username and password are sent in plain text encoded with base64. The `camera_web.sh` script sends this HTTP code if the requested page

requires authorization; it also uses the `grep` command to search for 'Authorization:', which is the HTTP header field where the username and password are stored, from incoming packets. If it finds that field, it base64 decodes it, and stores it in the file `password.txt`. This gives the honeypot the capability to store usernames and passwords to see what kind of credentials attackers are using. All file requests made by an attacker are stored in a similar file `requests.txt`.

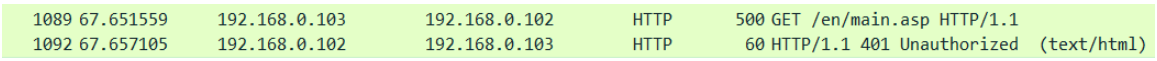


Figure 26. Wireshark capture of the TITAThink Camera showing access to the settings page is unauthorized

Honeyd also requires a personality to mimic the Nmap fingerprint of the device for OS detection. Unfortunately, the version of Honeyd used in this research does not have the updated Nmap fingerprint database used in the latest version of Nmap. Trying to update the Nmap database within Honeyd leads to Honeyd breaking. Further research and code edits may need to be done to Honeyd itself to accommodate this. Using Nmap to scan the original device (Figure 27), the OS appears to be Linux 2.6.x or between Linux 2.6.9 and 2.6.33; however, the Nmap database used by this version of Honeyd does not have a fingerprint for this version of Linux. To be as close as possible, the OS personality chosen was Linux kernel 2.4.18 - 2.5.70. This does not make the honeypot appear identical to the actual device, but the OS fingerprint is similar and may not be noticed by a level 2 or 3 user. They may attribute the difference to just a different firmware on the device.

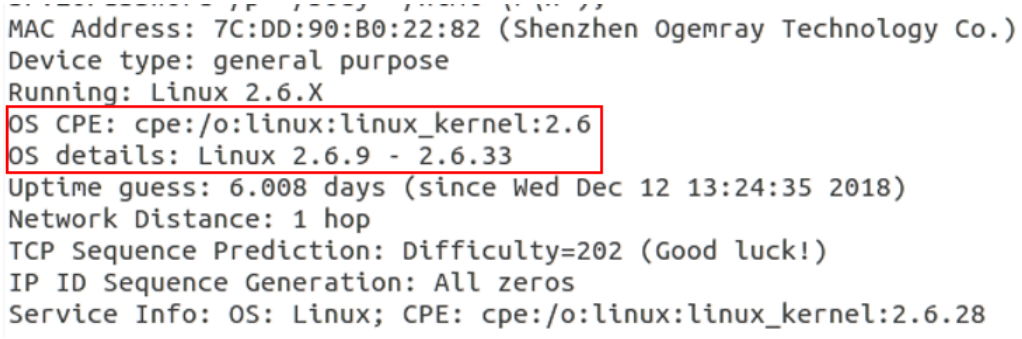


Figure 27. Detected OS of a TCP SYN scan on the TITAThink camera

3.4.3 Proliphix Thermostat Honeypot.

Upon examination of the thermostat, there are fewer pages to navigate than the camera. The main page has a listing of temperatures, heat and cold settings, as well as the date and time. All of the information is listed in the HTML code and can be easily passed to a client by Honeyd. There are no updating images as in the TITAThink camera, so the honeypot should be able to look very authentic.

The Python script governing the function of this application is `thermostat.py` (Appendix C). First, this script gets the current date and time and puts it in the correct format to write to the file. It uses the Python weather-api to get the current temperature in the area specified in the program. It then determines whether or not this temperature is hot or cold, and turns the heat or cool setting on accordingly. If the current temperature is greater than 70 degrees, the cool setting is turned on and set to 73 and the heat setting is set to 65. If the current temperature is less 70 degrees, then the heat setting is turned on and set to 68 and the cool setting is set to 75. All of these settings can be altered within `thermostat.py`.

Test queries were made against both the thermostat and completed honeypot to compare query response times. The Proliphix thermostat was approximately 1.1 seconds slower, so the `thermostat.py` script sleeps for 1.1 seconds before sending out

the file. The script goes line by line printing the template file while keeping track of the line number of the file. The line numbers of lines that contain the date, time, and temperature are predetermined ahead of time. If the script finds one of these lines, instead of writing the line from the template file, it writes an altered line with the correct information. As stated previously, this is done instead of just creating a new file with the correct information because the file sent by the IoT device has carriage return and line feeds after the lines in the file. If a new file was created, the carriage returns would disappear and decrease authenticity.

The bash script, `thermostat_web.sh` (Appendix C), functions almost identically to the script for the TITAThink camera. Wireshark was used to capture the packets sent by the device when trying to navigate to the various pages (Figure 28). The script stores all requests from a client to see what kind of files are requested. There is not a series of redirects like in the TITAThink camera, so the conditionals can be straightforward. If a file is requested by the client, the server gives it a file in response, as seen in line 11 of the Wireshark capture. One noticeable similarity is when the client tries to access the login page, they are prompted with HTTP code 401 Unauthorized, as shown in line 383 of the Wireshark capture. The web browser prompts a user for a username and password in this case, and their response is stored in a file, `password.txt`. Trying to access a file that does not exist always results in a request for a username and password. All file requests made by the user are stored in the file, `requests.txt`.

9	2.988118	192.168.0.103	192.168.0.104	HTTP	435	GET / HTTP/1.1
11	3.045400	192.168.0.104	192.168.0.103	HTTP	137	HTTP/1.1 200 OK
328	12.427587	192.168.0.104	192.168.0.103	HTTP	223	HTTP/1.1 401 Authorization Required
338	13.472248	192.168.0.103	192.168.0.104	HTTP	413	GET /favicon.ico HTTP/1.1
339	13.519178	192.168.0.104	192.168.0.103	HTTP	223	HTTP/1.1 401 Authorization Required
382	17.731372	192.168.0.103	192.168.0.104	HTTP	445	GET /etc/passwd HTTP/1.1
383	17.778458	192.168.0.104	192.168.0.103	HTTP	223	HTTP/1.1 401 Authorization Required
394	19.194071	192.168.0.103	192.168.0.104	HTTP	412	GET /favicon.ico HTTP/1.1
395	19.241029	192.168.0.104	192.168.0.103	HTTP	223	HTTP/1.1 401 Authorization Required

Figure 28. Wireshark capture of the Proliphix thermostat show accessing the main page as well as an unauthorized page

Since the Python script, `thermostat.py`, requires the use of the Internet to get accurate weather data, the deployed honeypot would require Internet connectivity. Depending on the deployment conditions, this might not be possible. To remedy this situation, the Python script `thermostat_nointernet.py` and bash script `thermostat_web_nointernet.sh` (Appendix C) were created. This Python script removes the Python weather-api call that requires Internet access and instead has a variable with the current temperature that can be altered by the user. The bash script is identical to `thermostat_web.sh` but replaces the calls to `thermostat.py` with `thermostat_nointernet.py`. Figure 29 shows the homepage of the Honeyd thermostat honeypot when it is running and fully configured.

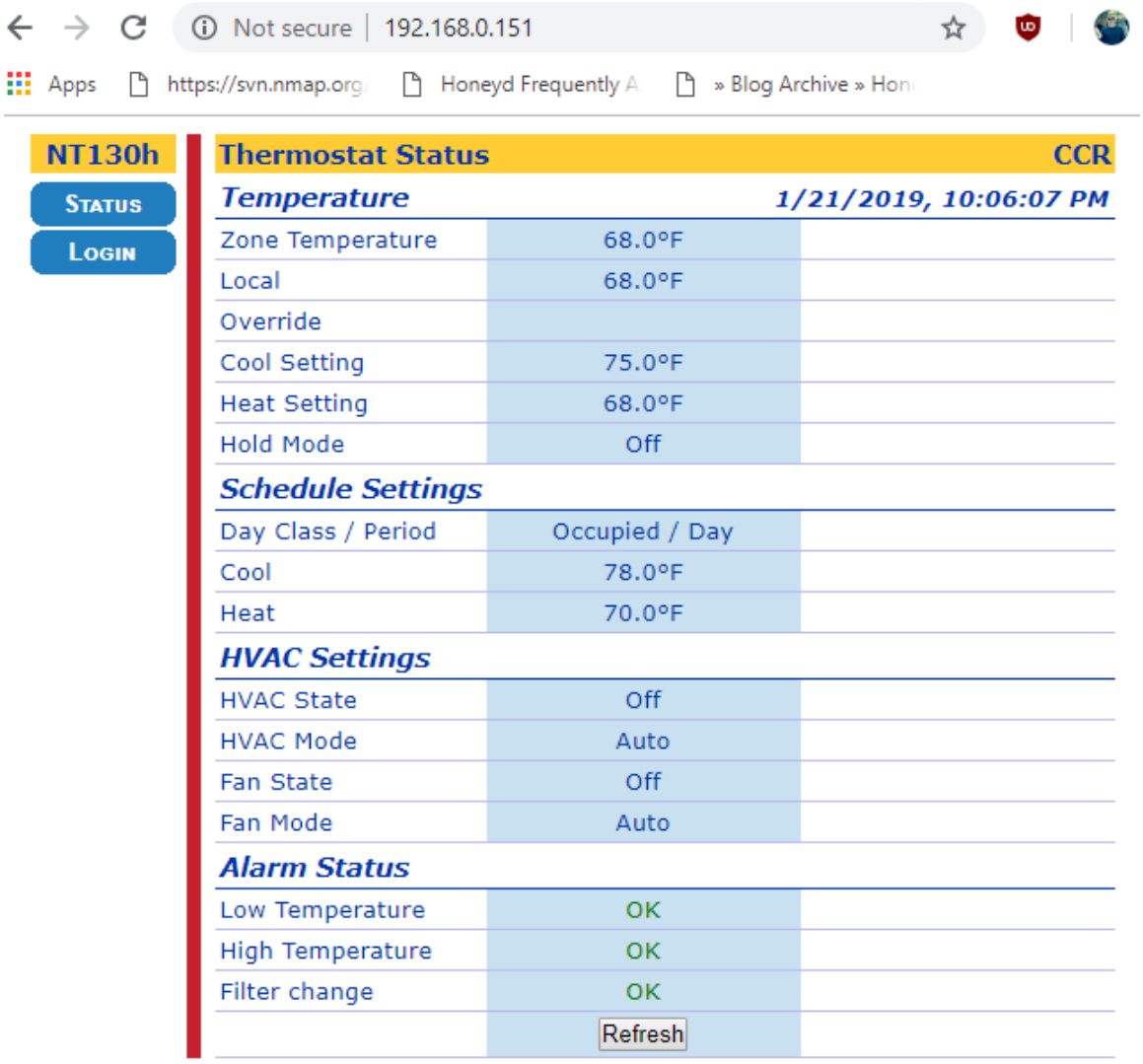


Figure 29. Homepage of Honeyd thermostat honeypot when it is fully configured

The Nmap scan on the Proliphix thermostat (Figure 30) shows that the detected OS was not incredibly accurate since only one service is available, HTTP. The best guess was some sort of D-Link print server, with various model names. On Honeyd's older version of the Nmap fingerprint database, only one generic D-Link Print Server was available as a personality. This one was chosen because it was likely the closest to what Nmap saw on the actual device.

```
MAC Address: 00:11:49:00:62:46 (Proliphix)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1
closed port
Device type: print server|WAP
Running: D-Link embedded
OS CPE: cpe:/h:dlink:dpr-1260 cpe:/h:dlink:dgl-4300 cpe:/h:dlink:dgl-4500 cpe:/h:dlink:dir-615
cpe:/h:dlink:dir-625 cpe:/h:dlink:dir-628 cpe:/h:dlink:dir-655 cpe:/h:dlink:dir-855
OS details: D-Link DPR-1260 print server; or DGL-4300, DGL-4500, DIR-615, DIR-625, DIR-628,
DIR-655, or DIR-855 WAP
Network Distance: 1 hop
TCP Sequence Prediction: Difficulty=166 (Good luck!)
IP ID Sequence Generation: Incremental
```

Figure 30. Detected OS of a TCP SYN scan on the Proliphix thermostat

3.4.4 ezOutlet2 Honeypot.

The ezOutlet2 starts with a split page where one half always contains a list of pages to navigate to and the other half shows the page that has been selected from the menu (Figure 31). The HTML code from the actual device is a very short HTML page that references two other HTML pages for each side. The page lists whether or not the power switch is in the on/off position, scheduling, date/time, and the IP address of the ezOutlet2 honeypot.

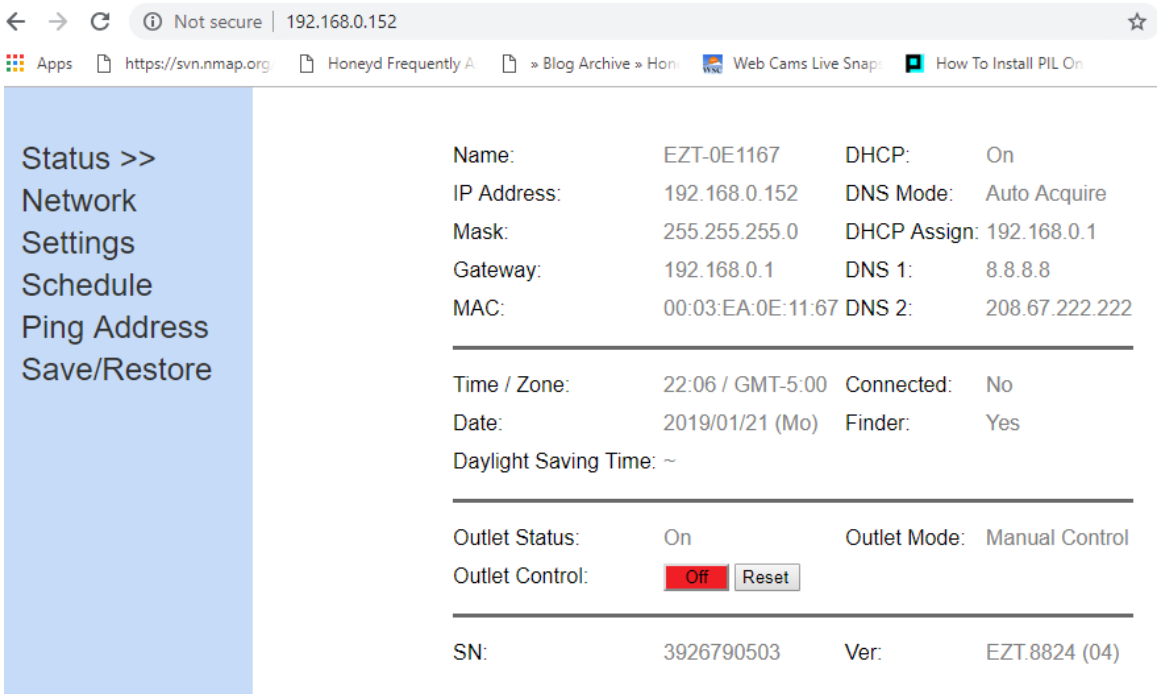


Figure 31. Homepage of Honeyd outlet honeypot when it is fully configured

The bash script, `outlet_web.sh` (Appendix D), functions in the same way as the camera and thermostat honeypot scripts. The Wireshark scan of the outlet (Figure 32) shows the main flow of packets upon accessing everything on the device. There are no redirects so that when an existing page is requested, HTTP code 200 is sent along with the page data as shown in line 2 of the Wireshark trace, as shown in line 27 of the Wireshark capture. Disabling the username and password unlocks the ability to see all pages on the ezOutlet2. If a file does not exist, HTTP code 404 Not Found is sent to the client as shown in line 305 of the Wireshark capture. Since there is no authorization, no passwords are stored, but every file request is still stored in `requests.txt`.

24	9.120349	192.168.0.103	192.168.0.101	HTTP	435 GET / HTTP/1.1
27	9.121878	192.168.0.101	192.168.0.103	HTTP	651 HTTP/1.1 200 OK (text/html)
34	9.134781	192.168.0.103	192.168.0.101	HTTP	362 GET /mchp.js HTTP/1.1
37	9.136573	192.168.0.101	192.168.0.103	HTTP	1175 HTTP/1.1 200 OK
38	9.137068	192.168.0.101	192.168.0.103	HTTP	1132 Continuation
40	9.138733	192.168.0.101	192.168.0.103	HTTP	1185 Continuation
41	9.138735	192.168.0.101	192.168.0.103	HTTP	117 Continuation
51	9.160707	192.168.0.103	192.168.0.101	HTTP	475 GET /menu.htm HTTP/1.1
52	9.160774	192.168.0.103	192.168.0.101	HTTP	477 GET /status.htm HTTP/1.1
59	9.165484	192.168.0.101	192.168.0.103	HTTP	191 HTTP/1.1 200 OK (text/html)
66	9.170383	192.168.0.103	192.168.0.101	HTTP	372 GET /mchp.js HTTP/1.1
69	9.172209	192.168.0.101	192.168.0.103	HTTP	1175 HTTP/1.1 200 OK
302	28.074993	192.168.0.103	192.168.0.101	HTTP	445 GET /etc/passwd HTTP/1.1
305	28.076319	192.168.0.101	192.168.0.103	HTTP	120 HTTP/1.1 404 Not found

Figure 32. Wireshark capture of the ezOutlet2 shown accessing the main page as well a nonexistent page

On the ezOutlet2 device, the power switch turns on and off through a simple script `invert.cgi`. The file, `invert.cgi`, can have two states, 0,0 and 1,0, on and off. Every time `invert.cgi` is requested by the client, the state switches. One of the conditionals in the `outlet_web.sh` script on the honeypot mimics this behavior (Figure 33). When the client requests `invert.cgi`, the HTTP code 200 OK is sent with the proper headers. Then the current state of `invert.cgi` is read, if it is 0,0, then 1,0 is written to `invert.cgi`. If it is 1,0, then 0,0 is written. Then the new updated version of `invert.cgi` is sent to the client. In the ezOutlet2 device, `reset.cgi` returns the power outlet to its default state, 0,0. Another conditional in the `outlet_web.sh` script mimics this functionality. When `reset.cgi` is requested by the client, HTTP code 200 OK is sent and `invert.cgi` is set to its default state of 0,0. Then `invert.cgi` is sent to the client.

```
        elif [ '/invert.cgi' = "$y" ] ; then
                REQUEST=$NEWREQUEST
                cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
state=`cat invert.cgi`
                if [ '0,0' = "$state" ] ; then
                        echo "1,0" > invert.cgi
                else
                        echo "0,0" > invert.cgi
                fi
cat invert.cgi
        elif [ '/reset.cgi' = "$y" ] ; then
                REQUEST=$NEWREQUEST
                cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
echo "0,0" > invert.cgi
```

Figure 33. Script for flipping and resetting the switch on the honeypot outlet

As with the other honeypots, a Python script called `outlet.py` (Appendix D) manages updates to the web pages of changing items such as the date and time and the IP addresses for the device and gateway. This script has to make changes to two pages, `status.htm` and `network.htm`. The status page has the IP address of the device, subnet mask, gateway IP, and the date and time. The network page has an IP address, subnet mask, and gateway IP that needs to be updated. The honeypot deployer can assign all of these values by changing the global variables gatewayIP, honeypotIP, and subnetMask within `outlet.py`. This script is run from the bash script whenever either of those pages are requested by the client and updates with the latest time when refreshing the page.

Similarly to the TITAThink camera, the ezOutlet2 has an error page which is

presented to the user through the use of a Python script, `error.py` (Appendix D). It prints the HTML for the error page, which is a short one line page, but with only a carriage-return at the end of the file, no line feed. For this reason, a Python script is required.

An Nmap scan on the outlet did not provide an exact OS match. The closest guess it could provide was IBM OS/2 Warp 2.0 with a certainty of 86% (Figure 34). This exact fingerprint was not in the Honeyd database, but the closest match was IBM OS/2 Warp 4.0, which is extremely close to the one on the actual device.

```
MAC Address: 00:03:EA:0E:11:67 (Mega System Technologies)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1
closed port
Device type: general purpose
Running (JUST GUESSING): IBM OS/2 4.X (86%)
OS CPE: cpe:/o:ibm:os2:4
Aggressive OS guesses: IBM OS/2 Warp 2.0 (86%)
No exact OS matches for host (test conditions non-ideal).
```

Figure 34. Nmap scan on the ezOutlet2 to determine the operating system

IV. Research Methodology

4.1 Goals

This research focuses on creating honeypots that simulate IoT services. Tests are run to determine how similar these simulated devices are compared to the actual devices and answer the following questions:

1. Do Honeyd IoT honeypots transmit faster or slower than the actual IoT device they are simulating?

2. Are IP packets sent by Honeyd IoT honeypots identical to the ones sent by the actual IoT device they are simulating?

3. Can Honeyd IoT honeypots produce identical Nmap scan results as the actual IoT device it is simulating?

4.2 Approach

4.2.1 Packet Timing and Content.

This experiment determines the average difference in the IP packets between a user accessing the real IoT device and the IoT honeypot of that device. Various response and control variables are highlighted in Sections 4.3 and 4.4.

The simulation scenario consists of a user accessing the web pages of the associated device through the use of a script. The web pages include: the main page, additional pages that include data changes that a user sees (such as date/time, an IP address, etc.), a page that requires user authentication, and a page that does not exist. In the background, the page contents as well as the HTTP/TCP/IP packet headers are captured so they can be compared to each other. A timer is also incorporated to see

how long it takes for the query to receive the main page of the device. There are some aspects that should be different in the page contents and the headers, such as date/time and IP addresses; these are ignored during the comparison. Another scenario is to have more users querying the device to see how this changes response time between the honeypot and IoT device, possibly causing connections to be dropped. These tests are run with three different numbers of total queries and three different numbers of simultaneous users, for a total of nine trials. These factors are discussed in more detail in Section 4.4.

The IoT devices and honeypots are all queried by a Python script, `gethttp.py` (Appendix E). This script takes an IP address, the number of iterations to query the device, and which type of device is being queried (camera, thermostat, or power outlet) as command line parameters. The reason the device type needs to be specified in the parameters is because each device has different pages with different web addresses that need to be compared. For example, the main login page, main camera page, settings page, and a page that does not exist are requested on the camera. This script gathers the HTTP header and HTML page data for the requested page. A timer is also used to determine how long it took the script to query the main page of the device and receive the data. If a device needs to be queried by multiple simultaneous connections, `gethttp.py` is run multiple times from the command line simultaneously using the `&` symbol and a fourth command line parameter that indicates which number connection this is. An example command for querying a device is: `python gethttp.py http://192.168.0.150 100 1 1 & python gethttp.py http://192.168.0.150 100 1 2 & python gethttp.py http://192.168.0.150 100 1 3`. This command runs three simultaneous connections on the IP address 192.168.0.150, parameter 3 being 1 means its querying a TITAThink camera, and the 4th parameter indicates how the folders that contain

the data are numbered. The number from this parameter is used to name the folder where this connection's data is stored. For three simultaneous connections, the folders are named Folder 1, Folder 2, and Folder 3.

Another Python script, `getHeaders.py` (Appendix E) is constantly listening on port 80 while requests are being made by `gethttp.py` and grabbing all TCP/IP packet headers and extracting the IP version, IP header length, IP Time To Live (TTL), IP Protocol, TCP source port, and TCP header length. It does this through the use of sockets. It also counts how many total packets are received. These are put into a file for comparison. This code was adapted from the basic Python sniffer by Silver Moon [43]. The HTTP headers from `gethttp.py` are stored in a separate file for their own comparison.

A Python script for each device, `cameraHTMLCompare.py`, `thermostatHTMLCompare.py`, and `outletHTMLCompare.py` (Appendix E), is used for comparing the HTML data from the captured HTTP code. They take three command line parameters: the folder where the HTML from the IoT device is stored, the folder where the HTML from the honeypot is stored, and a filename for the Comma-Separated Values (CSV) file. These scripts iterate through every file and starts by going line by line comparing each line of the HTML, getting the total number of lines, and keeping track of each time a line is different. There are a predetermined set of line numbers for each device that are expected to be different on specific pages. For example, line 592 in the main page of the Proliphix thermostat contains the date and time. If the main page is being compared and a difference is detected on one of those lines, an expected line difference value is incremented. It then loops through the files again, this time comparing each character on a line and getting a total number of characters. It keeps track of the number of times a character is different. There is set of predetermined ranges of characters that are expected to be different, and if a

different character falls in any of these ranges, then an expected character difference value is incremented. These values are then used in calculating the percent similarity between the two devices based on lines, characters, lines excluding expected differences, and characters excluding expected differences. A line is then written to a csv file for each file in the specified folder.

The header comparisons are done by `headerCompare.py` (Appendix E). The program takes five command line parameters: two filenames for the TCP/IP header capture from the IoT device and honeypot, two filenames for the HTTP header capture from the IoT device and honeypot, and a filename for the csv that will save the comparisons. The program starts by looping through both the IoT device and honeypot TCP/IP header capture files. It goes line by line looking for the word 'IP', because this denotes the beginning of an IP header. When it finds this, it increments the packet counter by one, because it found a new packet. It then looks for the fields: Version, IP Header Length, TTL, Protocol, Source Port, and TCP header length. The line following one of these lines is the value for that header field for this particular packet. These values are then stored in their own list, which contains all the values for this particular field received from the IoT device or honeypot. More loops then parse through each of the lists and compare the values in them to the corresponding list for the other device. For example, the lists `iotIPver` and `honeyIPver` contain the list of IP versions that were gathered from the IoT device and honeypot. These lists are compared until the end of the shorter list is reached. The number of differences are tallied in their own variable, in this example it is called `diffipver`. It denotes the number of times the IP version was different. This is done for all the TCP/IP header fields.

A similar process happens for the HTTP header file. Both the IoT device and honeypot HTTP header files are parsed to find lines that indicate the start of a

header. Since HTTP header fields differ between the device, both the fields and values are being compared. A dictionary is used to store the items, where the key of the dictionary is the HTTP header field or value and the value of that key is the number of times it occurred in the file. If the field or value does not exist in the dictionary, it is added and the value is set to 1. If it does, then the value is incremented by 1. If any of the lines include a day of the week (i.e., Mon, Tue, Wed, etc.) it is ignored. To complete the comparison, both dictionaries are parsed by key. A running total of the number of HTTP header items is obtained by taking the sum of all the values in the IoT device dictionary file. If a key does not exist in the other file, then that keys value is added to a variable that keeps track of the number of differences. If the key does exist, then the values should be the same. If they are not, then the difference between the two values is added to the difference variable. A line is then written to the csv file containing the total number of TCP/IP packets for each device, the number of differences for each TCP/IP field, the number of differences found between the HTTP headers, and the total number of field and values obtained from the HTTP headers.

4.2.2 Nmap Scans.

This experiment uses the Nmap network scanning tool to gather information about IoT devices. It has the ability to determine running services and their ports on a device as well as a guess on the operating system being used by the device. This is based on special tests that Nmap runs and compares to a database of fingerprints. Sections 4.3 and 4.4 highlight the different types of Nmap scans that are run.

A Python script, `nmapScanner.py` (Appendix E), is used to run the Nmap scans on the IoT devices and Honeypots. The script takes 2 command line parameters, the IP address to scan and the number of times to run the scan. The script loops the

specified number of times and run each of the three Nmap scans: SYN, UDP, and FIN. The Nmap scan command used for each of the scans is `nmap -<scan> -T4 -A -v <IP Address>`. The `<scan>` field is replaced with sS for SYN, sU for UDP, and sF for FIN. The content compared includes: the running services, both TCP and UDP, as well as the detected operating system, and the device manufacturer. The output of the Nmap scans are put into text files so they can be observed. The desired content that is compared is extracted from the outputs of the scans and inserted into an Excel spreadsheet to take averages and make comparisons. There are three different types of scans on each device and each scan is run five times for a total of 15 trials per device. Nmap runs the same tests every time it executes. Therefore, five trials is sufficient because it ensures that the Nmap data remained constant and allows for a good scan time average to be taken. These factors are shown in more detail in Section 4.4.

4.3 System Boundaries

Figure 35 shows the System Under Test (SUT), the Honeyd framework of HTTP IoT devices. The main portion of this system, the Component Under Test (CUT), is the main configuration file, `iotHoneyd.conf`, that governs all the running services, scripts, and OS emulation of the honeypots.

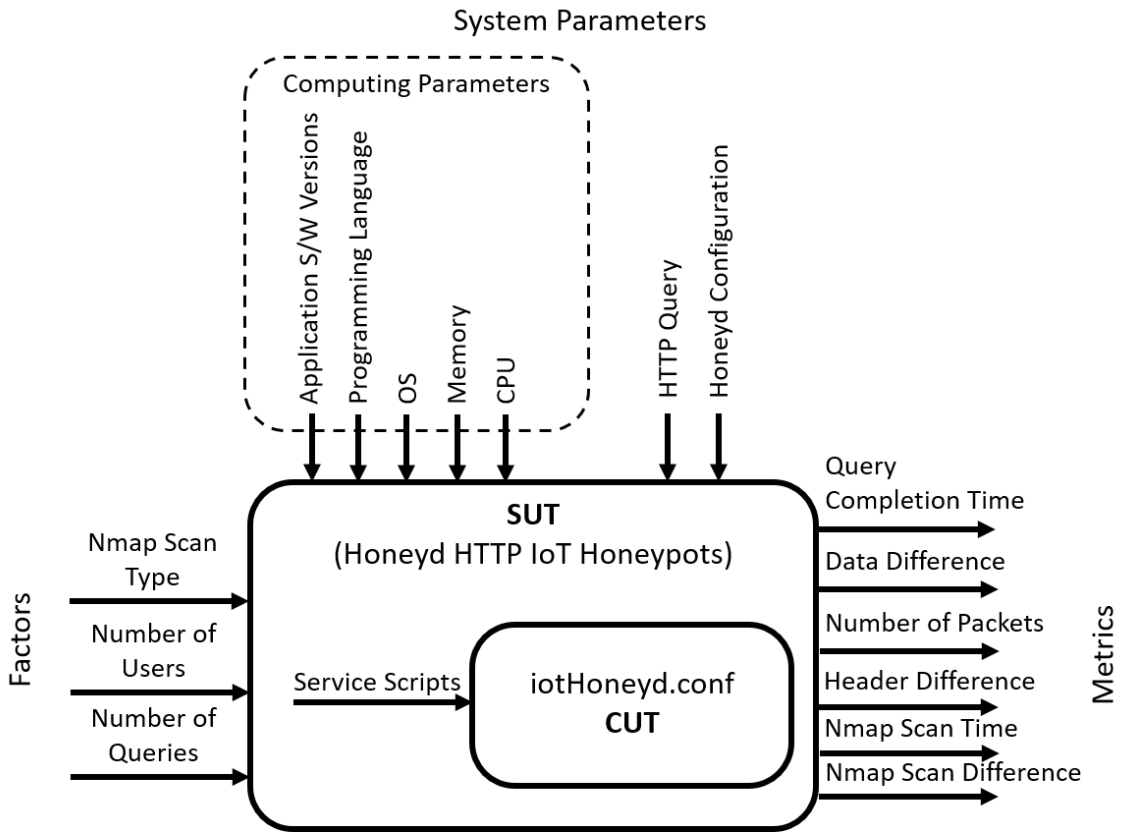


Figure 35. HTTP IoT Honeyd framework

4.4 Parameters and Factors

4.4.1 Assumptions.

When running these scenarios, various assumptions need to be made to ensure that certain problems that may occur do not affect the tests run. The assumptions include:

- Each connection attempt ends with a success. If a connection fails, an entirely empty web page is used for comparison purposes.
- The same commands are used on both the IoT devices and honeypots.

- Date and time of access are different for each attempt.
- Non-standard commands are not attempted against any device.
- All interactions take place on the same network with the same network hardware.
- All interactions begin from the same location with the same hardware. Each interaction uses the same machine so processor specifications are not a factor.
- The same version of all software is used.
- There are no outside interaction with the devices other than the user in the scenario.

4.4.2 System Parameters.

- **Computing Parameters** - The computer that contains the system is constant, a Lenovo laptop with an Intel i7 processor clocked at 2.9 GHz running Windows 10 and 32 GB of RAM. The virtual machine running Honeyd is an Ubuntu 12.04.5 machine with 1 processor core and 4 GB of RAM. The service scripts are written in bash version 4.2.25 and Python version 2.7.3.
- **Query** - The query always attempts to access the web page of the device. The query uses the HTTP protocol on TCP port 80.
- **Honeyd Configuration** - The configuration file that governs the characteristics of the three honeypots is constant. It has a list of the services available as well as the OS personality, all of which are constant.

4.4.3 Factors.

- **Number of Queries (nQ)** - This is the number of times that the user tries to access the device or honeypot. It is the same number between the device and honeypot and it is the same number between different IoT devices for the sake of consistency. The packets from each pair of queries is compared, and the average differences and completion time is taken. A higher number of queries should produce a better average for the differences between the device and honeypot. The different number of queries for each trial are 100, 500, and 1000. These number of queries are chosen because the increasing number of queries ensures that a device can handle many repeated requests. Increasing the number of queries past 1000 made the tests take too long.

- **Number of Users (nU)** - This is the number of users querying the device or honeypot simultaneously. In theory, both devices should be able to handle the same number of users before suffering performance losses. However, it is possible that a device cannot handle a certain number of users. Varying the number of users to see how query time changes could be an effective test. The number of potential users is 1, 5, 10, and 20. Originally, 1, 10, and 20 users were going to be used for all devices. The number is reduced for the Proliphix thermostat because it could only handle 1 user and 20 users took a very long time to complete. The TITAThink Camera and ezOutlet2 is tested by 1, 10, and 20 simultaneous users. The Proliphix Thermostat is tested by 1, 5, and 10 simultaneous users. These numbers are chosen because they increase the load on the devices to see a change in query response time, but not so much as to immediately crash the device or take too long to complete.

- **Nmap Scan Type (sT)** - This is the type of Nmap scan that is executed

on the device or honeypot. Nmap has various scans to choose that use a different method to probe for information. The Nmap scan types are: the TCP SYN, UDP, and TCP FIN scans [31]. These scans are chosen because they provide variation in how Nmap scans the device to see if different scans produced different results. A UDP scan is necessary for the devices with UDP services.

4.4.4 Metrics.

- **Query Completion Time (QT)** - A numerical value, measured in milliseconds. This measures the amount of time it takes for the user to receive a complete response from the IoT device or IoT honeypot. This is accomplished by a multiple scripts making the requests at the same time and a built-in timer calculates how long the query takes. In networking, time can be more variable because of other network traffic at the time. The isolated testbed network as seen in Figure 8 helps with this interference.

- **Data Difference (DD)** - A numerical value, measured in number of lines and number of characters. This measures the number of different lines/characters between the HTML data from the actual IoT device and the IoT honeypot. The page size should be the same between both devices. Therefore, getting the raw number of different lines/characters should be an accurate measure of how different the packets are from one another. There are some lines/characters that are expected to be different, and these are calculated separately to see how the percentage improves when those are taken into account.

- **Number of Packets (NP)** - A numerical value, measured in number of packets. This measures the number of packets that are sent in a particular trial. If both the device and honeypot send the same amount of data, they should send a similar number of packets. Sometimes packets are lost in a transmission

and are resent. This may mean that the number of packets are not exactly the same, but they should ideally be within a small margin of error.

- **Header Difference (HD)** - A numerical value, measured in number of differences. This measures the number of differences in the TCP, IP, and HTTP headers. The fields in the TCP and IP headers are the same, and the information in those fields is compared to make sure they are the same. The HTTP header fields and contents are custom to each device, and both of these are compared between the IoT device and the honeypot to make sure they are identical.

- **Nmap Scan Time (ST)** - A numerical value, measured in seconds. Nmap scans run a series of tests on a device to gather information about its services and operating system. The amount of time it takes for a scan to complete may be an indication of the difference between the IoT device and honeypot.

- **Nmap Scan Differences (SD)** - A numerical value, measured in number of differences. Nmap scans provide different information including open ports, detected operating system, and more. The information to be compared is the running services and detected operating system. Ideally, all this information should be the same between a honeypot and the device it is simulating. This information is recorded as the number of different services and whether or not the OS is different.

4.5 Methodology

The evaluation technique for this experiment is direct measurement of time and number of differences. Some of the scripts listed in this section are only for compiling data into a single file/folder for easier processing. They have no other function related

to the program so their source code is not listed in this thesis. The process for the experiments are:

Packet Content Test

1. Connect IoT device to network
2. Start the honeypots with the script `startHoney.sh` (Appendix A)
3. Start capturing TCP/IP packets by running `getHeaders.py` (Appendix E)
4. EITHER
   (a) Run the `gethttp.py` (Appendix E) script to connect to the IoT device for 100, 500, or 1000 iterations. Create multiple simultaneous connections based on the experiment (1, 5, 10, or 20 connections)
   (b) Run the `gethttp.py` (Appendix E) script to connect to the honeypot device for 100, 500, or 1000 iterations. Create multiple simultaneous connections based on the experiment (1, 5, 10, or 20 connections)
5. When `gethttp.py` (Appendix E) finishes, stop the `getHeaders.py` (Appendix E) script
6. Stop the honeypots
7. Use the scripts `changeheadfilename.py` and `concatHTML.py` to put all the query times and header information into the same files and the HTML data into the same folder
8. Use the script `cameraHTMLCompare.py`, `thermostatHTMLCompare.py`, or `outletHTMLCompare.py` (Appendix E) to compare files for a specific device and store the data in a csv
9. Use the script `headerCompare.py` (Appendix E) to compare the TCP, IP, and HTTP headers to each other and store the data in a csv
10. Transpose the query response times and other data into an Excel file for analysis

Nmap Scan Test

1. Connect IoT device to network
2. Start the honeypots with the script `startHoney.sh` (Appendix A)
3. Begin each of the 3 types of scan, 5 times each, on a device or honeypot using the `nmapScanner.py` (Appendix E) script. Each type of scan is run individually, waiting for 60 seconds after completion.
4. Repeat for all 6 devices/honeypots
5. Examine the service differences, OS differences, manufacturer differences, and scan times and put them in an Excel file for analysis.

4.6 Apparatus

The workstation that contains all the virtual machines is a Lenovo W541 laptop with an Intel i7 processor clocked at 2.9 GHz running Windows 10. It also has 32 GB of RAM to accommodate for multiple virtual machines running on it. This laptop is running Windows 10 and has Wireshark running on it for scanning the IoT devices to construct the honeypots.

The systems used to run the honeypots includes Honeyd and virtual machines run through VMWare Workstation Pro 14, a program for managing virtual machines. Honeyd is run on a virtual machine running Ubuntu 12.04.5 and this configuration runs all the honeypots since Honeyd can run many virtual machines. Three distinct honeypots are running on their own IP addresses with the ports related to their services open.

There is an Ubuntu 14.04.02 virtual machine using VMWare. This machine is on the same LAN as the other devices and is the main host for querying the honeypots and IoT devices, as well as running the Nmap scans. This singular machine creates the simultaneous connections to each device. It is given 2 processor cores and 2 GB

of RAM.

The IoT devices, TITAThink TT520PW, Proliphix NT130h, and ezOutlet2 EZ-22b are all contained on the same LAN as the honeypots. They are physically connected via Ethernet to a TP-Link N300 Router. The laptop running the honeypots and testing the honeypots is also connected to this network. No other devices are on this network and the router is not connected to the Internet, which limits extraneous network traffic.

4.7 Results

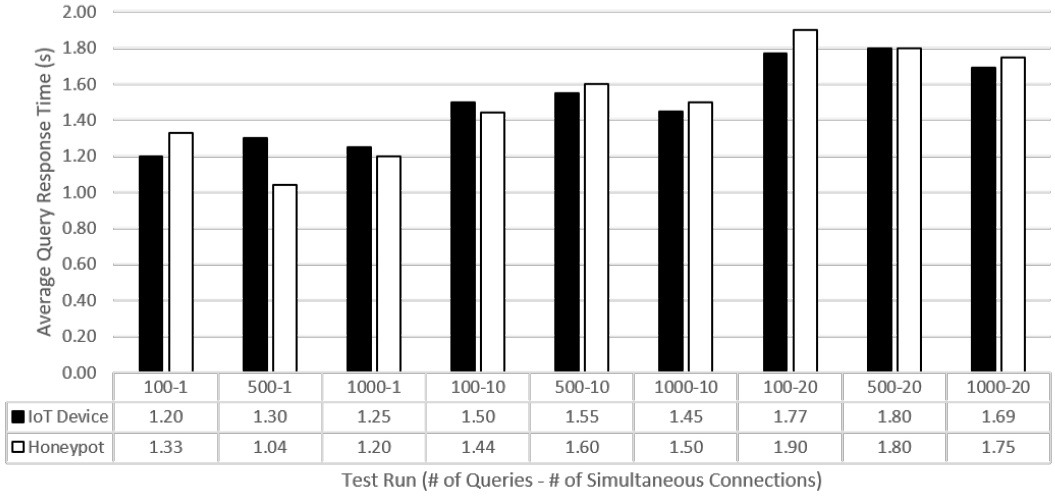
The data gathered from the experiments are processed in Excel. It is used to perform the statistical analysis as well as to generate plots to show the differences between the IoT devices and the honeypots. Bar graphs and scatter plots that show data points very close together indicates that an IoT device is very similar to its honeypot. Some of graphs used include:

- **Bar Graphs**
  - Average Query Response Time Between IoT Device and Honeypot
  - Percent Similarity of IoT and Honeypot Data
  - Percent Similarity of IoT and Honeypot Headers
  - Number of TCP/IP Packets on IoT Device and Honeypot
  - Average Nmap Scan Time on IoT Device and Honeypot
- **Scatter Plots w/ Trendline**
  - IoT Device Average Query Response Time Based on Number of Simultaneous Connections

– Honeypot Average Query Response Time Based on Number of Simultaneous Connections

The figures below highlight some of the potential graphs that are used to highlight the data. Figure 36 is a bar graph that shows the average query response time for all trials on both devices. This allows direct comparison between the device and honeypot on each trial. Each bar provides the results of an individual test, and the labels used in the x-axis signify which test is run based on the number of queries and the number of simultaneous connections. For example, the label '100-1' indicates the trial with 100 queries by 1 user. This method for labeling trials is used throughout the thesis. Figures 37 and 38 are bar graphs show the percent similarity of the HTML code and TCP/IP/HTTP headers between the device and honeypot. Each graph highlights the three trials of varying queries for the same number of simultaneous connections. The result is three graphs, one for each set of simultaneous connections. Figure 37 breaks each set of bars into the line and character similarity with and without expected differences. Figure 38 breaks each set of bars into TCP/IP and HTTP percent similarity. Figure 39 shows the number of TCP/IP packets that are sent by the IoT device or honeypot depending on the trial. They are grouped together to allow comparison of the number of packets sent for each trial. Figure 40 shows the average completion time for the three Nmap scans on both the IoT device and honeypot. Finally, Figure 41 is an example scatter plot of all average query response times for every trial for an IoT device or honeypot. Some of the points on these graphs overlap due to extremely similar response times between the trials. There are three points to indicate the three trials, 100, 500, and 1000 queries, that take place with the indicated number of users in the x-axis. The trendline included shows how the average query response time increases as more simultaneous users query the device. Putting the graph of the IoT device and honeypot next to each other with

the same scale allows comparison between the trendline. It shows which device is affected more from the increasing number of users. These graphs provide a helpful visual representation for comparison of the data obtained from the tests highlighted in this chapter.



	100-1	500-1	1000-1	100-10	500-10	1000-10	100-20	500-20	1000-20
IoT Device	1.20	1.30	1.25	1.50	1.55	1.45	1.77	1.80	1.69
Honeypot	1.33	1.04	1.20	1.44	1.60	1.50	1.90	1.80	1.75

Figure 36. Example bar graph of average query response time

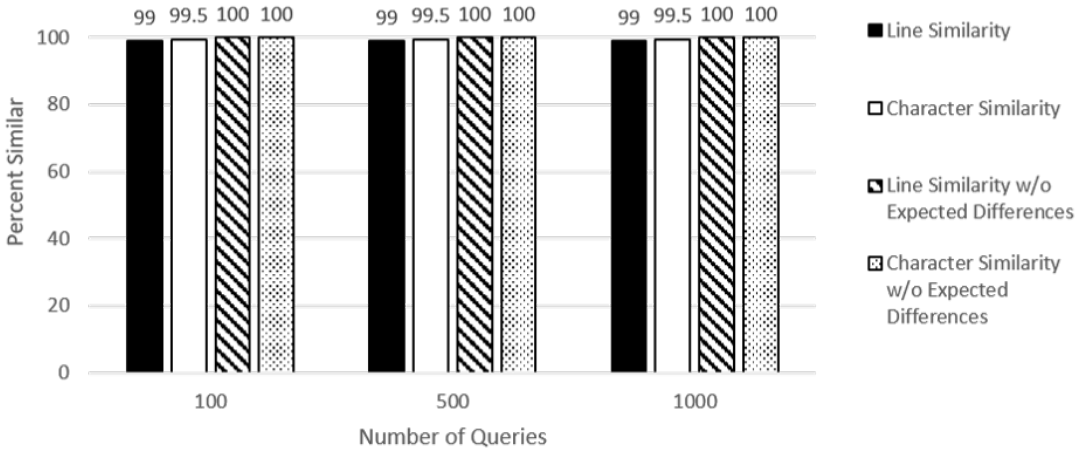


Figure 37. Example bar graph showing code percent similarity for 1 user

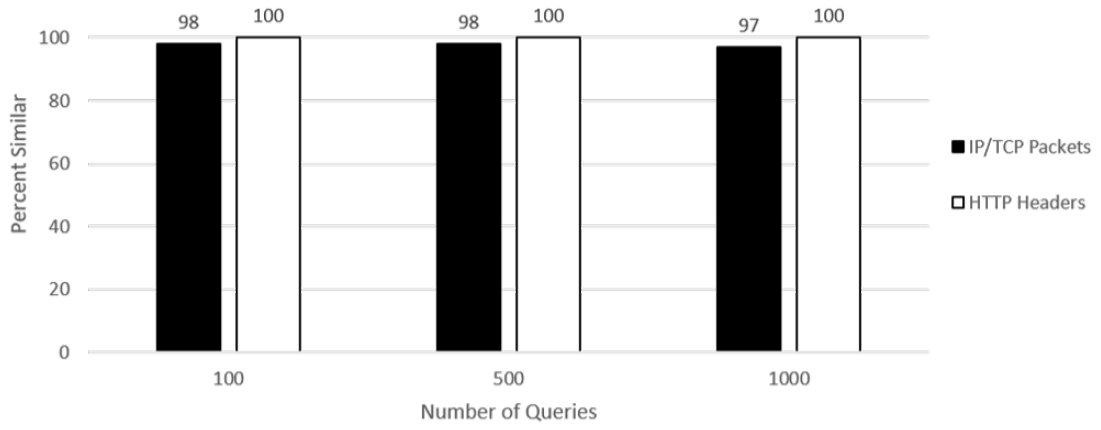


Figure 38. Example bar graph of TCP/IP and HTTP header percent similarity for the trials with 1 user

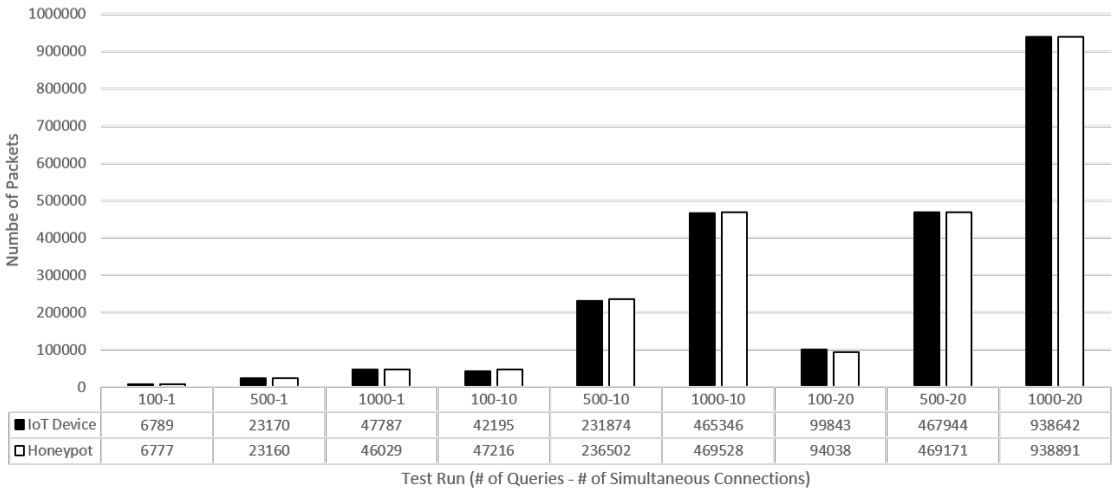


Figure 39. Example bar graph of number of TCP/IP packets

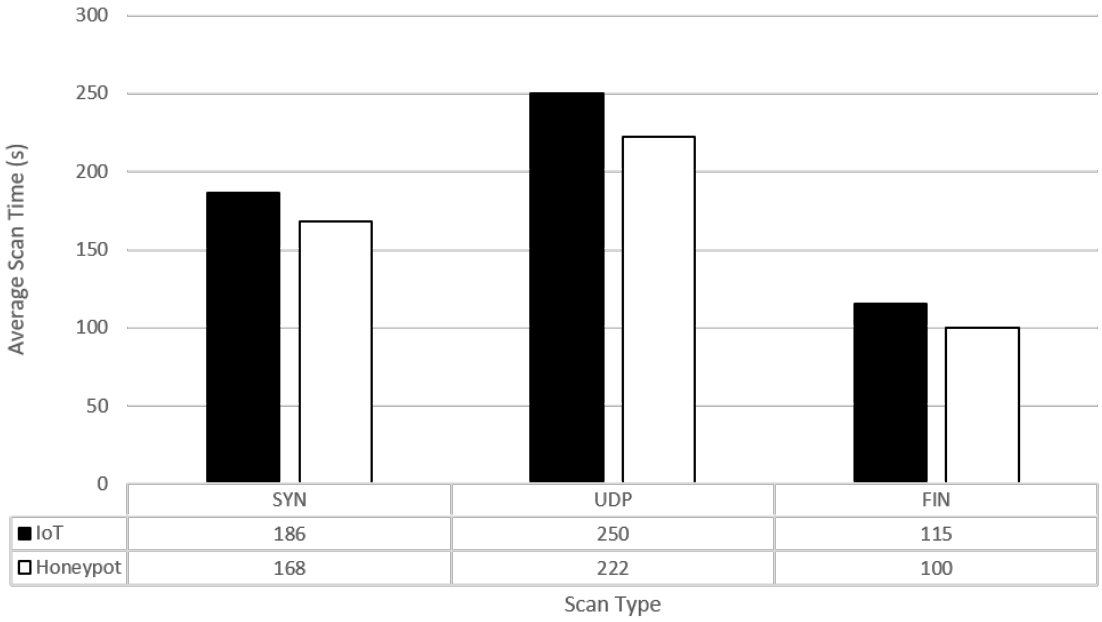


Figure 40. Example bar graph of average Nmap scan time

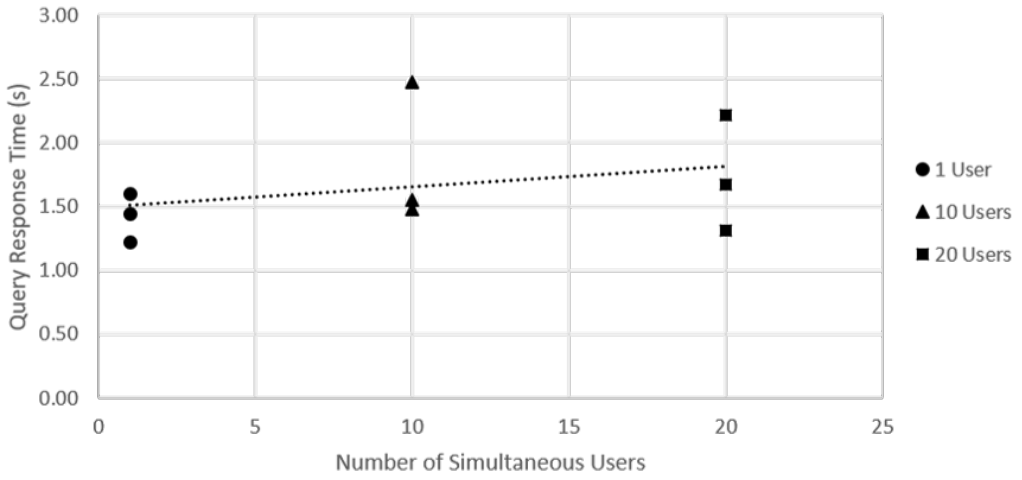


Figure 41. Example scatter plot of average query response time with trendline

Since the experiment is comparing how similar an IoT device is to an IoT honeypot, using a t-test is the ideal test for this research. A t-test is used to determine

if there is a statistically-significant difference between the means of two sets of data [44]. The null hypothesis, μ_0, is that there is no difference between the means of two sets of data. If the p-value is less-than or equal to 0.05, the null can be rejected and it can be concluded that there is a difference between the means with 95% confidence. Otherwise, the null fails to be rejected; then it can be assumed that there is no statisticallyu-significant difference between the mean values of the data. A t-test assumes that both sets of data are normally distributed.

It is generally accepted that a set of data with at least 50 data points can assumed to be normally distributed [45]. If there are not that many data points, an Anderson-Darling test can be performed to determine with 95% confidence whether or not a given set of data is evenly distributed [46]. The null hypothesis, μ_0, is that the data follows the normal distribution. If the p-value is less-than or equal to 0.05, the null can be rejected and it can be concluded that the data does not follow the normal distribution with 95% confidence.

If a data set does not have a normal distribution, an alternative to the t-test is a Mann-Whitney U test (sometimes referred to as the Wilcoxon Rank-Sum Test) [47], [48]. The null hypothesis, μ_0, of the Mann-Whitney U test is that the two sets of data came from the same population. If the p-value is less-than 0.05, the null can be rejected and it can be concluded that the data does not come from the same population with 95% confidence. While the null hypothesis of the t-test and Mann-Whitney U test are technically different, in essence they are the same and the Mann-Whitney U test is commonly used in place of a t-test when the data is not normally distributed [48].

Another consideration is that t-tests come in two forms, equal and unequal variance t-tests. The test used is entirely dependent upon the variance of the sets of data. If there are the same number of data points in both sets of data, an equal

variance t-test can be used [44]. If not, an F-test can be used to determine whether or not the variances of two sets of data are equal [49]. The null hypothesis, μ_0, is that the variance of two sets of data is equal. If the p-value is less-than or equal to 0.05, the null can be rejected and it can be concluded that the variance of both sets of data are not equal. In this case, an unequal variance t-test can be used. If the p-value is greater-than 0.05, the null fails to be rejected which means that it can be assumed with 95% certainty that the variances are equal. A equal variance t-test can be used in this case. The t-tests and F-tests are run in Excel using the Analysis ToolPak add-in that comes packaged with Excel. The Anderson-Darling test is run in an Excel document created by McNeese [46]. The Mann-Whitney test is run in Excel using the Real Statistics Resource Pack [48].

These statistical tests are run on the query response time and Nmap scan time data. This is because it is quantitative data with many data points. They are not run on the code, header, and Nmap similarity data because the numbers for these metrics are gathered from qualitative comparisons. There are not two sets of quantitative data to perform a t-test on. Two sets of qualitative data are compared to get a singular number for percent similarity. These sets of data are qualitatively analyzed in the following chapter.

4.8 Chapter Summary

This chapter discusses the research goals and questions that are attempting to be answered in this research. It also shows a diagram of the SUT and the assumptions, parameters, factors, and metrics of the experiment. The chapter also covers the experimentation methodology, the apparatus for testing, and how the results are presented and analyzed.

V. Results and Analysis

5.1 Overview

This chapter highlights the results of the experiments outlined in Chapter 4, showing the resulting information for each metric from Section 4.4.4. These metrics are used to determine if Honeyd is able to successfully produce convincing web-based IoT honeypots.

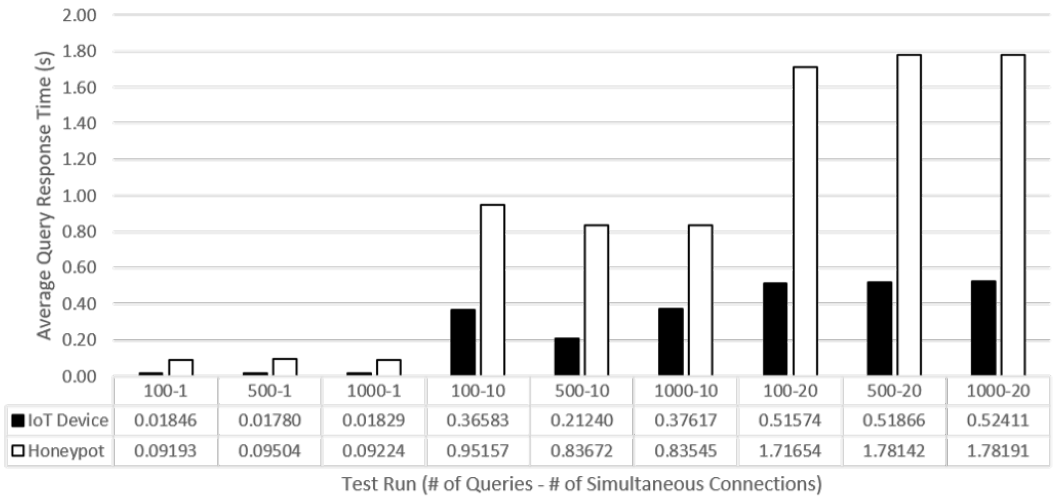
5.2 Metric 1 - Query Completion Time

5.2.1 TITAThink Camera.

Between the IoT device and honeypot, one thing remains certain, as the number of simultaneous users increases, the response times for each query increases as well (Figures 42, 43, and 44). The honeypot's query response time appears to grow at a much faster rate compared to the TITAThink camera. It is expected that the response time would grow, but it appears that the camera was able to accommodate more users than the honeypot. Importantly, the response times stay relatively the same regardless of the number of queries, either 100, 500, or 1000. Only the number of users affects the time. The data for this experiment can be found in Table 5 in Appendix F.

Both sets of data for the camera and the honeypot have the same number of entries, meaning an equal variance t-test can be used. With the null hypothesis set as $\mu_0 = \mu_1$, all of the t-Tests reject the null hypothesis by having p-values that are less than 0.05. This means that with 95% confidence, none of the sets of query response times could be considered identical, because there is a statistically-significant difference between the means. These tests are shown in Figures 75, 76, 77, 78, 79, 80, 81, 82, and 83 in Appendix G.

The reason for this discrepancy could be due to the lack of computing power on the VM running the honeypot. If more resources are allocated to Honeyd, it could process requests faster and have response times more in line with the actual device. However, since the IoT device is so small and low powered, it may not be the amount of computing resources but how the resources are allocated. A lower resource intensive operating system than Ubuntu may decrease response times. In addition, making the code more efficient and using a programming language that is faster than Python may also bring response times down. As far as increasing speeds when there are more simultaneous users, this could prove to be quite the challenge because response times increase much faster for the honeypot than for the IoT device.



	100-1	500-1	1000-1	100-10	500-10	1000-10	100-20	500-20	1000-20
IoT Device	0.01846	0.01780	0.01829	0.36583	0.21240	0.37617	0.51574	0.51866	0.52411
Honeypot	0.09193	0.09504	0.09224	0.95157	0.83672	0.83545	1.71654	1.78142	1.78191

Figure 42. Camera average query response time for device and honeypot

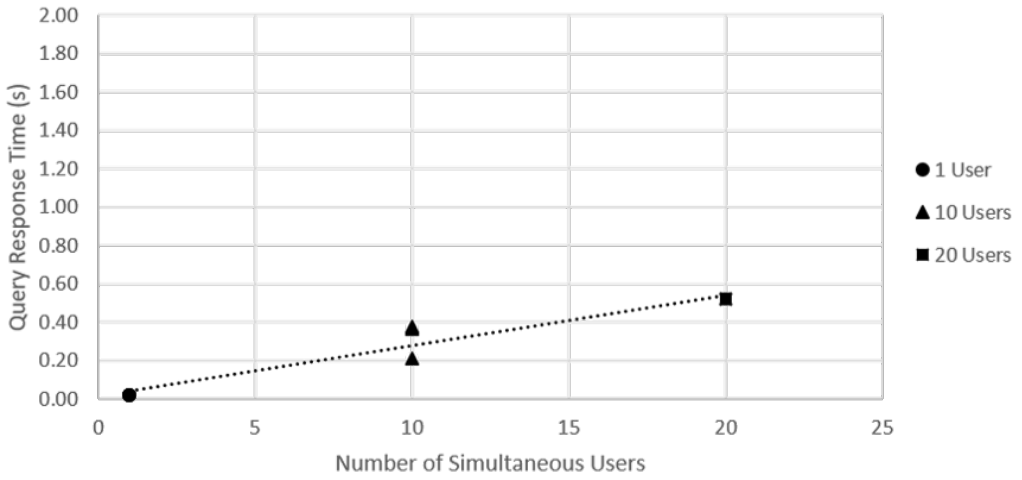


Figure 43. IoT camera average query response versus number of users

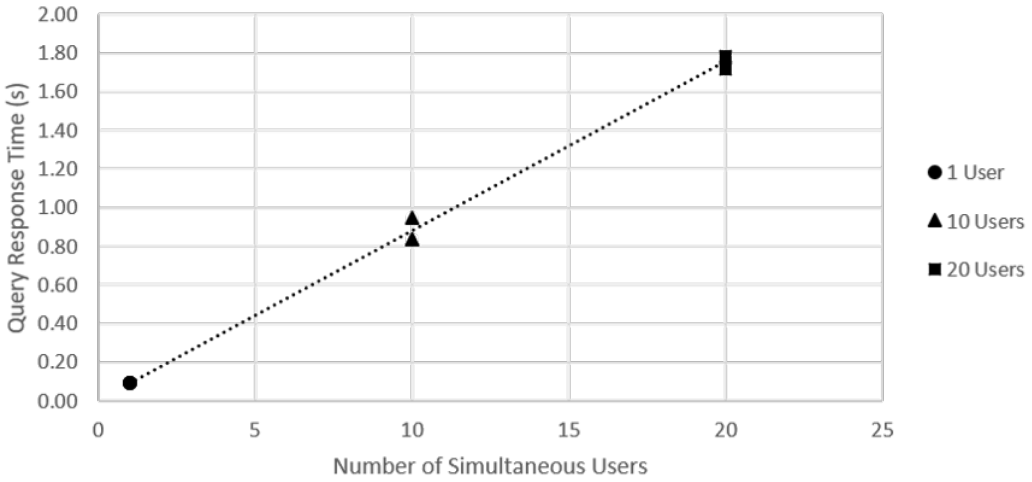


Figure 44. Camera honeypot average query response versus number of users

5.2.2 Proliphix Thermostat.

The test results on the thermostat follow the same general pattern as the camera; as the number of users increases, the average query response time increases as well.

This increase is due more to the fact that there are outliers in some of the tests. One of the trials with five users actually had a longer average time than any of the response times on the trials with ten users. Tests with the same number of users against the thermostat are not extremely similar, differing by 1 second in some trials. The thermostat seems to be more non-deterministic in query response time, responding faster or slower for no reason. The honeypot average query response time increases, but not nearly as much as the camera honeypot does with increased users. Figures 45, 46, and 47 show the average response times for each trial as well as showing how they change as users are added. The data for this experiment can be found in Table 6 in Appendix F.

The trials with only one user all have the same number of data points, so an equal variance t-test can be used. The trials with five and ten users have dropped packets, meaning an F-test is performed to determine which t-test to use. All of these trials reject the null hypothesis that the variances between the data sets are the same. This means that an unequal variance t-test is used. The null hypothesis for the t-Tests is $\mu_0 = \mu_1$ to determine if the average query response times have no significant difference in their means with a 95% confidence. The t-tests from the first trials, with only one user, all reject the null meaning that there is a statistically-significant difference between the means. However, for the trial with 100 queries, the p-value of 0.013 is very close and almost fails to reject the null. The means are nearly identical and it is the larger variance of the thermostat that causes the test to reject the null. The trials with five simultaneous users also reject the null hypothesis, but again the trial with 100 queries had a p-value of 0.0104 which is also very close to failing to reject the null. Finally, the trial with 10 users actually fails to reject the null on the trial with 100 queries having a p-value of 0.115. This means that there is not a statistically-significant difference between the means of the thermostat and honeypot

average query response times, with 95% confidence. The 500 query trial with ten users rejects the null with a p-value of 0.037. This is the closest to almost failing to reject the null of any trial, so it was very close to not having a statistically-significant difference between the mean. The 1000 query trials rejects the null outright. The results of the F and t-tests can be found in Figures 84, 85, 86, 87, 88, 89, 90, 91, and 92 in Appendix G.

The Proliphix thermostat is that it is not able to adequately handle more than one simultaneous connection. Tests are reduced to 5 and 10 simultaneous users instead of 10 and 20 for this reason. This was the main reason why the average query response times does not change very much when more users are added for specific trials. The significantly higher outliers are due to the fact that the thermostat sometimes stalls on a query that was happening simultaneously, taking upwards of a minute to fulfill that request instead of rejecting it outright. This skews the average query response time to be very high if this happens multiple times. It also causes some larger variances, which affects the results of the t-tests, if the thermostat was more non-deterministic in the time it took to respond to a query. This result and the ability to only handle one user at a time was probably due to the older hardware on the device. The Proliphix thermostat is by far the oldest device tested.

To make the honeypot respond as quickly as the thermostat, a delay is added because the thermostat is about a full second slower than the honeypot. The only problem being that due to the non-deterministic nature of how the thermostat responds, there are still some discrepancies. A solution in the future would be to add a random delay instead of a fixed number to increase the variability of the honeypot's response time as well. Honeyd has no problem handling multiple simultaneous connections, so possible modifications might need to be made to the source code to institute a function that would reject connections once a certain number of simulta-

neous users are reached.

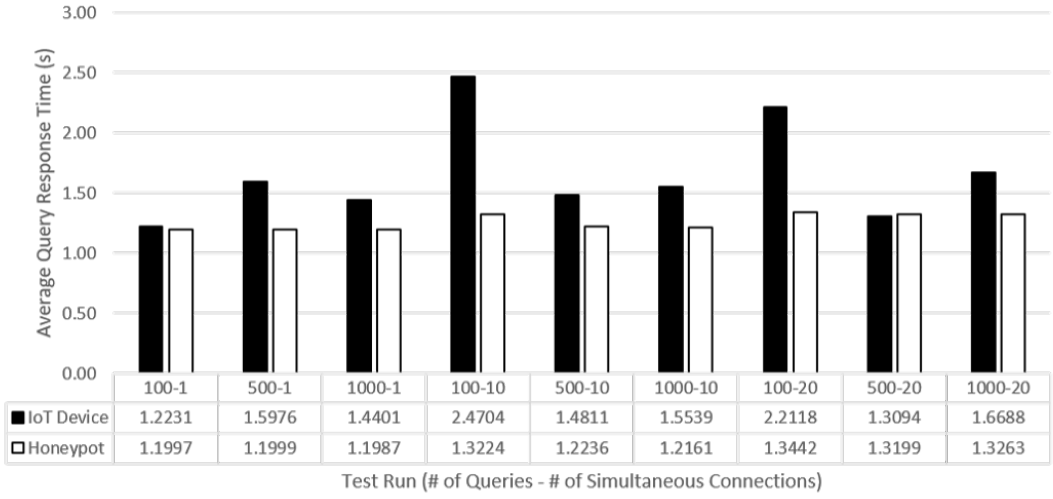


Figure 45. Thermostat average query response time for device and honeypot

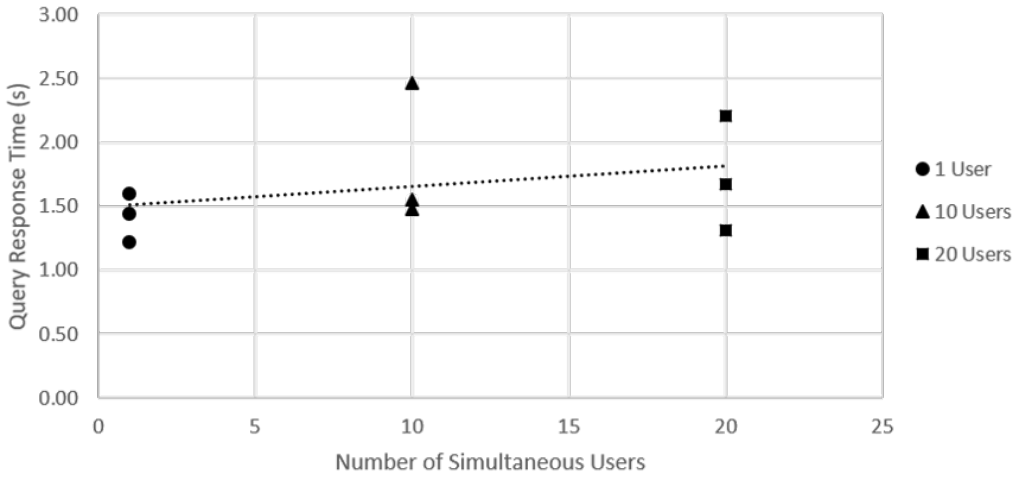


Figure 46. IoT thermostat average query response versus number of users

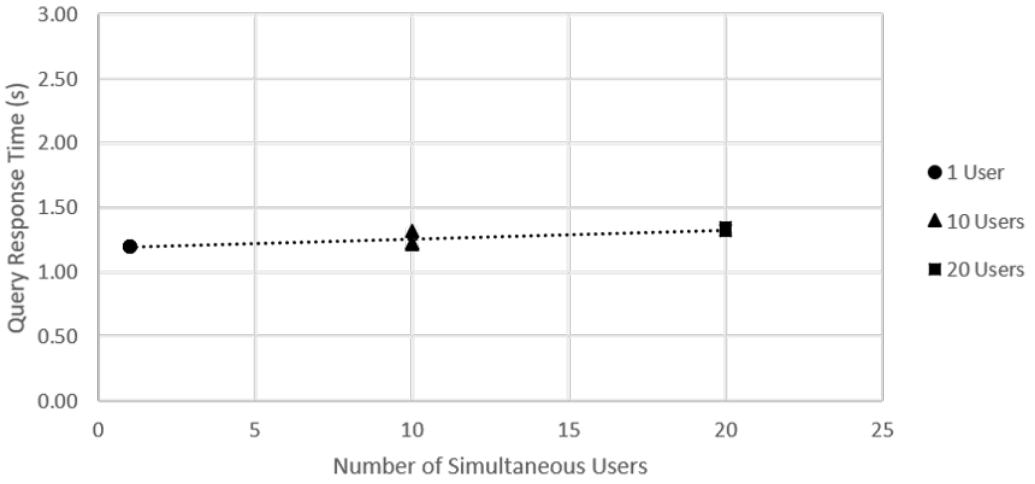


Figure 47. Thermostat honeypot average query response versus number of users

5.2.3 ezOutlet2 Power Outlet.

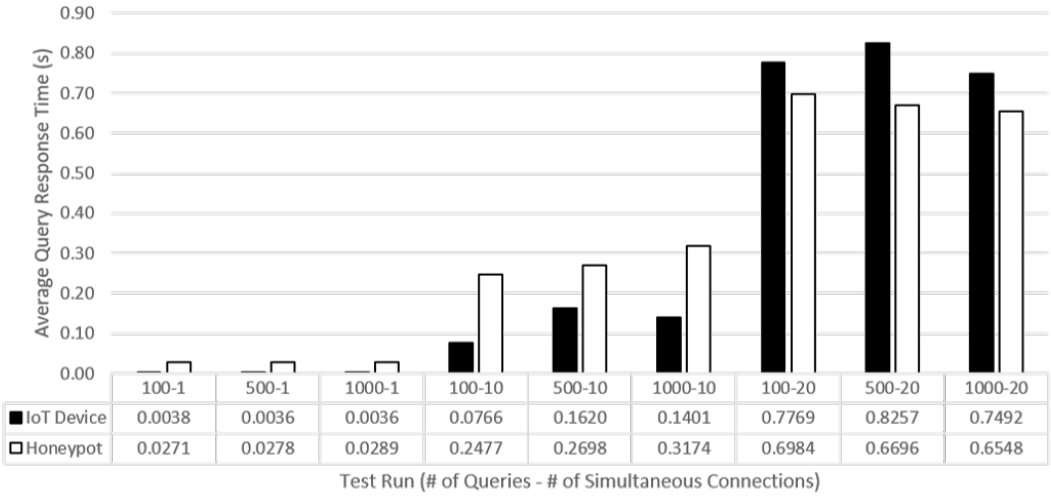
As seen in the other scenarios, when the number of users increased, the average query response time increases as well. The ezOutlet2 seems to grow slightly faster than the honeypot based off the trendline from Figures 49 and 50. Also worth noting is that while the ezOutlet2 seems to be significantly faster than the honeypot in the one and ten user scenarios, it actually responds slower than the honeypot in the twenty users scenario. This is shown by the raw average query response time by trial highlighted in Figure 48. This is a unique scenario in comparison to the other devices in that sometimes the honeypot is slower than the IoT device and sometimes faster, depending on how many users are trying to access the device. The data for this experiment can be found in Table 7 in Appendix F.

Only the trials with twenty users have dropped packets, which means the other trials have an equal number of data points. Therefore, the one and ten user trials use an equal variance t-Test. The F-tests determine that unequal variance t-tests

are used for the twenty user trials. The null hypothesis for the t-Tests is $\mu_0 = \mu_1$ to determine if the average query response times have no significant difference in their means with a 95% confidence. Performing the t-tests shows that all trials, save one, rejected the null hypothesis that there was no statistical difference between the mean query response times. This means that it can be assumed that the average query response times are not the same with 95% confidence. The one trial where the test fails to reject the null was the trial with 100 queries and twenty users which has a p-value of 0.186. This trial has no statistically-significant difference between the means of the average query response time. Unlike the thermostat tests, there are no p-values that are very close to failing to reject the null. The results of the F and t-tests can be found in Figures 93, 94, 95, 96, 97, 98, 99, 100, and 101 in Appendix G.

In comparison to the other IoT devices, the ezOutlet2 responds the fastest when only one user is accessing it. The limited amount of data being transmitted by the ezOutlet2 contributes to its fast response time. As in the other two sections, bringing the honeypot response times down would probably require optimizing the Honeyd code, providing more resources to the VM, and optimizing the scripts being used by Honeyd in this system. The discrepancy in query response time is too large to pass any t-tests. The main reason why there is one success in failing to reject the null is because the ezOutlet2 begins dropping packets and experiencing slowdowns when 20 users query the device at the same time. This provides a much larger variance of 6.68 compared to the other trials. This caused the t-test to fail to reject the null in comparison to the other tests because there was a larger range of values from the slowdowns experienced. The other tests with 20 users also have larger variances, but there are many more data points to get a more accurate reading on the data, and it shows that statistically there was a significant difference between the means. This is also the reason why the ezOutlet2 is slower than the honeypot at 20 users despite

being faster in the initial trials. Twenty users is around the ceiling for the number of users the ezOutlet2 can handle.



	100-1	500-1	1000-1	100-10	500-10	1000-10	100-20	500-20	1000-20
IoT Device	0.0038	0.0036	0.0036	0.0766	0.1620	0.1401	0.7769	0.8257	0.7492
Honeypot	0.0271	0.0278	0.0289	0.2477	0.2698	0.3174	0.6984	0.6696	0.6548

Figure 48. Outlet average query response time for device and honeypot

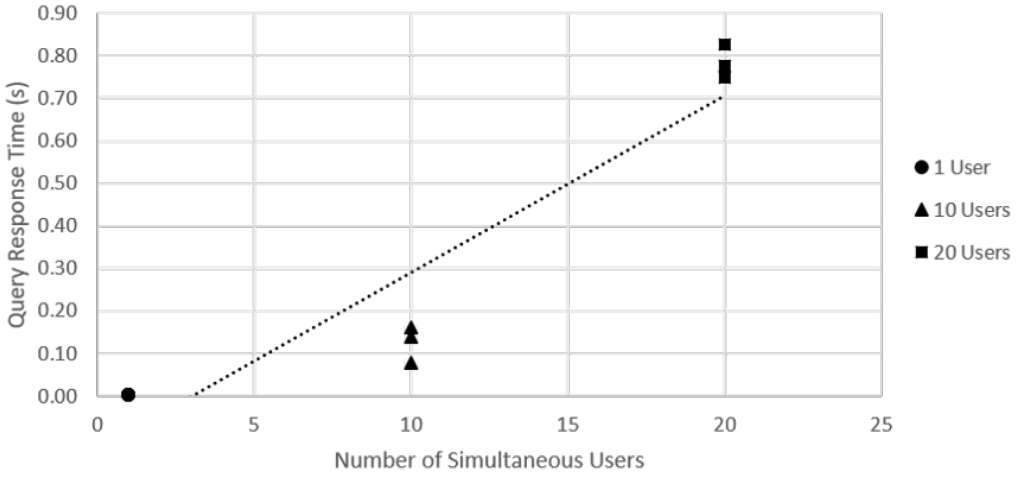


Figure 49. IoT outlet average query response versus number of users

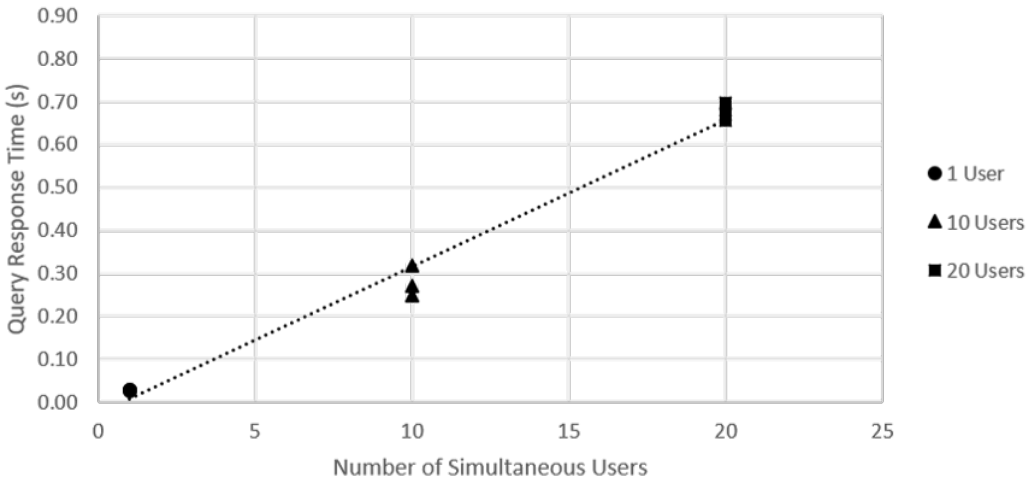


Figure 50. Outlet honeypot average query response versus number of users

5.3 Metric 2 - Data Difference

5.3.1 TITAThink Camera.

Comparing the HTML codes proves to be quite successful. The data sent from both devices is identical in every scenario. The HTML code for this device has no expected differences, unlike on the other devices. In addition, the number of users does not introduce any loss of data. Both the IoT device and honeypot can handle the queries of up to 20 users without losing data. Figures 51, 52, and 53 show the percent similarity for each test completed. The data for this experiment can be found in Table 8 in Appendix F.

Early on, one of the challenges seemed to be incorporating a static picture system that appeared as a dynamic video feed. However, because the TITAThink Camera uses an Motion Joint Photographic Experts Group (MJPEG) Live as the video feed, it took in a Joint Photographic Experts Group (JPG) image as input in the HTML code. The file extension for an MJPEG Live is the same as a regular JPG image, so

it was simple to send a static image instead of the live image without altering the HTML code at all. This allows for 100% code similarity.

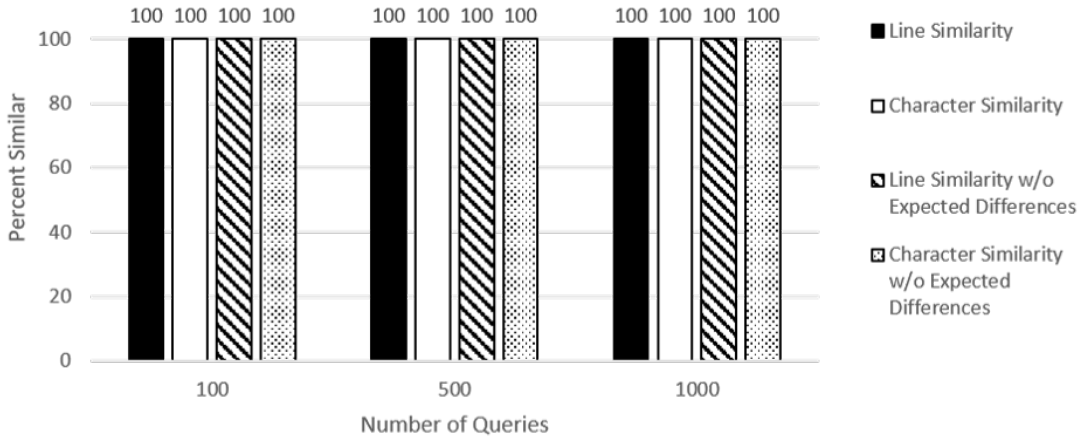


Figure 51. Code similarity for camera with 1 user

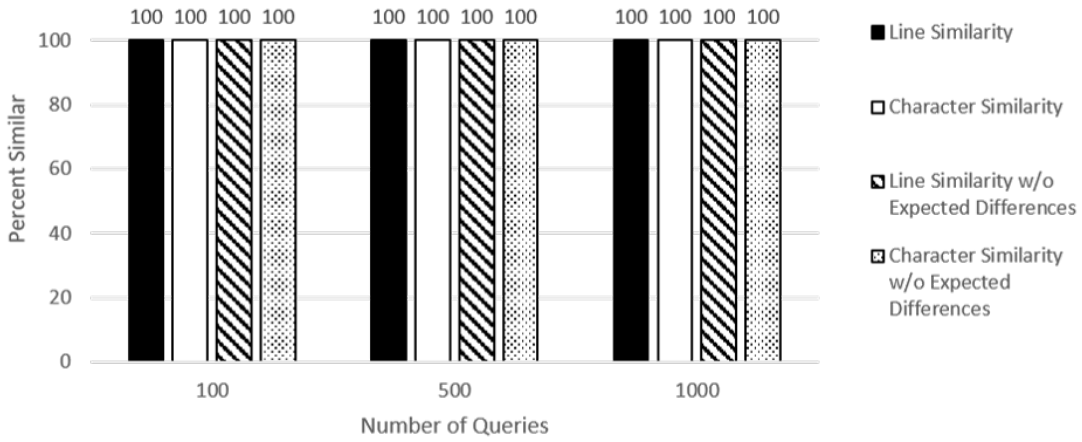


Figure 52. Code similarity for camera with 10 simultaneous users

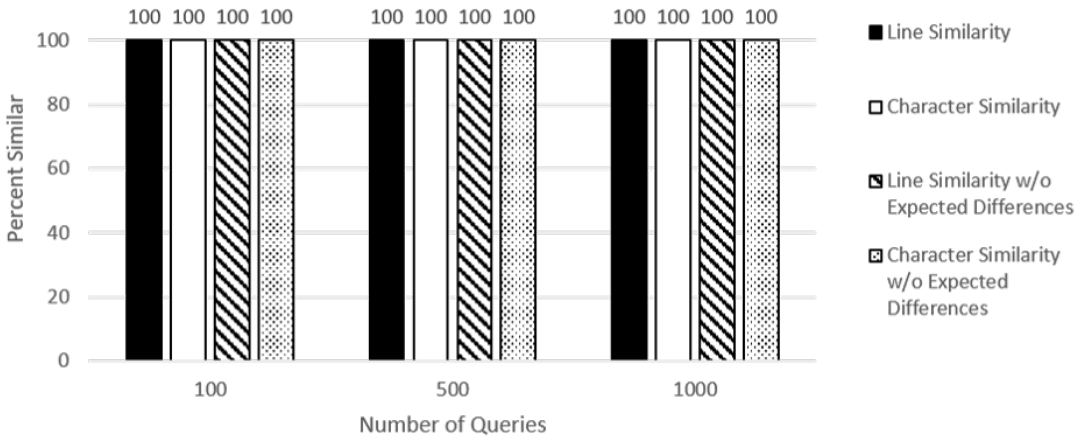


Figure 53. Code similarity for camera with 20 simultaneous users

5.3.2 Proliphix Thermostat.

The tests run on the thermostat are not as successful as on the camera. The results for one simultaneous user has approximately 99.5% code similarity. The code similarity for five and ten simultaneous users, however, is approximately 20% and 10% respectively. Figures 54, 55, and 56 highlight the similarities of the code for each trial. The data for this experiment can be found in Table 9 in Appendix F.

There are some expected differences in these files. Data such as the date, time, and temperature are expected to be different, and the tests correctly show where the files are different and ignores these expected differences. One of the reasons for the small discrepancy in the one user trial has to do with one line in the main page, "var adStat = parseInt("0");", which would randomly alternate between 0 and 1. It is usually 0, but sometimes it changes to 1 for no apparent reason. This discrepancy is very infrequent because it still produces a very high percentage of similarity. The only solution is to determine exactly why it is changing randomly and then alter the script to match the other device. The percentages are low on the five and ten user trials because the thermostat could not serve the requests of more than one user.

This produces many comparisons of a page from the honeypot with a blank HTML page from the thermostat. Before the thermostat starts rejecting the other users, it might send a response such as "Please try again". This response is also analyzed as 0% similarity. As stated previously, a potential solution to this would be to change the Honeyd source code to reject more than one simultaneous connections.

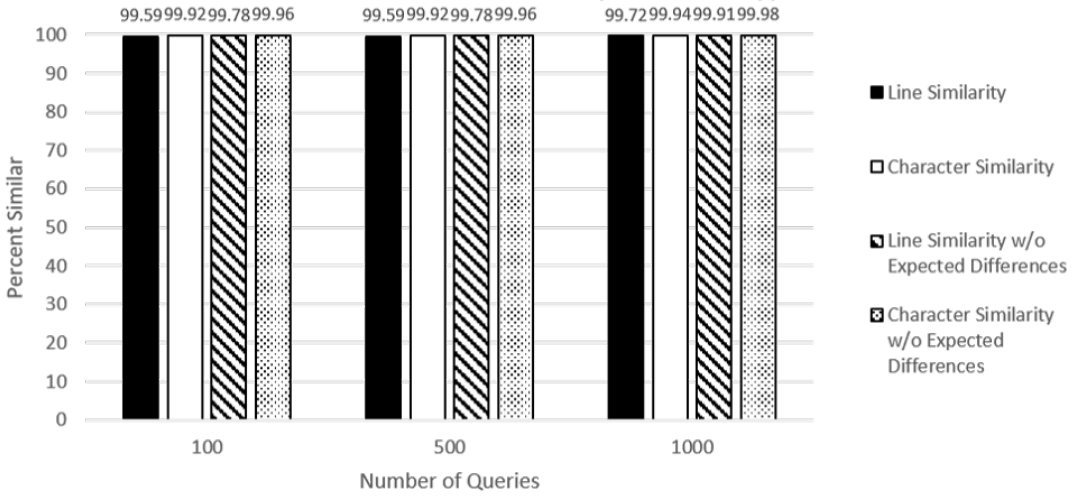


Figure 54. Code similarity for thermostat with 1 user

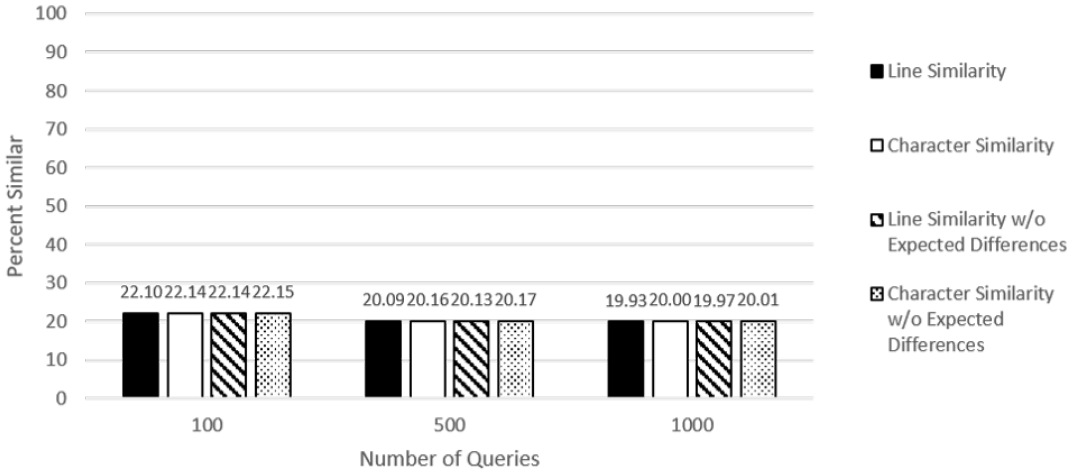


Figure 55. Code similarity for thermostat with 5 simultaneous users

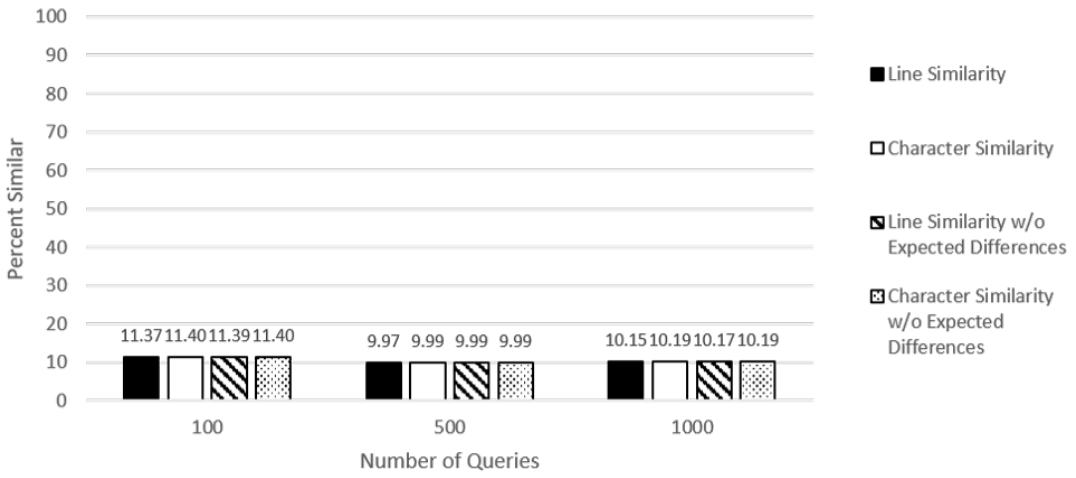


Figure 56. Code similarity for thermostat with 10 simultaneous users

5.3.3 ezOutlet Power Outlet.

The ezOutlet2 honeypot has a much higher code similarity than the Proliphix thermostat. This is mostly due to the fact that the outlet is able to handle more users than the thermostat. The tests for one and ten users has 100% code similarity, but the tests with twenty users only have averages between 86% and 95% code similarity. The breakdown of code similarity based on the tests can be seen in Figures 57, 58, and 59. The data for this experiment can be found in Table 10 in Appendix F.

Just like the thermostat, this device has some expected differences, mainly with the date, time, and IP addresses. The tests are able to properly identify all the differences and then ignore the ones that are supposed to be there. The reason there was some discrepancy when twenty users are querying the device is because the ezOutlet2 begins to drop some packets as too many users query the device. This limitation is not found with the honeypot. To rectify this issue, changes would need to be made to the Honeyd source code to disallow that many users from accessing the device at the same time. Twenty users seems to be the upper limit for the ezOutlet2,

so the comparison is still very close. It is not nearly as limited as the thermostat which can only handle one user.

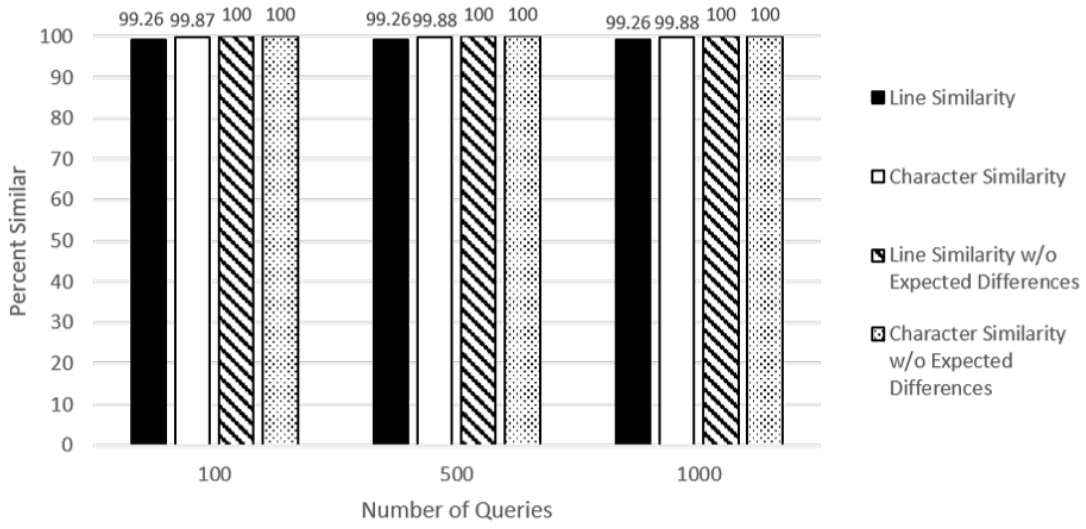


Figure 57. Code similarity for outlet with 1 user

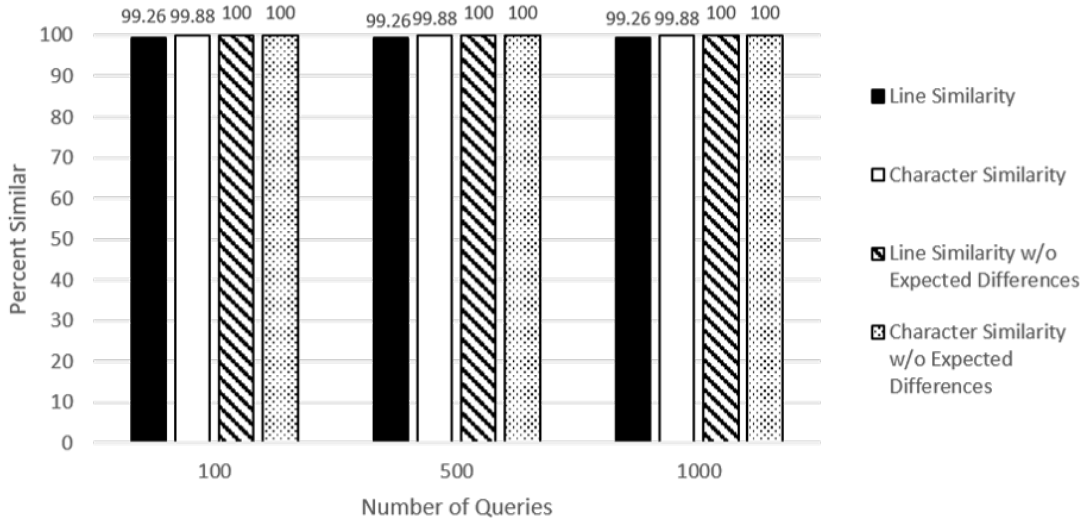


Figure 58. Code similarity for outlet with 10 simultaneous users

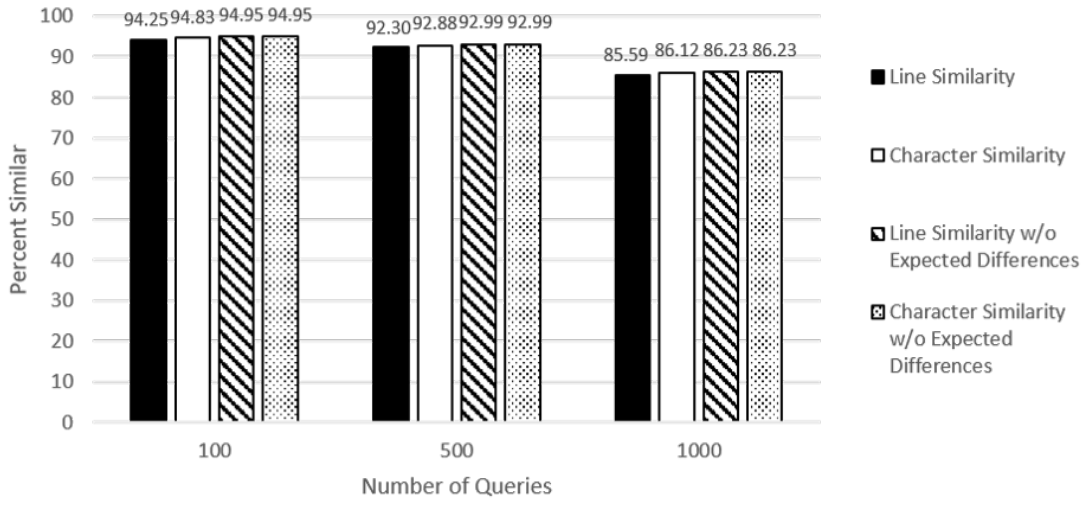


Figure 59. Code similarity for outlet with 20 simultaneous users

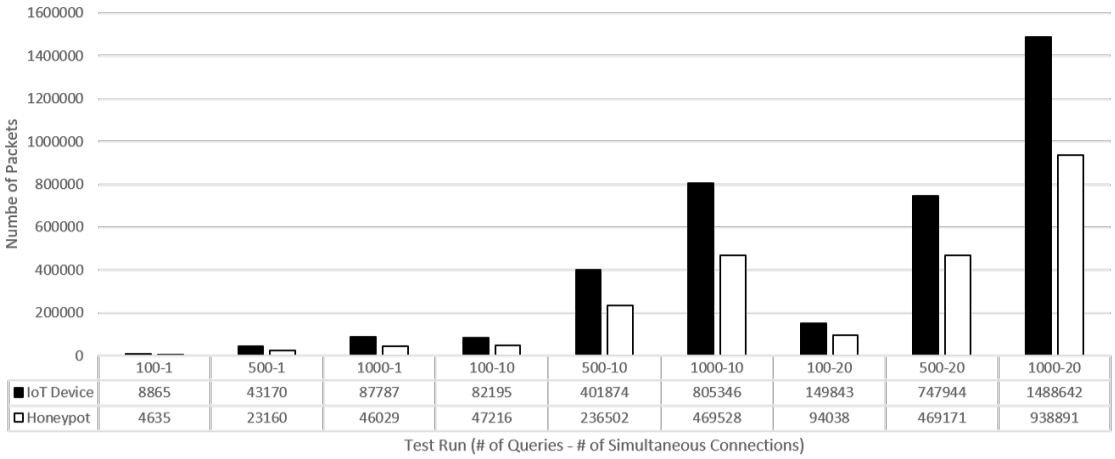
5.4 Metric 3 - Number of Packets

The number of packets is always going to be variable. Sometimes packets get dropped or interference causes them to get resent. That is why differing tests of various query numbers are run to try and normalize for this variability. However, it is still possible that a device ends up sending more packets than another for one reason or another, and these tests try to illustrate exactly how many packets each device sends.

5.4.1 TITAThink Camera.

In all experimental trials, the TITAThink camera sends nearly double the number of packets as the honeypot does. It is not affected by the number of queries or users, and the number is always significantly more. Figure 60 shows how the number of packets differs between both devices in each trial. The data for this experiment can be found in Table 11 in Appendix F.

The reason that the number of packets is so different is likely due to the fact that the TITAThink camera is sending a live video feed, while the honeypot is only sending a static image. It takes much more data to send a live feed. To account for this, an alternative strategy may be to continually send the picture repeatedly while a user is on the page. This would require more back-end work and may even alter query response time, but it may be an avenue worth exploring to bring this figure more in line with the actual device.



	100-1	500-1	1000-1	100-10	500-10	1000-10	100-20	500-20	1000-20
IoT Device	8865	43170	87787	82195	401874	805346	149843	747944	1488642
Honeypot	4635	23160	46029	47216	236502	469528	94038	469171	938891

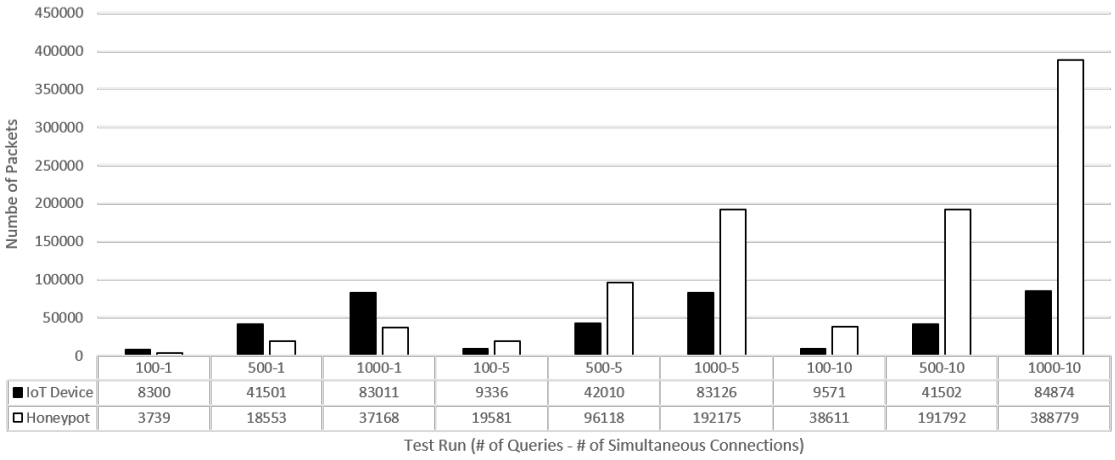
Figure 60. The number of TCP/IP packets sent by both the IoT camera and honeypot camera

5.4.2 Proliphix Thermostat.

In the trials with only one user, the thermostat produces more packets than the honeypot. However, in the trials with more than one user, the honeypot produces many more packets than the thermostat. These results are highlighted in Figure 61. The data for this experiment can be found in Table 12 in Appendix F.

The reason the honeypot is shown as producing more packets when there is multiple users is due to the inability of the Proliphix thermostat to serve more than one user. It originally follows suit with the TITAThink camera by sending more packets

than the honeypot. This pattern would have likely continued if it does not drop the majority of the requests that are sent to it. Based on the tests with no dropped packets, the thermostat is sending more data than the honeypot. To bring this value more in line, in depth Wireshark scans would need to be done to observe exactly what the thermostat is sending with these requests. That way, the scripts can be altered to make these numbers more similar. Also, as stated in Section 5.3.2, changes to Honeyd's code to only allow a certain number of connections would increase the similarity in the trials with more than one user.



	100-1	500-1	1000-1	100-5	500-5	1000-5	100-10	500-10	1000-10
IoT Device	8300	41501	83011	9336	42010	83126	9571	41502	84874
Honeypot	3739	18553	37168	19581	96118	192175	38611	191792	388779

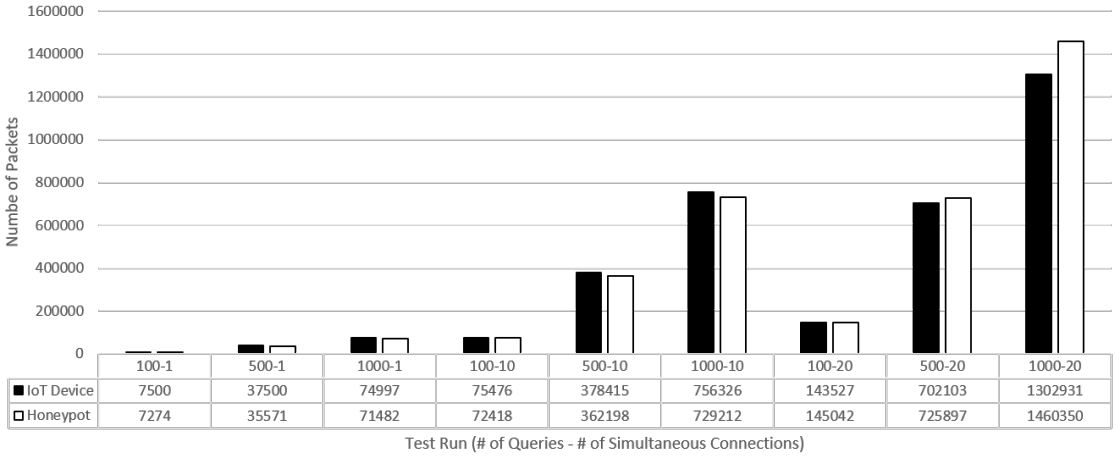
Figure 61. The number of TCP/IP packets sent by both the IoT thermostat and honeypot thermostat

5.4.3 ezOutlet Power Outlet.

Throughout all tests, the number of packets sent by the ezOutlet2 and the honeypot are very similar. Almost all tests have very similar packet numbers, though the ezOutlet2 always produces a few more packets. The tests with discrepancies come from the tests with 20 users. The 100 query and 500 query, 20 user tests have the honeypot with slightly more packets, but the 1000 query 20 user test has 160,000 more packets received from the honeypot than the IoT device. This is due to the

dropped packets discussed in Section 5.3.3. The breakdown of the packet numbers can be seen in Figure 62. The data for this experiment can be found in Table 10 in Appendix F.

This device had a similar trend as the other IoT devices by sending more packets than the honeypots during the experiments, excluding the trials where packets were dropped. These numbers are more similar in comparison with the other devices. In depth Wireshark scans should be done to determine exactly what the ezOutlet is sending. The script could be altered to send more data and bring these values even closer. Since this device is also dropping packets at a certain number of users, altering the Honeyd code to drop packets once a certain number of users is reached would not only help the code similarity, but would also make the number of packets more similar.



	100-1	500-1	1000-1	100-10	500-10	1000-10	100-20	500-20	1000-20
IoT Device	7500	37500	74997	75476	378415	756326	143527	702103	1302931
Honeypot	7274	35571	71482	72418	362198	729212	145042	725897	1460350

Figure 62. The number of TCP/IP packets sent by both the IoT outlet and honeypot outlet

5.5 Metric 4 - Header Difference

5.5.1 TITAThink Camera.

When comparing both the TCP, IP, and HTTP headers between the device and honeypot every test results in 100% similarity. There is no difference in any of the contents of the TCP/IP headers, and the HTTP headers have the same fields with the correct information in them. Date and time are included in the HTTP packets, but they are ignored because they are expected to be different. Figures 63, 64, and 65 show the breakdown of how similar the headers are for each run. The data for this experiment can be found in Table 11 in Appendix F.

The reason the headers are so similar is because the TITACamera happens to share the same TCP/IP header values that Honeyd sends. Honeyd handles the transport layer protocol functions for its honeypots, and it is not possible to change them without source code modification. It does work out for this scenario, but more extensive code modifications may be required for other scenarios.

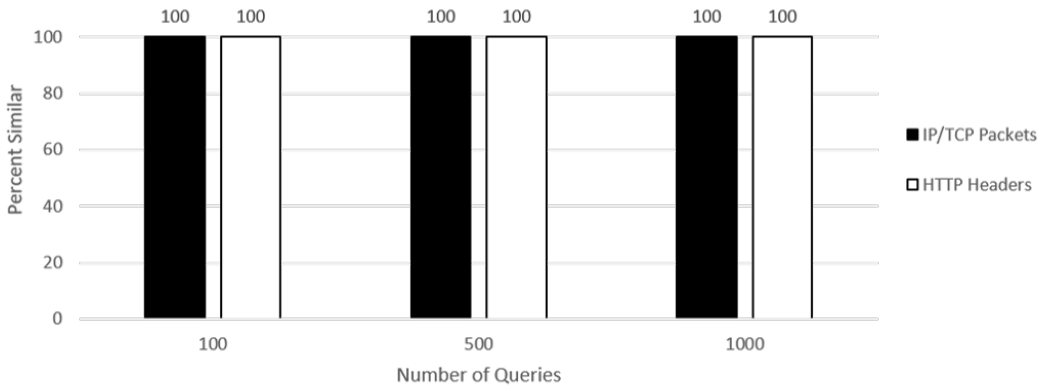


Figure 63. Camera header similarity 1 user

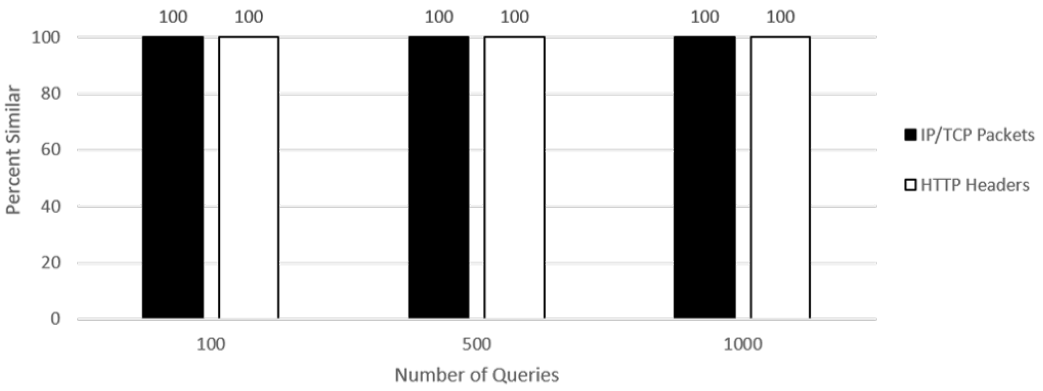


Figure 64. Camera header similarity 10 users

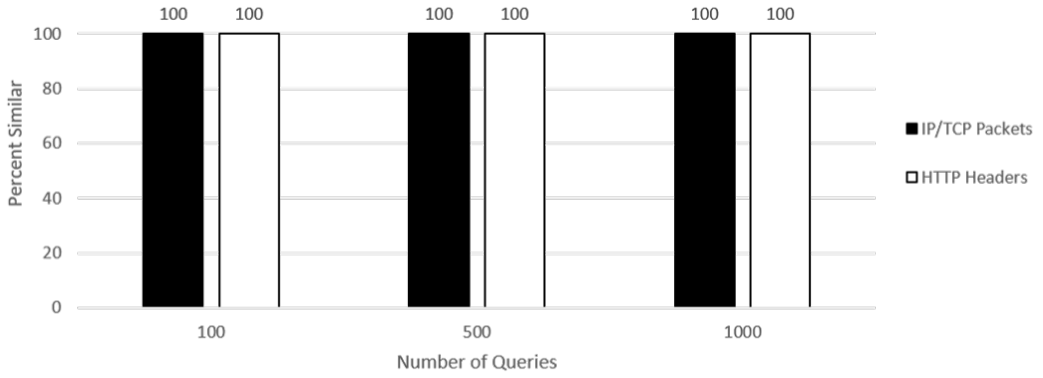


Figure 65. Camera header similarity 20 users

5.5.2 Proliphix Thermostat.

Despite the difference in packet numbers, all of the fields in the TCP/IP packets are the same between the thermostat and the honeypot. Regardless of the trial, the same information appears, and the HTTP headers are nearly as similar. In all nine trials, the HTTP header similarity ranged from 99.5%-100%. Figures 66, 67, and 68 show the breakdown of how similar the headers are for each run. The data for this experiment can be found in Table 12 in Appendix F.

The discrepancies come from when the thermostat would very rarely have the content-header length set to 16 instead of 26. Upon investigation, this has to do with the "Please Try Again" page that the thermostat sends if it does not drop the connection of an additional user, but is still unable to process the request. If a user makes a request for any page and the thermostat is already occupied with another connection, it sends a "Please Try Again" page with the different content-header length, or it drops the connection entirely and stops serving it. As stated previously, rectifying this issue would require altering Honeyd code to limit the number of connections and sending this alternate header.

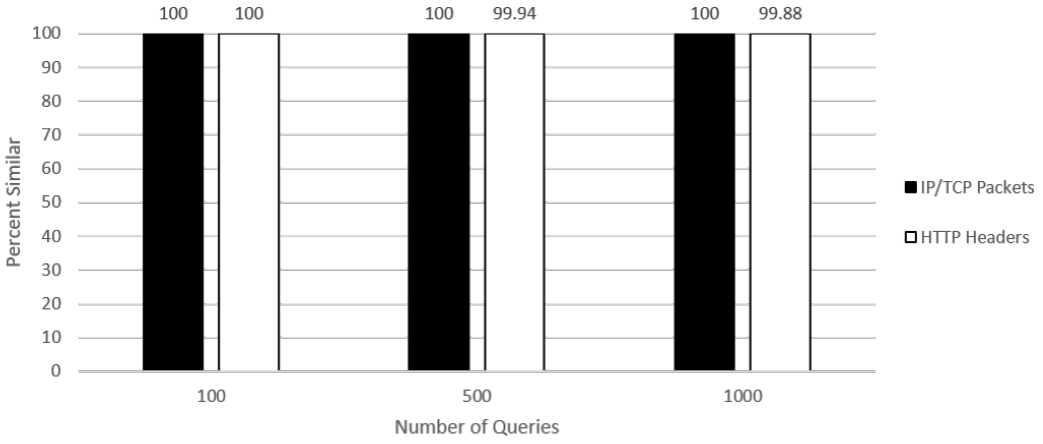


Figure 66. Thermostat header similarity 1 user

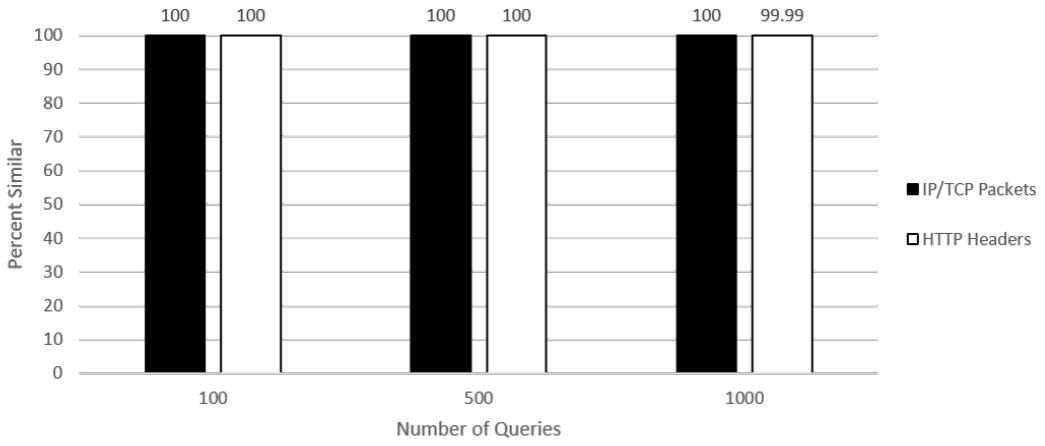


Figure 67. Thermostat header similarity 5 users

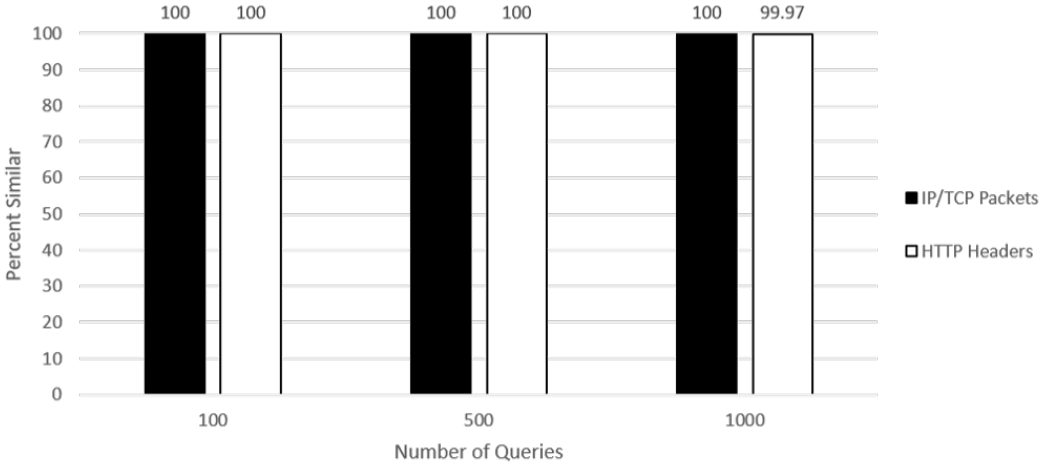


Figure 68. Thermostat header similarity 10 users

5.5.3 ezOutlet Power Outlet.

Even though the packet numbers are the same for the power outlet, the packet headers do not meet this same level of success. The HTTP headers have perfect similarity across all packets, but the TTL in the IP header is different on every packet from the honeypot. With that exception, all other TCP/IP headers are the

same for all tests. The TTL difference makes the headers approximately 20% less similar. Figures 69, 70, and 71 show the breakdown of the similarity of headers between the device and honeypot. The data for this experiment can be found in Table 13 in Appendix F.

The TTL sent by Honeyd in the IP header is preset. Honeyd handles all the TCP/IP traffic in the background, and there does not seem to be a way to change the contents of these headers easily. A potential workaround to this would be to alter the source code of Honeyd to allow the user to choose the contents of the TCP/IP headers. The ability to do this would allow 100% similarity in all headers for any type of honeypot.

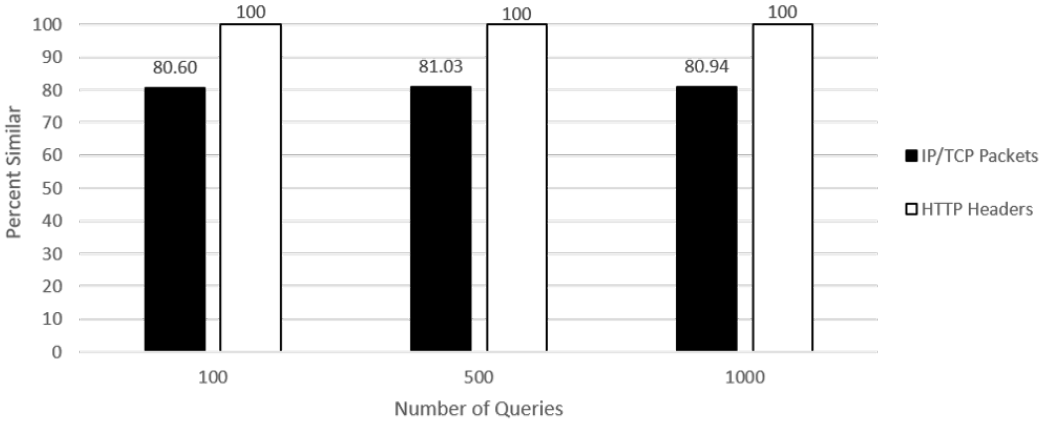


Figure 69. Power outlet header similarity 1 user

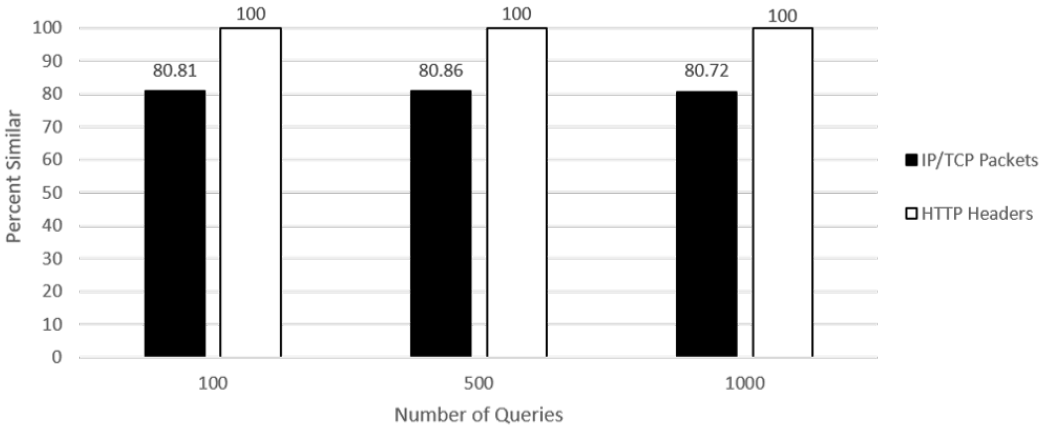


Figure 70. Power outlet header similarity 10 users

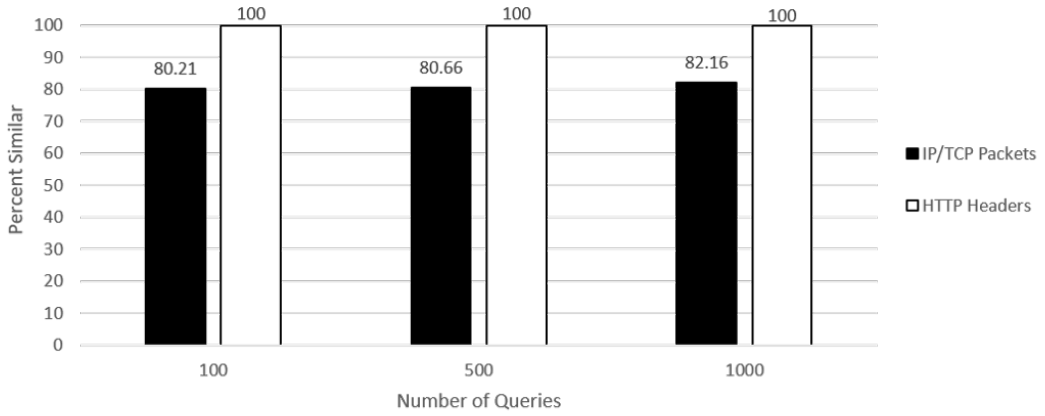


Figure 71. Power outlet header similarity 20 users

5.6 Metric 5 - Nmap Scan Time

5.6.1 TITAThink Camera.

For two of the scans, SYN and FIN, the honeypot is a bit slower than the TITA-Think camera. The honeypot is slower than the camera during normal queries so it is expected that the Nmap scan would also be slower as a result. However, the camera is 5-times slower than the honeypot in the UDP scan. These discrepancies in time

can be viewed in Figure 72. The data for this experiment can be found in Table 14 in Appendix F.

As stated in Section 4.7, normal distribution is assumed for data with at least 50 data points, but this test does not have that many data points [45]. The Anderson-Darling test for normality is used to see if the data is normally distributed for a t-test. The results of the Anderson-Darling tests are highlighted in Table 2. If both p-values from the Anderson-Darling test are greater than 0.05, the final p-value is calculated from a t-test. Otherwise, a Mann-Whitney U test calculates the final p-value.

None of the data sets have a two-sided p-value greater than 0.05 from the Anderson-Darling test, so a Mann-Whitney U test is used. After performing the Mann-Whitney U tests on the data, none of the scans have a two-sided p-value greater than 0.05, which means they all reject the null. These are included in the 'Final p-value' column in Table 2. Therefore it can be concluded that none of the trials of response times between the two devices could be from the same population with 95% confidence. The full test results can be found in Figures 102, 103, and 104 in Appendix G.

Based on the previous experiments, it makes sense for the honeypot to be slower in the SYN and FIN scans. This could be due to lack of resources in the VM, more overhead from the OS, sub-optimal code, or a sub-optimal programming language. The large UDP discrepancy, however, is attributed to the camera rejecting UDP packets that are sent too quickly. Observing the output of the Nmap scans, Nmap keeps increasing the sending delay because it is sending packets too quickly, which the honeypot can handle but the camera cannot. A solution to bring these numbers closer together would be to determine what exactly Nmap sends for the UDP scan and write a script to increase the delay when the honeypot receives these packets.

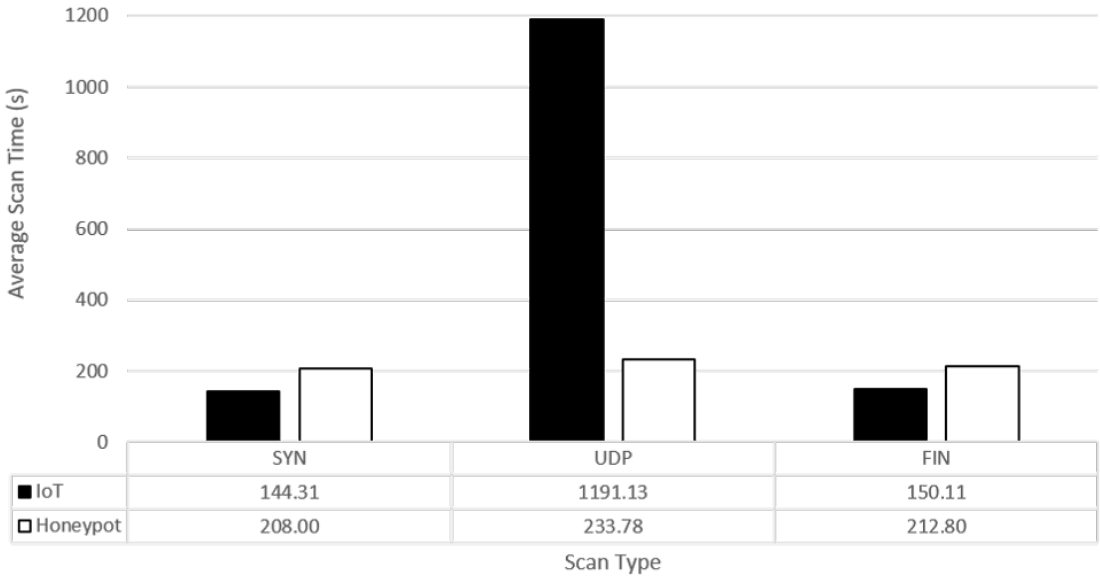


Figure 72. Average Nmap scan time for camera

Table 2. TITAThink camera and honeypot Nmap scan time Anderson-Darling results and final p-value

Scan Type	Camera p-value	Honeypot p-value	Both > 0.05?	Final p-value
SYN	0.0831	0.0071	No	0.01219
UDP	0.0724	0.0264	No	0.01219
FIN	0.0111	0.1226	No	0.01219

5.6.2 Proliphix Thermostat.

In all three Nmap scans, the scans on the honeypot take longer to complete than those on the thermostat. The SYN and FIN scans on the honeypot take almost 3 times longer than the thermostat scan. The UDP scan also takes significantly longer, similar to those associated with the camera. An overview of all the average time differences between scans can be seen in Figure 73. The data for this experiment can be found in Table 15 in Appendix F.

The results of the Anderson-Darling tests for normality are highlighted in Table 3. If both p-values from the Anderson-Darling test are greater than 0.05, the final p-value is calculated from a t-test. Otherwise, a Mann-Whitney U test calculates the final p-value.

None of the data sets have a two-sided p-value greater than 0.05 from the Anderson-Darling test, so a Mann-Whitney U test is used. After performing the Mann-Whitney U tests on the data, none of the scans have a two-sided p-value greater than 0.05, which means they all reject the null. These are included in the 'Final p-value' column in Table 3. Therefore it can be concluded that none of the trials of response times between the two devices can be from the same population with 95% confidence. The full test results can be found in Figures 105, 106, and 107 in Appendix G.

The honeypot is slowed down to be more in line with the thermostat device query response time, but this makes the Nmap scans significantly slower. Unlike the camera, speeding up the device by providing more resources to VM or optimizing the code is not ideal because the honeypot is already intentionally slowed down to better simulate the thermostat query response time. The speed of the Nmap scans may just be how fast Honeyd reacts, and there may not be a way to make the honeypot perform more like the thermostat. Despite all the UDP ports being closed on the honeypot, the UDP scan still takes an extremely long time. Examining the output of the Nmap scans, the sending delay is not increased by too many dropped probes like in the camera scenario, it just took a long time. The only possible explanation is that Honeyd must be reacting to Nmap's scans in a way that slows everything down when the UDP ports are all closed. They have to be closed in this scenario because there are no UDP services on this device.

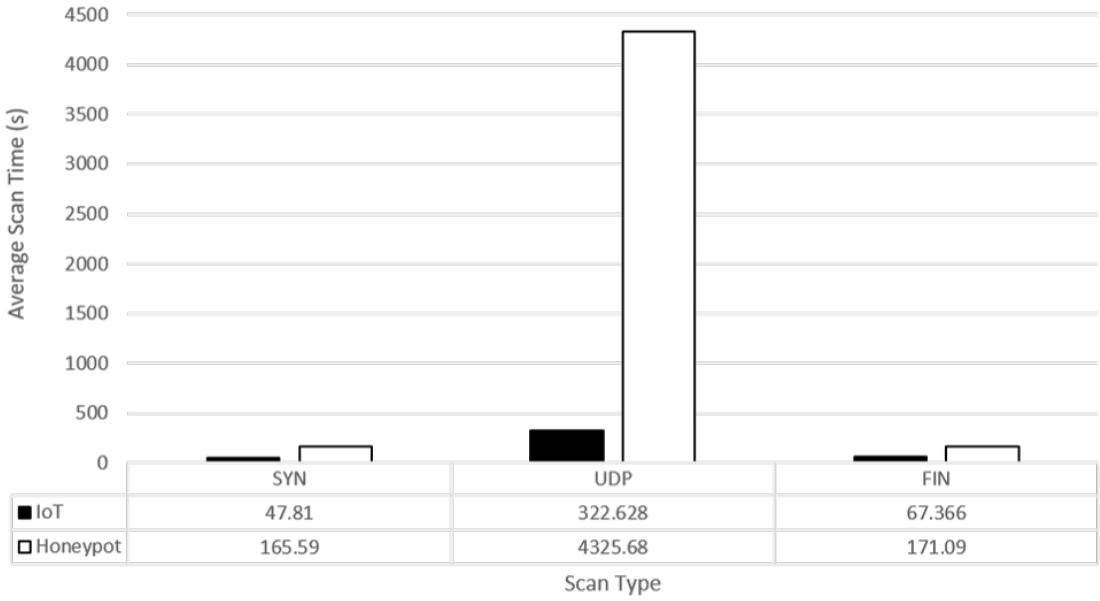


Figure 73. Average Nmap scan time for thermostat

Table 3. Proliphix thermostat and honeypot Nmap scan time Anderson-Darling results and final p-value

Scan Type	Thermostat p-value	Honeypot p-value	Both > 0.05?	Final p-value
SYN	0.4426	0.0036	No	0.0122
UDP	0.0144	0.2427	No	0.0122
FIN	0.0434	0.2431	No	0.0122

5.6.3 ezOutlet Power Outlet.

The UDP and FIN scans are significantly slower on the ezOutlet2. However, the SYN scans are quite similar on both the device and honeypot, only differing by around 45 seconds. This is close in comparison to the other devices. The UDP and FIN scans on the honeypot are very fast, considering how slow the UDP scan is on the thermostat honeypot. The breakdown of average scan times can be seen on Figure

74. The data for this experiment can be found in Table 16 in Appendix F.

The results of the Anderson-Darling tests for normality are highlighted in Table 4. If both p-values from the Anderson-Darling test are greater than 0.05, the final p-value is calculated from a t-test. Otherwise, a Mann-Whitney U test calculates the final p-value.

The FIN test has a p-value greater than 0.05 for the Anderson-Darling test, so a t-test is run for those trials. The others have a two-sided p-value less than 0.05, so a Mann-Whitney U test is used. After performing that t-test and the Mann-Whitney U tests on the data, none of the scans have a two-sided p-value greater than 0.05, and they all reject the null. These are included in the 'Final p-value' column in Table 4. Therefore it can be concluded that none of the trials of response times between the two devices could be from the same population or have the same mean with 95% confidence. The full results of the tests can be found in Figures 108, 109, and 110 in Appendix G.

It seems illogical that the UDP scan would be so fast on the outlet honeypot compared to the thermostat honeypot, despite the fact that they are configured nearly identically in the Honeyd configuration file. Neither device has any UDP services so the ports are not configured in the configuration file. There are two possibilities as to why it was so much faster. First, the thermostat honeypot has a 1 second delay added to bring it more in line with the Proliphix thermostat. This additional delay could have been enough to necessitate Nmap increasing the transmission delay between each of its queries. Another potential reason might also be the amount of data sent. The thermostat sends an HTML page with many more lines of code than the outlet. In comparison, the outlet sends only two lines of HTML code for querying the root page, while the thermostat sends 800 lines of HTML code. This equates to more packets, increasing the amount of time for each query by Nmap.

The long times for the FIN and UDP scans on the ezOutlet2 are associated with how the device responds to Nmap's test scripts. They either come too quickly or in such a way that the device reacts much slower than the other IoT devices. To bring the honeypot more in line, artificial delays would need to be instituted to react to the unique type of queries that Nmap sends for these types of scans. This would prove to be challenging, because these delays could not affect normal operation since the power outlet honeypot already responds slower than the ezOutlet2.

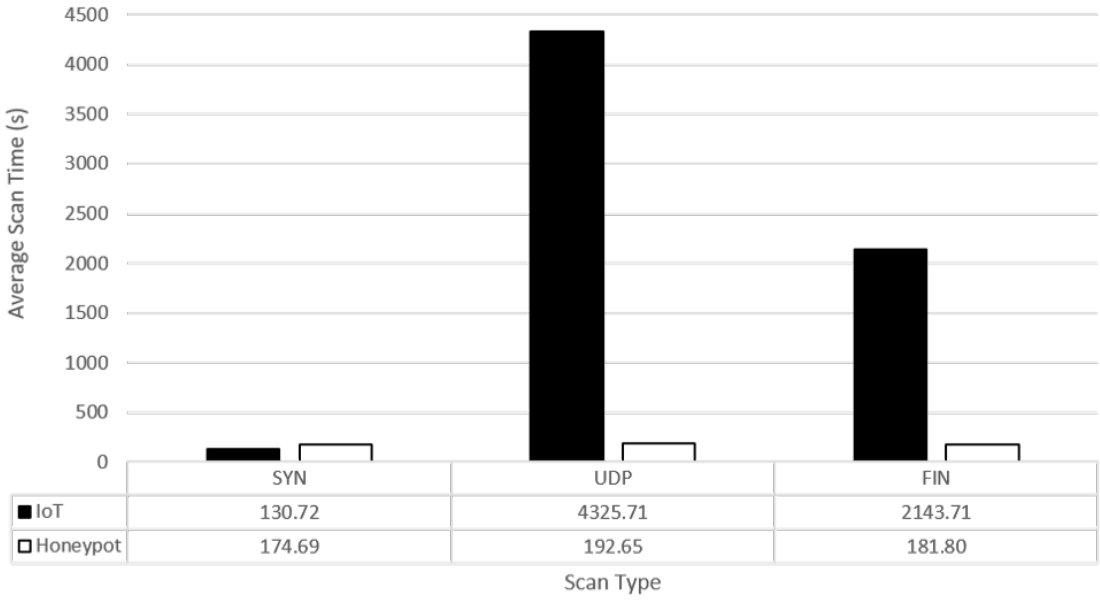


Figure 74. Average Nmap scan time for power outlet

Table 4. ezOutlet2 power outlet and honeypot Nmap scan time Anderson-Darling results and final p-value

Scan Type	Outlet p-value	Honeypot p-value	Both > 0.05?	Final p-value
SYN	0.1938	0.0037	No	0.0303
UDP	0.5177	0.0008	No	1.2855E-09
FIN	0.2151	0.1384	Yes	0.0122

5.7 Metric 6 - Nmap Scan Difference

5.7.1 TITAThink Camera.

The main component of the camera and the honeypot, the services, are all identical. All TCP and UDP services are properly shown for both devices in the Nmap scan. The services displayed are TCP ports 80, 554, and 49152 and UDP ports 443, 990, 1900, 1901, 3702, 16896, 18676, 19956, 22986, 30697, 32772, and 32777. The manufacturer, Shenzhen Ogemray Technology Co., is tied to the MAC Address and is properly shown in all five trials of the three scans. The OS does not show up properly for any of the scans. Nmap lists the camera as Linux 2.6.9-2.6.33 in the SYN and FIN scans, but it has no OS for the UDP scans. The honeypot has no OS for the SYN and FIN scans and appears as a D-Link DES-1210 for the UDP scan.

The services and manufacturer show up properly because this is one of the main selling points of Honeyd, its ability to simulate specific services with its own IP and MAC address. The difference in OS is to be expected since the older version of Honeyd uses old Nmap fingerprints, not the ones used by the latest version. To rectify this issue, Honeyd's source code would need to be changed to accommodate the new format for Nmap scan fingerprints. It is also possible that using an older version of Nmap might produce results that are more similar to each other. However,

current level 2 and 3 users are likely to be using the latest version of the software. Another possibility is editing Honeyd's own fingerprint database to add a fingerprint that is identical to the one scanned from the actual device. Prior to this thesis, different methods of attempting to change the fingerprint database in Honeyd were unsuccessful, as a result, this would also require some changes to Honeyd's source code.

5.7.2 Proliphix Thermostat.

The thermostat only has one TCP service, HTTP on TCP port 80. This service properly shows up for both the SYN and FIN scans, and no service correctly shows on the UDP scan on the honeypot. The OS is not exactly correct, but closer than the camera. The thermostat shows up as "D-Link DPR-1260 print server", and the aggressive guess for the honeypot is "D-Link DP-300U", but its second guess is "D-Link DPR-1260 print server". This shows up for both the SYN and FIN scans, and no OS is detected in the UDP scan for both the thermostat and honeypot. This is considered a success, since Nmap sees the OS scan as a 'guess', and it does guess the same OS for both systems, albeit in a different order. Finally, the manufacturer "Proliphix" is tied to the MAC address and is properly identified in all scans.

Fortunately, the personality chosen for the honeypot had a similar fingerprint to the thermostat despite being based off the old Nmap fingerprint format. As stated in the previous section, the best way to potentially improve the OS scanning is to update Honeyd with the latest Nmap fingerprint format.

5.7.3 ezOutlet Power Outlet.

Like the thermostat, the ezOutlet2 only has one service, HTTP on TCP port 80. This service is properly displayed in both the Nmap SYN and FIN scans. In addition,

no services are properly detected on the UDP scan. The OS is not detected correctly in any test. For the ezOutlet2, the OS "IBM OS/2" is detected on the SYN and FIN scans, and no OS is detected on the UDP scans. For the honeypot, no OS is detected on the SYN and FIN scans, and the OS "Brother HL-2700CN printer" is detect on the UDP scan. Finally, the manufacturer "Mega System Technologies" is tied to the MAC address and is properly identified in all scans.

As with the other tests, the OS discrepancy is mostly due to Honeyd using the old Nmap fingerprint format as well as an outdated OS fingerprint database. The way to rectify this would be to modify Honeyd's source code updating the OS personality to Nmap's new format, as well as updating the OS fingerprint database.

5.8 Summary

In this chapter, each metric that is a product of the experiments highlighted in Chapter IV has their results analyzed and discussed. Each section tries to highlight the results of the experiments as well as rationalize the outcome of the tests.

VI. Conclusions

6.1 Introduction

This chapter summarizes the findings of this research, highlights its significance, and discusses future areas of research. Section 6.2 provides concluding thoughts on the research findings. Section 6.3 discusses the significance of this research. Section 6.4 discusses some of the limitations of this research. Section 6.5 discusses the implementation and scalability of this research. Section 6.6 discusses potential future avenues for this research.

6.2 Research Conclusions

In broad terms, this research was successful in creating functioning honeypots that have scripts which provide the appearance of various working IoT devices. The hypothesis laid out in Chapter I is confirmed. Honeyd can create a near copy of real IoT devices. Many of the tests performed provide results that may be contrary to the idea that Honeyd can create perfect copies of IoT devices. The following sections review the questions that this research tried to answer and whether this necessarily means that Honeyd cannot be used to create viable IoT honeypots.

6.2.1 Response Time.

The initial question posed in Chapter IV was: Do Honeyd IoT honeypots transmit faster or slower than the actual IoT device they are simulating? Based on the results in Chapter V, the data partially supports this. The TITAThink camera and ezOutlet2 are both faster than the honeypots, and the honeypot is faster than the Proliphix thermostat. The easy answer would be to slow down honeypots that were faster with delays, but the delay is not always deterministic and there is some variances.

Chapter V provides potential solutions to speed up honeypots, but those solutions are not necessarily easy to implement.

When creating a honeypot, it should be as similar to the device as possible. So ideally, the honeypot should not be faster or slower than the device it is simulating. Even though the honeypots commonly had a statistically-significant deviance from having the same response time as the real device, the observed times are still very small. For a single user, the response times in one trial were:

	IoT Device	Honeypot
TITAThink Camera	0.018 s	0.092 s
Proliphix Thermostat	1.22 s	1.20 s
ezOutlet2	0.004 s	0.027 s

The devices with the largest difference in time differ on the order of fractions of a second. To an attacker interacting with these devices, it would be unnoticeable. This does not even take into account the fact that an IoT device is not always accessed via Ethernet on the same LAN. It can be accessed from thousands of miles away or even through WiFi. The distance an attack comes from or the medium through which an attack travels also affects response time and could increase it considerably. In that case, it would obscure the true speed of a device even more to an attacker, making the minute differences in time negligible. So even though statistically they may seem different, in a real world scenario of trying to fool an attacker, the time differences are more than adequate.

6.2.2 Data Similarity.

The initial question posed in Chapter IV is: Are IP packets sent by Honeyd IoT honeypots identical to the ones sent by the actual IoT device they are simulating? Based on the results in Chapter V, the data partially supports this. The ability for

Honeyd to act as its own web server goes a long way in being authentic without having to alter HTML code. If this were not the case, and it could only print raw HTML code, then no pictures could ever be sent unless a secondary web server was initiated and the HTML code changed to reflect that. In these experiments, the main differences displayed were when packets were dropped by the actual IoT devices because of too many connections. If an attacker were trying to interact with a honeypot they would be trying to interact with it in two ways:

1. Personally interacting with the device to gather information, run an exploit, etc.

2. Trying to deny service to a device through a Denial of Service (DoS) Attack.

In the first case, a user would probably only have one connection with the device and would not even notice that it does not drop packets like the physical version. In the second case, the user would be flooding a honeypot with requests to try and deny service. If the honeypot could handle more traffic, this might actually be beneficial. An attacker would be wasting even more time and resources trying to shut down a device that is not only more resilient than the original IoT device, but one that is not even real. So in a way, the difference in data from dropped packets, while showing that a honeypot is not exactly the same as its IoT device, may actually be a benefit in a real world scenario.

The only other case of differing data was the one line that would randomly change its value in the Proliphix Thermostat as stated in Chapter V. While this is a difference, it is negligible and would likely be virtually unnoticed by an attacker. Even if they were combing through the HTML code, the value itself is not incorrect. Sometimes it is the value reported by the honeypot, but rarely it is not. Again, in a real world scenario this would not be problem.

The packet headers, on the other hand, can pose a significant risk to the authenticity of a honeypot. TTL is one of the primary ways an attacker might be tipped off that a device is actually a honeypot. A device of the same product line has the same TTL across the board. If an attacker knows this value and it differs from the honeypot they are interacting with, this can be a red flag. It is not ideal that the TTL was different between the ezOutlet2 and its honeypot. Even so, an attacker would need to know the TTL of the exact device they are interacting with to be 100% certain they are interacting with a honeypot, making this less of an issue.

6.2.3 Nmap Scans.

The initial question posed in Chapter IV was: Can Honeyd IoT honeypots produce identical Nmap scan results to the actual IoT device they are simulating? Based on the results in Chapter V, the data does not support this. Honeyd is good at replicating the services and manufacturer, but struggled with the OS emulation. While important, the OS emulation is very hard to replicate in Honeyd's current state. Due to the limitations that come from using Nmap's old fingerprint format, the OS is rarely identified correctly. It is not even possible to choose the same OS detected on the real device because they usually did not exist in Honeyd's database. Furthermore, the large variability that appeared in scan times has no easy fix. This was especially evident in cases like the ezOutlet2, where the honeypot is slower than the IoT device in regular queries but the Nmap scans on the honeypot are faster than the IoT device. Adding an artificial delay in the code to make the Nmap scan times more similar makes the difference in query response time larger.

These differences may go unnoticed by a level 1 or 2 user. Unless they have the Nmap scan results of the exact device with which they believe to be interacting, they would have no comparison to determine whether the scan results are correct or too

slow. However, a level 3 user would probably have a large database of scan results whereby to compare. In this regard, IoT honeypots made by Honeyd may not be adequate for fooling Nmap unless Honeyd is updated to work with the latest version of Nmap.

6.3 Research Significance

Previous IoT honeypot research highlighted in Chapter II dealt with the simulation of specific services and the network traffic of IoT devices [32] [33] or non-specific IoT devices [34]. The research in this thesis tries to recreate as close as possible the appearance and function of actual IoT devices to determine whether or not Honeyd is a suitable framework within which to simulate these devices. It provides the ability to see how Honeyd is capable of simulating web IoT devices that have static pages of information, such as the Proliphix thermostat and ezOutlet2, and how it may not be optimal for simulating web IoT devices that have a dynamically updating page, such as the TITAThink camera. In addition, this thesis looks at the various parts of an IoT device (data, headers, etc.) to determine how closely it can resemble the real device. It also explores common tools used for reconnaissance to see if they have the capability of fooling users who utilize those tools as well. This research could be used by defense agencies and companies who seek to delay or examine attackers on their own networks by making convincing honeypots of IoT devices, one of the most prevalent devices that make up the internet today.

6.4 Research Limitations

One of the main limitations of this research is the age and prevalence of the devices used. The Proliphix thermostat is over 10 years old [40] and while it can still be found on IoT search engines like Shodan, its age means that its use only decreases over time.

The company TITAThink is certainly not the most popular IoT camera maker and could probably be classified as obscure. While the ezOutlet2 is the most purchased IoT power switch on Amazon, it is still a very niche item that is not commonplace in most homes. When people think of popular IoT devices they think of the Nest thermostat and camera [50] or the Ring video doorbell [51]. While these devices are more popular in the US, older electronic devices that have no use in the US are shipped to developing countries. In 2005, 400,000 pieces of old computer hardware were offloaded to Nigeria every month. It is estimated that 75% of it was non-functioning, but devices that are functioning are used within developing nations such as Nigeria [52]. This includes IoT devices as they become more prevalent. Despite this, these devices provide a proof of concept of the ability of Honeyd to simulate any IoT device that uses TCP port 80.

Another limitation is the age of Honeyd used in this research. The version of Honeyd used in this research, 1.5c, was last updated by its author, Niels Provos, in 2007 [53]. The latest release version, 1.6d, was updated in 2013 [54], and this version had stability problems when it was initially tried for this research. This limits its ability to properly simulate modern day web features, because it was written in a day when web pages were very simple. It also lacks the ability to have correct responses for modern versions of programs like Nmap.

Despite these limitations, this research still provides a good framework that could be translated onto different types of devices. Simple convincing web-based IoT honeypots can be created with relative ease, and this research shows that they can be made to look convincing to an attacker.

6.5 Scalability

This research can be implemented using any machine running Ubuntu 12 or with the capabilities to create a VM that does. More processing power and RAM increases scalability. The VM containing the honeypots in this research is given 1 processor core and 4GB of RAM. It is able to simultaneously run three IoT honeypots with ease. Honeyd is capable of completely filling an IP space by simply configuring more honeypots in the configuration file. The honeypots in this research do not drop any packets with an increased number of queries or users. It is likely that this hardware is also capable of completely filling an IP space with honeypots and would only notice performance drops when multiple users are simultaneously interacting with the honeypots. Twenty simultaneous users do not cause any dropped packets and that is an unlikely real life scenario. If that scenario did occur and packets were dropped, that would be acceptable as stated in Section 6.2.2. The attacker is wasting time trying to deny service on a fake device.

6.6 Future Work

In Chapter IV there is a recurring theme about how the current version of Honeyd is limited in how it can simulate an operating system. It also has limitations in its ability to lock out a certain number of users. One area of research could explore the possibility of updating Honeyd to incorporate the latest Nmap fingerprint format and updating its fingerprint database to the latest nmap-os-db [6]. Additional research could also examine if there is a way to get Honeyd itself, or a script run by Honeyd, to limit the number of users accessing a honeypot. This would enable the honeypots to more closely resemble the actual device.

Honeycomb is a new piece of software published by Cymmetria that is able to create honeypots using Docker as a container to run specific services for those honey-

pots [55]. This is not the same Honeycomb from Kreibich and Crowcroft discussed in Chapter II [36]. Their aim is to be a one stop shop for various honeypots that can be quickly and easily deployed. It is a very powerful piece of software, but it is so new that there are few services for it. It would be interesting to see how well it is able to create IoT honeypots based on the metrics in this thesis. It would be especially interesting to see how it reacts to network scanners like Nmap, as it has no claims to have OS personalities and network stack simulation like Honeyd.

Finally, another avenue of research would be to examine how well Honeyd can create honeypots of IoT devices that use a different service other than HTTP. With so many IoT devices interacting with the user through the cloud and mobile applications [12] [13], is it possible to simulate these services using an old but powerful piece of software like Honeyd?

6.7 Chapter Summary

This chapter highlights the concluding thoughts on the results of the research in this thesis. Even though the research highlights differences in data between the actual devices and the honeypot, it shows that the honeypots could still be a potentially powerful tool in a real scenario. The significance and limitations of the research are discussed and then future areas of research are suggested.

Appendix A. Honeyd Configuration Code

iotHoneyd.conf

```
#set default actions
create default
set default default tcp action block
set default default udp action block
set default default icmp action block

#TITAThink Camera
create titacamera
set titacamera personality "Linux 2.3.28-33"
set titacamera default tcp action reset
add titacamera tcp port 80 "TitaCamera/camera_web.sh"
add titacamera tcp port 554 open
add titacamera tcp port 49152 open
add titacamera udp port 443 filtered
add titacamera udp port 990 filtered
add titacamera udp port 1900 filtered
add titacamera udp port 1901 filtered
add titacamera udp port 3702 open
add titacamera udp port 16896 filtered
add titacamera udp port 18676 filtered
add titacamera udp port 19956 filtered
add titacamera udp port 22986 filtered
add titacamera udp port 30697 filtered
add titacamera udp port 32772 filtered
add titacamera udp port 32777 filtered

#Proliphix Thermostat
create proliphixthermostat
set proliphixthermostat personality "D-Link Print Server"
set proliphixthermostat default tcp action reset
#Regular Thermostat Interface when Internet is available
#add proliphixthermostat tcp port 80 "ProliphixThermostat/
   thermostat_web.sh"
#Thermostat Interface when no internet is available
add proliphixthermostat tcp port 80 "ProliphixThermostat/
   thermostat_web_nointernet.sh"

#ezOutlet2 Power Outlet
create ezoutlet
```

```
set ezoutlet personality "IBM OS/2 Warp 4.0"
set ezoutlet default tcp action reset
add ezoutlet tcp port 80 "ezOutlet/outlet_web.sh"

#Assign MAC and IP addresses
set titacamera ethernet "7C:DD:90:B0:22:82"
bind 192.168.0.150 titacamera
#dhcp titacamera on eth0
set proliphixthermostat ethernet "00:11:49:00:62:46"
bind 192.168.0.151 proliphixthermostat
#dhcp proliphixthermostat on eth0
set ezoutlet ethernet "00:03:EA:0E:11:67"
bind 192.168.0.152 ezoutlet
#dhcp ezoutlet on eth0
```

startHoney.sh

```
#!/bin/bash

#Bash script to begin running Honeyd
sudo honeyd -d -u 0 -g 0 -f iotHoneyd.conf
```

Appendix B. TITAThink Camera Honeypot Code

camera.py

```
#!/usr/bin/python

from PIL import Image
from PIL import ImageFont
from PIL import ImageDraw
from datetime import datetime, timedelta
import subprocess
import os
import time
import shutil

#IP ADDRESSES
#Change this to match the IP address of the titacamera honeypot in iotHoneyd.conf
honeypotIP = '192.168.0.150'

#Get the current date time and put it in the right format for the photo in drawImage
def getTimeDate():
    now = datetime.now()
    #Uncomment this line to have the time only update every 6 seconds. This was an old piece of code because some cameras only have
    #now = now.replace(second=(now.second - now.second % 6))
    return now.strftime("%a %b  %d %H:%M:%S %Y")

#Draw a date and time overlay on top of a camera feed photo
def drawImage():
    img = Image.open("player.png")
    draw = ImageDraw.Draw(img)
    font = ImageFont.truetype("images/Courier-Bold.ttf", 36)

    draw.text((0, 0),getTimeDate(),(255,255,255), font=font)
    draw.text((0, 30),"East Flightline", (255,255,255), font=font)
    img.save('player.png')
    #subprocess.call(['chmod', '666', 'flightline.jpg'])

#Will get the filename of image to use and will write that image to player.png
def getImage():
    #Get the current date time and then completely strip out the date for comparison
    dt = datetime.now()
    dtString = dt.strftime("%H-%M-%S")
    time = datetime.strptime(dtString, "%H-%M-%S")
    #print time
    #Default image is midnight
    image = "00-00-00.jpg"
    #loop through each file in the camerafeeds folder
```

```
44      #Filename format is HH:MM:SS
45      for filename in sorted(os.listdir('cameraFeeds')):
46    #Strip off file extension
47          fileTimeStr = filename[:-4]
48          fileTime = datetime.strptime(fileTimeStr, "%H-%M-%S")
49          #print fileTime
50          #If the current time is larger than the time of the file, then
      make that the new image.
51          #Eventually it will get to a time it is not greater than, it
      will check which it is closer to and then break
52          if time > fileTime:
53              image = filename
54          else:
55              td1 = dt - datetime.strptime(image[:-4], "%H-%M-%S")
56              #print td1
57              #Create datetime of 30minutes. If the difference in time
      between the image selected in the previous iteration is greater than
       30 minutes, choose the image in this iteration
58        #Half t for each time the number of photos is doubled
59              t = "00-30-00"
60              cmpTime = datetime.strptime(t, "%H-%M-%S")
61              if td1 < timedelta(minutes=30):
62                  image = filename
63              break
64
65      return image
66 #Change the player.jpg image to the correct image based on the time of
      day
67 shutil.copyfile(str('cameraFeeds/' + getImage()), 'player.png')
68 #Uncomment this if you want the image to be overlayed with the current
      date and time
69 #drawImage()
70 #Open html files to write to the client
71 f = open('home.html', 'rb')
72 #print each line in the home file to be output to the client
73 for line in f:
74      print line.rstrip("\n")
```

camera_web.sh

```
1 #!/bin/sh
2 REQUEST=""
3 DIRPATH='/home/lstafira/Desktop/python-web-server/TitaCamera'
4 #Navigate to corrent directory
5 cd $DIRPATH
6 #Read in all the incoming packets
7 while read name
8 #If there is a password in the packet, pull it out, decode it from
      base64 and save it to password.txt
9 password=`echo "$name" | grep "Authorization:"`
10 p=`echo $password | cut -d " " -f 3`
11 if [ ! -z "$p" ] ; then
12   echo `echo "$p" | base64 --decode` >> captured/password.txt
```

```
  chmod 666 captured/password.txt
fi
#If the packet has no words, break, but if it does, get the GET line and
     extract the second parameter, which is the requested file
do
  LINE=`echo "$name" | egrep -i "[a-z:]"`
  if [ -z "$LINE" ]
  then
    break
  fi
  NEWREQUEST=`echo "$name" | grep "GET"`
  y=`echo $NEWREQUEST | cut -d " " -f 2`
  echo "$NEWREQUEST" >> captured/requests.txt
  chmod 666 captured/requests.txt
#For each requested file, send the appropriate response header and then
    print the correct html file
  if [ '/' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    DATE=$(python timedate.py)
    cat << _eof_
HTTP/1.1 302 Redirect
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache
Location: http://192.168.0.150/default.asp

<html><head></head><body>
    This document has moved to a new <a href="http://192.168.0.150/
    default.asp">location</a>.
    Please update your documents to reflect the new location.
</body></html>

_eof_
#default.asp
  elif [ '/default.asp' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    DATE=$(python timedate.py)
    cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
python htmlprint.py redirect.html
#/form/default
  elif [ '/form/default' = "$y" ] ; then
    REQUEST=$NEWREQUEST
```

```
62      DATE=$(python timedate.py)
63      cat << _eof_
64 HTTP/1.0 302 Redirect
65 Date: $DATE
66 Server: Webs
67 Content-Type: text/html
68 Pragma: no-cache
69 Cache-Control: no-cache
70 Location: http://192.168.0.150/en/login.asp
71
72 <html><head></head><body>
73      This document has moved to a new <a href="http://192.168.0.150/en/
       login.asp">location</a>.
74      Please update your documents to reflect the new location.
75 </body></html>
76
77 _eof_
78 #/en/login.asp
79    elif [ '/en/login.asp' = "$y" ] ; then
80      REQUEST=$NEWREQUEST
81      DATE=$(python timedate.py)
82      cat << _eof_
83 HTTP/1.0 200 OK
84 Date: $DATE
85 Server: Webs
86 Content-Type: text/html
87 Pragma: no-cache
88 Cache-Control: no-cache
89
90 _eof_
91 python htmlprint.py login.html
92 #/en/images/login.png
93    elif [ '/en/images/login.png' = "$y" ] ; then
94      REQUEST=$NEWREQUEST
95      DATE=$(python timedate.py)
96      cat << _eof_
97 HTTP/1.0 200 OK
98 Date: $DATE
99 Server: Webs
100 Content-Type: text/html
101 Pragma: no-cache
102 Cache-Control: no-cache
103
104 _eof_
105 cat images/login.png
106 #/favicon.ico
107    elif [ '/favicon.ico' = "$y" ] ; then
108      REQUEST=$NEWREQUEST
109      DATE=$(python timedate.py)
110      cat << _eof_
111 HTTP/1.1 401 Unauthorized
112 Date: $DATE
```

```
WWW-Authenticate: Basic realm="TT520PW"
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

<html><head><title>Document Error: Unauthorized</title></head>
    <body><h2>Access Error: Unauthorized</h2>
when trying to obtain <b>/favicon.ico</b><br><p>Access to this document
    requires a User ID!</p></body></html>

_eof_
#/form/liveRedirect?lang=en
  elif [ '/form/liveRedirect?lang=en' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    DATE=$(python timedate.py)
    cat << _eof_
HTTP/1.0 302 Redirect
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache
Location: http://192.168.0.150/en/player/flash_hd.asp

<html><head></head><body>
    This document has moved to a new <a href="http://192.168.0.150/en/
    flash_hd.asp">location</a>.
    Please update your documents to reflect the new location.
</body></html>

_eof_
#/en/player/flahs_hd.asp
  elif [ '/en/player/flash_hd.asp' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    DATE=$(python timedate.py)
    cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
python camera.py
#/live/0/mjpeg.jpg
  elif [ '/live/0/mjpeg.jpg' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    DATE=$(python timedate.py)
    cat << _eof_
HTTP/1.0 200 OK
```

```
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
cat player.png
#/en/images/frame_left_top.png
   elif [ '/en/images/frame_left_top.png' = "$y" ] ; then
     REQUEST=$NEWREQUEST
     DATE=$(python timedate.py)
     cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
cat images/frame_left_top.png
#/images/flash_hd_right_top.png
   elif [ '/images/flash_hd_right_top.png' = "$y" ] ; then
     REQUEST=$NEWREQUEST
     DATE=$(python timedate.py)
     cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
cat images/flash_hd_right_top.png
#/images/flash_hd_bottom.png
   elif [ '/images/flash_hd_bottom.png' = "$y" ] ; then
     REQUEST=$NEWREQUEST
     DATE=$(python timedate.py)
     cat << _eof_
HTTP/1.0 200 OK
Date: $DATE
Server: Webs
Content-Type: text/html
Pragma: no-cache
Cache-Control: no-cache

_eof_
cat images/flash_hd_bottom.png
#/en/css/top.css
   elif [ '/en/css/top.css' = "$y" ] ; then
```

```
215     REQUEST=$NEWREQUEST
216     DATE=$(python timedate.py)
217     cat << _eof_
218 HTTP/1.0 200 OK
219 Date: $DATE
220 Server: Webs
221 Content-Type: text/html
222 Pragma: no-cache
223 Cache-Control: no-cache
224
225 _eof_
226 cat top.css
227 #/en/main.asp
228   elif [ '/en/main.asp' = "$y" ] ; then
229     REQUEST=$NEWREQUEST
230     DATE=$(python timedate.py)
231     cat << _eof_
232 HTTP/1.0 401 Unauthorized
233 Date: $DATE
234 Server: Webs
235 Content-Type: text/html
236 Pragma: no-cache
237 Cache-Control: no-cache
238 WWW-Authenticate: Basic realm="TT520PW"
239
240 _eof_
241 python error.py "$y"
242 #/en/player/mjpeg_hd.asp?stream=0
243   elif [ '/en/player/mjpeg_hd.asp?stream=0' = "$y" ] ; then
244     REQUEST=$NEWREQUEST
245     DATE=$(python timedate.py)
246     cat << _eof_
247 HTTP/1.0 200 OK
248 Date: $DATE
249 Server: Webs
250 Content-Type: text/html
251 Pragma: no-cache
252 Cache-Control: no-cache
253
254 _eof_
255 python camera.py
256 #/en/player/mjpeg_hd.asp?stream=1
257   elif [ '/en/player/mjpeg_hd.asp?stream=1' = "$y" ] ; then
258     REQUEST=$NEWREQUEST
259     DATE=$(python timedate.py)
260     cat << _eof_
261 HTTP/1.0 200 OK
262 Date: $DATE
263 Server: Webs
264 Content-Type: text/html
265 Pragma: no-cache
266 Cache-Control: no-cache
```

```
267
268 _eof_
269 cat home_small.html
270   elif [ -z "$y" ] ; then
271     REQUEST=$NEWREQUEST
272 #Otherwise, The page does not exist
273   else
274     REQUEST=$NEWREQUEST
275     DATE=$(python timedate.py)
276     cat << _eof_
277 HTTP/1.0 401 Unauthorized
278 Date: $DATE
279 Server: Webs
280 Content-Type: text/html
281 Pragma: no-cache
282 Cache-Control: no-cache
283 WWW-Authenticate: Basic realm="TT520PW"
284
285 _eof_
286 python error.py "$y"
287   fi
288 done
```

htmlprint.py

```
1 import sys
2
3
4 #Open the file and print out the bytes line by line
5 f=open(sys.argv[1], 'rb')
6
7 for line in f:
8   print(line.rstrip("\n"))
9 #close the file
10 f.close()
```

timedate.py

```
1 #!/usr/bin/python
2 from datetime import datetime
3
4 #Get the current date and time and print it out. This is used by the
    camera_web.sh script to get the current date and time for headers.
5 def getTimeDate():
6     now = datetime.now()
7     return now.strftime("%a %b %d %H:%M:%S %Y")
8
9 print getTimeDate()
```

error.py

```
1 import sys
2
3
4 #Print the error page for the camera, the 1st command line parameter is
      the file that could not be opened
5 print str("<html><head><title>Document Error: Unauthorized</title></head
      >\r")
6 print str("\t\t<body><h2>Access Error: Unauthorized</h2>\r")
7 print str("\t\twhen trying to obtain <b>" + sys.argv[1] + "</b><br><p>
      Access to this document requires a User ID!</p></body></html>\r")
8 print "\r"
```

Appendix C. Proliphix Thermostat Honeypot Code

thermostat.py

```
import sys
import calendar
import time
import datetime
#File used for getting the correct temperature and printing out the Proliphix Thermostat home page. This is the internet version, so only use this if the machine running your honeypots has internet access.

#USE PORT 8081
#Globals
city = 'dayton'
heatTemp = 68.0
coolTemp = 73.0

#using weather-api for weather pulling
from weather import Weather, Unit

#Time needs to be in miliseconds for the method used by the Proliphix thermostat
def TodayDateToMs():
    ts = calendar.timegm(time.localtime())
    return ts

if sys.argv[1] == '/' or sys.argv[1] == '/status.shtml':
  f = open('thermostat.html', 'rb')
date = datetime.date.today()

#get weather info
weather = Weather(unit=Unit.FAHRENHEIT)
location = weather.lookup_by_location(city)
condition = location.condition
temp = condition.temp

#current line number
x = 1

#Wait for 1.1 seconds, this is about the delay of the actual thermostat
time.sleep(1.1)

for line in f:
    #Line 625 in Template
    if x == 625:
  date = TodayDateToMs()
        print str("v = parseInt(\"" + str(date) + "\");")
    #Line 631 in Template
    elif x == 631:
        if temp > 70:
```

```
45              print str("avgtemp = \"73.0\"")
46          else:
47              print str("avgtemp = \"68.0\"")
48      #Line 632 in Template
49      elif x == 632:
50    if temp > 70:
51        print str("localtemp = \"73.0\"")
52    else:
53        print str("localtemp = \"68.0\"")
54      #Line 683 in Template
55      elif x == 683:
56          if temp > 70:
57              print str("printFSC(\"Cool Setting\", \"73.0\" + \"&deg;\" +
      ts,\"\");")
58          else:
59              print str("printFSC(\"Cool Setting\", \"75.0\" + \"&deg;\" +
      ts,\"\");")
60      #Line 688 in Template
61      elif x == 688:
62          if temp > 70:
63              print str("printFSC(\"Heat Setting\", \"65.0\" + \"&deg;\"+
      ts,\"\");")
64          else:
65              print str("printFSC(\"Heat Setting\", \"68.0\" + \"&deg;\"+
      ts,\"\");")
66      else:
67          #f2.write(line)
68    print line.rstrip("\n")
69      #Increment
70      x = x + 1
```

thermostat_nointernet.py

```
1  import sys
2  import calendar
3  import time
4  import datetime
5
6  #Program takes the current temperature and creates the main page of the
       thermostat. This version is for honeypots with no internet access.
       Set the desired temperature in the 'temp' value
7
8  #USE PORT 8081
9  #Globals
10 city = 'dayton'
11 #Temperature heater and cooler kick in
12 heatTemp = 68.0
13 coolTemp = 73.0
14
15 #Current weather temperature hardcoded in because of no internet.
16 temp = 70
17
18 #using weather-api for weather pulling
```

```
from weather import Weather, Unit

#Time needs to be in miliseconds for the method used by the Proliphix
    thermostat
def TodayDateToMs():
    ts = calendar.timegm(time.localtime())
    return ts

if sys.argv[1] == '/' or sys.argv[1] == '/status.shtml':
  f = open('thermostat.html', 'rb')
date = datetime.date.today()

#current line number
x = 1

#Wait for 1.1 seconds, this is about the delay of the actual thermostat
time.sleep(1.1)

for line in f:
    #Line 625 in Template
    if x == 625:
  date = TodayDateToMs()
        print str("v = parseInt(\"" + str(date) + "\");")
    #Line 631 in Template
    elif x == 631:
        if temp > 70:
            print str("avgtemp = \"73.0\"")
        else:
            print str("avgtemp = \"68.0\"")
    #Line 632 in Template
    elif x == 632:
  if temp > 70:
      print str("localtemp = \"73.0\"")
  else:
      print str("localtemp = \"68.0\"")
    #Line 683 in Template
    elif x == 683:
        if temp > 70:
            print str("printFSC(\"Cool Setting\", \"73.0\" + \"&deg;\" +
    ts,\"\");")
        else:
            print str("printFSC(\"Cool Setting\", \"75.0\" + \"&deg;\" +
    ts,\"\");")
    #Line 688 in Template
    elif x == 688:
        if temp > 70:
            print str("printFSC(\"Heat Setting\", \"65.0\" + \"&deg;\"+
    ts,\"\");")
        else:
            print str("printFSC(\"Heat Setting\", \"68.0\" + \"&deg;\"+
    ts,\"\");")
    else:
```

```
66          #f2.write(line)
67    print line.rstrip("\n")
68      x = x + 1
```

thermostat_web.sh

```
1  #!/bin/sh
2  REQUEST=""
3  DIRPATH='/home/lstafira/Desktop/python-web-server/ProliphixThermostat'
4  #Navigate to corrent directory
5  cd $DIRPATH
6  #Read in all the incoming packets
7  while read name
8  #If there is a password in the packet, pull it out, decode it from
       base64 and save it to password.txt
9  password=`echo "$name" | grep "Authorization:"`
10 p=`echo $password | cut -d " " -f 3`
11 if [ ! -z "$p" ] ; then
12   echo `echo "$p" | base64 --decode` >> captured/password.txt
13   chmod 666 captured/password.txt
14 fi
15 #If the packet has no words, break, but if it does, get the GET line and
       extract the second parameter, which is the requested file
16 do
17   LINE=`echo "$name" | egrep -i "[a-z:]"`
18   if [ -z "$LINE" ]
19   then
20     break
21   fi
22   NEWREQUEST=`echo "$name" | grep "GET"`
23   y=`echo $NEWREQUEST | cut -d " " -f 2`
24   echo "$NEWREQUEST" >> captured/requests.txt
25   chmod 666 captured/requests.txt
26 #For each requested file, send the appropriate response header and then
       print the correct html file
27   if [ '/' = "$y" ] ; then
28     REQUEST=$NEWREQUEST
29     cat << _eof_
30 HTTP/1.1 200 OK
31 Server: Ubicom/1.1
32 Connection: close
33 Cache-Control: no-cache
34
35 _eof_
36 python thermostat.py "$y"
37 #/index.shtml
38   elif [ '/index.shtml' = "$y" ] ; then
39     REQUEST=$NEWREQUEST
40     cat << _eof_
41 HTTP/1.1 401 Unauthorized
42 content-length: 26
43 Server: Ubicom/1.1
44 www-authenticate: Basic realm="tstat"
```

```
Cache-Control: no-cache

401 Authorization Required
_eof_
#/status.shtml
   elif [ '/status.shtml' = "$y" ] ; then
     REQUEST=$NEWREQUEST
     cat << _eof_
HTTP/1.1 200 OK
Server: Ubicom/1.1
Connection: close
Cache-Control: no-cache

_eof_
python thermostat.py "$y"
#else, the page is unauthorized
   elif [ -z "$y" ] ; then
     REQUEST=$NEWREQUEST
   else
   cat << _eof_
HTTP/1.1 401 Unauthorized
content-length: 26
Server: Ubicom/1.1
www-authenticate: Basic realm="tstat"
Cache-Control: no-cache

401 Authorization Required
_eof_
   fi
done
```

thermostat_web_nointernet.sh

```
#!/bin/sh
REQUEST=""
DIRPATH='/home/lstafira/Desktop/python-web-server/ProliphixThermostat'
#Navigate to corrent directory
cd $DIRPATH
#Read in all the incoming packets
while read name
#If there is a password in the packet, pull it out, decode it from
    base64 and save it to password.txt
password=`echo "$name" | grep "Authorization:"`
p=`echo $password | cut -d " " -f 3`
if [ ! -z "$p" ] ; then
  echo `echo "$p" | base64 --decode` >> captured/password.txt
  chmod 666 captured/password.txt
fi
#If the packet has no words, break, but if it does, get the GET line and
     extract the second parameter, which is the requested file
do
  LINE=`echo "$name" | egrep -i "[a-z:]"`
  if [ -z "$LINE" ]
```

```
19    then
20      break
21    fi
22    NEWREQUEST=`echo "$name" | grep "GET"`
23    y=`echo $NEWREQUEST | cut -d " " -f 2`
24    echo "$NEWREQUEST" >> captured/requests.txt
25    chmod 666 captured/requests.txt
26  #For each requested file, send the appropriate response header and then
      print the correct html file
27    if [ '/' = "$y" ] ; then
28      REQUEST=$NEWREQUEST
29      cat << _eof_
30  HTTP/1.1 200 OK
31  Server: Ubicom/1.1
32  Connection: close
33  Cache-Control: no-cache
34
35  _eof_
36  python thermostat_nointernet.py "$y"
37  #/index.shtml
38    elif [ '/index.shtml' = "$y" ] ; then
39      REQUEST=$NEWREQUEST
40      cat << _eof_
41  HTTP/1.1 401 Unauthorized
42  content-length: 26
43  Server: Ubicom/1.1
44  www-authenticate: Basic realm="tstat"
45  Cache-Control: no-cache
46
47  401 Authorization Required
48  _eof_
49  #/status.shtml
50    elif [ '/status.shtml' = "$y" ] ; then
51      REQUEST=$NEWREQUEST
52      cat << _eof_
53  HTTP/1.1 200 OK
54  Server: Ubicom/1.1
55  Connection: close
56  Cache-Control: no-cache
57
58  _eof_
59  python thermostat_nointernet.py "$y"
60  #else print the unauthorized page
61    elif [ -z "$y" ] ; then
62      REQUEST=$NEWREQUEST
63    else
64    cat << _eof_
65  HTTP/1.1 401 Unauthorized
66  content-length: 26
67  Server: Ubicom/1.1
68  www-authenticate: Basic realm="tstat"
69  Cache-Control: no-cache
```

```
70
71 401 Authorization Required
72 _eof_
73   fi
74 done
```

htmlprint.py

```
1 import sys
2
3 #Open file and print out the bytes for output to the client
4 f=open(sys.argv[1], 'rb')
5
6 for line in f:
7   print(line)
8 #close the file
9 f.close()
```

error.py

```
1 #print the 404 page
2 print str("404: File not found\r")
```

Appendix D. ezOutlet2 Power Outlet Code

outlet.py

```
from __future__ import print_function
import sys
import calendar
import time
from datetime import datetime

#GLOBALS
#Change these IP addresses to match the gateway and subnet mask of the network the honeypot is on and the honypot's IP from the Honeyd configuration file
gatewayIP = '192.168.0.1'
honeypotIP = '192.168.0.152'
subnetMask = '255.255.255.0'
#datetime
dt = datetime.now()

#Only argument is which file to file, 1 is the status page 2 is the network page
choice = int(sys.argv[1])

if choice == 1:
    #Open status file
    f = open('status.htm', 'rb')
    x = 1
    #Write the whole file, changing the date and time
    for line in f:
        if x == 31:
      print("<tr><td>IP  Address:</td><td id=z>" + honeypotIP + "</td><td>DNS Mode:</td><td id=z>\r")
        elif x == 36:
      print("d.writeln(\"</td></tr><tr><td>Mask:</td><td id=z>" + subnetMask + "</td><td>\"+(a?\"DHCP Assign:</td><td id=z>" + gatewayIP + "\":\"DNS 1:</td><td id=z>8.8.8.8\")+\"</td></tr>\");\r")
        elif x == 37:
      print("d.writeln(\"<tr><td>Gateway:</td><td id=z>" + gatewayIP + "</td><td>\"+(a?\"DNS 1:</td><td id=z>8.8.8.8\":\"DNS 2:</td><td id=z>208.67.222.222\")+\"</td></tr>\");\r")
        #Line 42 in Template
        elif x == 42:
            print("<tr><td>Time / Zone:</td><td id=z>" + dt.strftime("%H:%M") + " / GMT-5:00</td><td>Connected:</td><td id=z>No</td></tr>\r")
        #Line 43 in Template
        elif x == 43:
            print("<tr><td>Date:</td><td id=z>" + dt.strftime("%Y/%m/%d(%a")[:-1] + ")</td><td>Finder:</td><td id=z>Yes</td></tr>\r")
        else:
```

```
37              print(line, end='')
38          x = x + 1
39      f.close()
40
41
42  if choice == 2:
43      #open network file
44      f = open('network.htm', 'rb')
45      x = 1
46      #Write the whole file, changing the IP addresses
47      for line in f:
48          #Line 59
49          if x == 59:
50        print("<tr><td>IP Address:</td><td><input type=\"text\" name=\"
      ipaddr\" value=\"" + honeypotIP + "\" onblur=\"CheckIP(\'ipaddr\')
      \"></td></tr>\r")
51          elif x == 60:
52        print("<tr><td>Mask:</td><td><input type=\"text\" name=\"mask\"
      value=\"" + subnetMask + "\" onblur=\"CheckIP(\'mask\')\"></td></tr
      >\r")
53          elif x == 61:
54        print("<tr><td>Gateway:</td><td><input type=\"text\" name=\"gate\"
       value=\"" + gatewayIP + "\" onblur=\"CheckIP(\'gate\')\"></td></tr
      >\r")
55          elif x == 63:
56        print("<tr><td>DNS Assigned by DHCP:</td><td>" + gatewayIP + "</td
      ></tr>\r")
57          else:
58        print(line, end='')
59          x = x + 1
60      f.close()
```

outlet_web.sh

```
1  #!/bin/sh
2  REQUEST=""
3  DIRPATH='/home/lstafira/Desktop/python-web-server/ezOutlet'
4  #Navigate to corrent directory
5  cd $DIRPATH
6  #Read in all the incoming packets
7  while read name
8  #If there is a password in the packet, pull it out, decode it from
       base64 and save it to password.txt
9  password=`echo "$name" | grep "Authorization:"`
10 p=`echo $password | cut -d " " -f 3`
11 if [ ! -z "$p" ] ; then
12   echo `echo "$p" | base64 --decode` >> captured/password.txt
13   chmod 666 captured/password.txt
14 fi
15 #If the packet has no words, break, but if it does, get the GET line and
       extract the second parameter, which is the requested file
16 do
17   LINE=`echo "$name" | egrep -i "[a-z:]"`
```

```
18    if [ -z "$LINE" ]
19    then
20      break
21    fi
22    NEWREQUEST=`echo "$name" | grep "GET"`
23    y=`echo $NEWREQUEST | cut -d " " -f 2`
24    echo "$NEWREQUEST" >> captured/requests.txt
25    chmod 666 captured/requests.txt
26 #For each requested file, send the appropriate response header and then
       print the correct html file
27    if [ '/' = "$y" ] ; then
28      REQUEST=$NEWREQUEST
29      cat << _eof_
30 HTTP/1.1 200 OK
31 Connection: close
32 Content-Type: text/html
33 Cache-Control: no-cache
34
35 _eof_
36 cat ezOutlet2.html
37 #/menu.htm
38    elif [ '/menu.htm' = "$y" ] ; then
39      REQUEST=$NEWREQUEST
40      cat << _eof_
41 HTTP/1.1 200 OK
42 Connection: close
43 Content-Type: text/html
44 Cache-Control: no-cache
45
46 _eof_
47 cat menu.htm
48 #/status.htm
49    elif [ '/status.htm' = "$y" ] ; then
50      REQUEST=$NEWREQUEST
51      cat << _eof_
52 HTTP/1.1 200 OK
53 Connection: close
54 Content-Type: text/html
55 Cache-Control: no-cache
56
57 _eof_
58 python outlet.py 1
59 #/network.htm
60    elif [ '/network.htm' = "$y" ] ; then
61      REQUEST=$NEWREQUEST
62      cat << _eof_
63 HTTP/1.1 200 OK
64 Connection: close
65 Content-Type: text/html
66 Cache-Control: no-cache
67
68 _eof_
```

```
python outlet.py 2
#settings.htm
  elif [ '/settings.htm' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
cat settings.htm
#schedule.htm
  elif [ '/schedule.htm' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
cat schedule.htm
#list.htm
  elif [ '/list.htm' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
cat list.htm
#about.htm
  elif [ '/about.htm' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    cat << _eof_
HTTP/1.1 200 OK
Connection: close
Content-Type: text/html
Cache-Control: no-cache

_eof_
cat about.htm
#mchp.js
  elif [ '/mchp.js' = "$y" ] ; then
    REQUEST=$NEWREQUEST
    cat << _eof_
HTTP/1.1 200 OK
Connection: close
Cache-Control: no-cache
```

```
121
122 _eof_
123 cat mchp.js.download
124 #invert.cgi
125   elif [ '/invert.cgi' = "$y" ] ; then
126     REQUEST=$NEWREQUEST
127     cat << _eof_
128 HTTP/1.1 200 OK
129 Connection: close
130 Content-Type: text/html
131 Cache-Control: no-cache
132
133 _eof_
134 #change the state of the invert.cgi file
135 state=`cat invert.cgi`
136     if [ '0,0' = "$state" ] ; then
137       echo "1,0" > invert.cgi
138     else
139       echo "0,0" > invert.cgi
140     fi
141 cat invert.cgi
142 #reset.cgi
143   elif [ '/reset.cgi' = "$y" ] ; then
144     REQUEST=$NEWREQUEST
145     cat << _eof_
146 HTTP/1.1 200 OK
147 Connection: close
148 Content-Type: text/html
149 Cache-Control: no-cache
150
151 _eof_
152 #put 0,0 into invert.cgi and cat 0,0 into the page
153 echo "0,0" > invert.cgi
154 cat reset.cgi
155 #else, say the page is not found
156   elif [ -z "$y" ] ; then
157     REQUEST=$NEWREQUEST
158   else
159   cat << _eof_
160 HTTP/1.1 404 Not Found
161 Connection: close
162
163 _eof_
164 python error.py
165   fi
166 done
```

htmlprint.py

```
import sys

#get the file and print out the bytes to the client
f=open(sys.argv[1], 'rb')

for line in f:
  print(line)
f.close()
```

error.py

```
#print the 404 page
print str("404: File not found\r")
```

Appendix E. Testing and Comparison Scripts

gethttp.py

```
#!/usr/bin/python

#This page will query a specified device a specified number of times and save the HTML files and HTTP headers to a specific file
import requests
import sys
import time
from datetime import datetime
import os
import shutil
import socket
from struct import *
import subprocess

#Command line parameters
#sys.argv[1] IP Address
#sys.argv[2] Number of iterations
#sys.argv[3] Which device to scan
#sys.argv[4] Used for running multiple simultaneous connections. It indicates how to label the folder for this scan. Usually it will be a number. User 1 is 1, user 2 is 2, etc. For singular scans, the date will be used instead of this variable.
# 1 - TitaCamera
# 2 - ProliphixThermostat
# 3 - ezOutlet

#Scanning will consist of all the main pages, however many that is, as well as an unauthorized page(If applicable), and accessing a page that does not exist.

#IP Address of web site accessed
url = sys.argv[1]
#Choice of device to scan
choice = int(sys.argv[3])
if choice != 1 and choice != 2 and choice != 3:
  print "Invalid device choice, please enter: 1, 2, or 3"
  sys.exit()
#Get current date and time and format it for file naming
dt = datetime.now().strftime('%Y-%m-%d---%H:%M:%S')
#Directory Name where files will be saved
#folder = str(sys.argv[1][7:20] + " :: " + dt)
folder = str(sys.argv[1][7:20] + " :: " + sys.argv[4])
os.makedirs(folder)
os.makedirs(str(folder+'/html'))
#File where the access times for each file will be stored
logFile = open(str(folder + '/accessTimeLog.txt'), 'wb')
headFile = open(str(folder + '/HTTP Header : ' + sys.argv[1][7:20] + " :: " + dt + '.txt'), 'wb')
```

```
42
43 #Request as many times as specified by the second argument
44 for x in range(0, int(sys.argv[2])):
45     time.sleep(0.5)
46     #begin timer
47     start = time.time()
48     #make the request
49     r = requests.get(url, allow_redirects=True)
50     #stop the timer
51     end = time.time()
52     #Access time is end time - start time
53     accessTime = end - start
54     #open a file to write what was received from the request
55     file = open(str(folder + '/html/root_' + str(x+1) + '.html'), 'wb')
56     file.write(r.content)
57     #print r.content
58     file.close
59     #write access time to log file and HTTP header to header file
60     logFile.write(str(str(accessTime) + '\n'))
61     #Loop through each item in the HTTP header
62     headFile.write('Root HTTP Header:\n')
63     for item in r.headers:
64         headFile.write(str(item + '\n' + r.headers[item] + '\n'))
65
66     #Extra files to scan for on TitaCamera
67     if choice == 1:
68   #Request for main login page
69   r = requests.get(str(url+'/form/default'), allow_redirects=True)
70   file = open(str(folder + '/html/login_' + str(x+1) + '.html'), 'wb')
71   file.write(r.content)
72   file.close
73   headFile.write('Login HTTP Header:\n')
74       for item in r.headers:
75             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
76   #Request for main camera page
77   r = requests.get(str(url+'/form/liveRedirect?lang=en'),
       allow_redirects=True)
78   file = open(str(folder + '/html/main_' + str(x+1) + '.html'), 'wb')
79   file.write(r.content)
80   file.close
81   headFile.write('Main HTTP Header:\n')
82       for item in r.headers:
83             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
84   #Request for settings page that requires authentication
85   r = requests.get(str(url+'/en/main.asp'), allow_redirects=True)
86   file = open(str(folder + '/html/settings_' + str(x+1) + '.html'), 'wb'
       )
87   file.write(r.content)
88   file.close
89   headFile.write('Settings HTTP Header:\n')
90       for item in r.headers:
91             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
```

```
92  #Request for page that does not exist, should require authentication
93  r = requests.get(str(url+'/passwd'), allow_redirects=True)
94  file = open(str(folder + '/html/dnePage_' + str(x+1) + '.html'), 'wb')
95  file.write(r.content)
96  file.close
97  headFile.write('DNE HTTP Header:\n')
98      for item in r.headers:
99            headFile.write(str(item + '\n' + r.headers[item] + '\n'))
100   #Extra files to scan for on Proliphix Thermostat
101    elif choice == 2:
102 time.sleep(0.5)
103 #Request for status page, should return same as root with button
      depressed
104 r = requests.get(str(url+'/en/status.shtml'), allow_redirects=True)
105 file = open(str(folder + '/html/status_' + str(x+1) + '.html'), 'wb')
106 file.write(r.content)
107 file.close
108 headFile.write('Status HTTP Header:\n')
109     for item in r.headers:
110           headFile.write(str(item + '\n' + r.headers[item] + '\n'))
111 #Request for settings page, requires authentication
112 time.sleep(0.5)
113 r = requests.get(str(url+'/index.shtml'), allow_redirects=True)
114 file = open(str(folder + '/html/settings_' + str(x+1) + '.html'), 'wb'
      )
115 file.write(r.content)
116 file.close
117 headFile.write('Settings HTTP Header:\n')
118     for item in r.headers:
119           headFile.write(str(item + '\n' + r.headers[item] + '\n'))
120 #Request for page that does not exist, should require authentication
121 time.sleep(0.5)
122 r = requests.get(str(url+'/passwd'), allow_redirects=True)
123 file = open(str(folder + '/html/dnePage_' + str(x+1) + '.html'), 'wb')
124 file.write(r.content)
125 file.close
126 headFile.write('DNE HTTP Header:\n')
127     for item in r.headers:
128           headFile.write(str(item + '\n' + r.headers[item] + '\n'))
129   #Extra files to scan for on ezOutlet
130    elif choice == 3:
131 #Request for menu which is always present from the main page
132 r = requests.get(str(url+'/menu.htm'), allow_redirects=True)
133 file = open(str(folder + '/html/menu_' + str(x+1) + '.html'), 'wb')
134 file.write(r.content)
135 file.close
136 headFile.write('Menu HTTP Header:\n')
137     for item in r.headers:
138           headFile.write(str(item + '\n' + r.headers[item] + '\n'))
139 #Request for status page, which is present from the start and can be
      renavigated to
140 r = requests.get(str(url+'/status.htm'), allow_redirects=True)
```

```
141   file = open(str(folder + '/html/status_' + str(x+1) + '.html'), 'wb')
142   file.write(r.content)
143   file.close
144   headFile.write('Status HTTP Header:\n')
145       for item in r.headers:
146             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
147   #Request for network page
148   r = requests.get(str(url+'/network.htm'), allow_redirects=True)
149   file = open(str(folder + '/html/network_' + str(x+1) + '.html'), 'wb')
150   file.write(r.content)
151   file.close
152   headFile.write('Network HTTP Header:\n')
153       for item in r.headers:
154             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
155   #Request for settings page
156   r = requests.get(str(url+'/settings.htm'), allow_redirects=True)
157   file = open(str(folder + '/html/settings_' + str(x+1) + '.html'), 'wb'
       )
158   file.write(r.content)
159   file.close
160   headFile.write('Settings HTTP Header:\n')
161       for item in r.headers:
162             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
163   #Request for schedule page
164   r = requests.get(str(url+'/schedule.htm'), allow_redirects=True)
165   file = open(str(folder + '/html/schedule_' + str(x+1) + '.html'), 'wb'
       )
166   file.write(r.content)
167   file.close
168   headFile.write('Schedule HTTP Header:\n')
169       for item in r.headers:
170             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
171   #Request for list page
172   r = requests.get(str(url+'/list.htm'), allow_redirects=True)
173   file = open(str(folder + '/html/list_' + str(x+1) + '.html'), 'wb')
174   file.write(r.content)
175   file.close
176   headFile.write('List HTTP Header:\n')
177       for item in r.headers:
178             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
179   #Request for about page
180   r = requests.get(str(url+'/about.htm'), allow_redirects=True)
181   file = open(str(folder + '/html/about_' + str(x+1) + '.html'), 'wb')
182   file.write(r.content)
183   file.close
184   headFile.write('About HTTP Header:\n')
185       for item in r.headers:
186             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
187   #Request for page that does not exist
188   r = requests.get(str(url+'/passwd'), allow_redirects=True)
189   file = open(str(folder + '/html/dnePage_' + str(x+1) + '.html'), 'wb')
190   file.write(r.content)
```

```
191   file.close
192   headFile.write('DNE HTTP Header:\n')
193       for item in r.headers:
194             headFile.write(str(item + '\n' + r.headers[item] + '\n'))
195
196 logFile.close()
197 headFile.close()
198 #Make the files writeable by anyone, right now only root can
199 subprocess.call(['chmod', '-R', '777', folder])
```

getHeaders.py

```
1  #Source: http://www.binarytides.com/python-packet-sniffer-code-linux
2  #This code extracts incoming TCP and IP packet headers and saves the
       important ones to a file
3  import socket
4  import sys
5  import time
6  from datetime import datetime
7  import os
8  from struct import *
9  import subprocess
10
11 #Create a socket that gets the RAW TCP packets
12 try:
13     s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.
       IPPROTO_TCP)
14 except socket.error , msg:
15     print 'Socket could not be created. Error Code : ' + str(msg[0]) + '
        Message ' + msg[1]
16     sys.exit()
17
18 #current sequence number
19 curr_seq = 0
20 #File where headers will be stored
21 headfile = open(str('Headers/IP TCP Headers : ' + datetime.now().
       strftime('%Y-%m-%d---%H:%M:%S') + '.txt'), 'wb')
22 #Receive the incoming TCP packets from the request
23 try:
24     while True:
25         packet = s.recvfrom(65565)
26
27         #recvfrom makes a tuple, the first portion of the tuple is the
       packet string
28         packet = packet[0]
29         #IP Header is 20 bytes, so first 20 characters make up the
       header
30         ip_header = packet[0:20]
31
32         #Unpack the headers
33         #! - Byte Order : Big-endian for networks
34         #B - unsigned char
35         #H - unisgned short
```

```
36          #4s - String len 4
37          iph = unpack('!BBHHHBBH4s4s' , ip_header)
38
39      #Unsigned char is 8 bits, which is the Version and IHL (IP header
          Length)
40          version_ihl = iph[0]
41          #Shift 4 bits to get the version
42          version = version_ihl >> 4
43          #Get the IHL by ANDing by 00001111 which will give the second half
         of the octet
44          ihl = version_ihl & 0xF
45          #Multiply by 4 to get number of bytes of data in the header (IHL
        is number of 32-bit words)
46          iph_length = ihl * 4
47          #Time to Live
48          ttl = iph[5]
49          #Protocol
50          protocol = iph[6]
51          #Write IP header to the header file
52          headfile.write('IP\n')
53          headfile.write('Version:\n')
54         headfile.write(str(version))
55          headfile.write('\n')
56          headfile.write('IP Header Length:\n')
57          headfile.write(str(ihl))
58          headfile.write('\n')
59          headfile.write('TTL:\n')
60          headfile.write(str(ttl))
61          headfile.write('\n')
62          headfile.write('Protocol:\n')
63          headfile.write(str(protocol))
64          headfile.write('\n')
65
66          #TCP header begins right after the IP header and is 20 bytes long
67          tcp_header = packet[iph_length:iph_length+20]
68          #Unpack the tcp header
69          tcph = unpack('!HHLLBBHHH' , tcp_header)
70          #Source port
71          source_port = tcph[0]
72          #Data offset and reserved bit
73          doff_reserved = tcph[4]
74          #Data offset is the length of the tcp header, bit shift by 4 to
        get the header length
75          tcph_length = doff_reserved >> 4
76
77          headfile.write('TCP\n')
78          headfile.write('Source Port:\n')
79          headfile.write(str(source_port))
80          headfile.write('\n')
81          headfile.write('TCP header length:\n')
82          headfile.write(str(tcph_length))
83          headfile.write('\n')
```

```
84
85        #Total header size in bytes is len of IP header and (TCP header *
      4)
86        #iph_length and tcph_length detail the number of 4 byte words.
87        h_size = iph_length + tcph_length * 4
88        #Length of data is length of the packet - the length of the
      headers
89        data_size = len(packet) - h_size
90
91        #Get all of the data from the packet
92        data = packet[h_size:]
93
94        #print 'Data : ' + data
95        #print
96  except KeyboardInterrupt:
97      #Make the files writeable by anyone, right now only root can
98      subprocess.call(['chmod', '-R', '777', 'Headers/'])
99      pass
```

nmapScanner.py

```
1  import sys
2  import os
3  from datetime import datetime
4  import subprocess
5  import time
6
7  #This program runs the SYN, UDP, and FIN Nmap scans a specified number
       of times by the user
8
9  #argv[1] contains the IP address to scan
10 #argv[2] number of times to scan
11 #The program will do all 3 scans. SYN (-sS), UDP (-sU), and FIN (-sF)
12 #nmap scan command is nmap <SCAN TYPE> -T4 -A -v <IP ADDR>
13
14 #Directory where files will be saved
15 folder = datetime.now().strftime('%Y-%m-%d---%H:%M:%S')
16 os.makedirs(str('nmap/' + folder))
17
18 #Run the 3 nmap scans a specified number of times
19 for x in range(0, int(sys.argv[2])):
20   #call all nmap scans and open the files to write them to
21    print str('Beginning SYN ' + str(x))
22    with open(str('nmap/' + folder +'/SYN' + str(x) + '.txt'), "w") as f:
23      subprocess.call(['nmap', '-sS', '-T4', '-A', '-v', sys.argv[1]],
       stdout=f)
24      time.sleep(60)
25    print str('Beginning UDP ' + str(x))
26    with open(str('nmap/' + folder +'/UDP' + str(x) + '.txt'), "w") as f:
27      subprocess.call(['nmap', '-sU', '-T4', '-A', '-v', sys.argv[1]],
       stdout=f)
28      time.sleep(60)
29    print str('Beginning FIN ' + str(x))
```

```
30    with open(str('nmap/' + folder +'/FIN' + str(x) + '.txt'), "w") as f:
31      subprocess.call(['nmap', '-sF', '-T4', '-A', '-v', sys.argv[1]],
      stdout=f)
32      time.sleep(60)
33
34 subprocess.call(['chmod', '-R', '777', 'nmap/'])
```

cameraHTMLCompare.py

```
1 #!/usr/bin/python
2
3 import sys
4 import itertools
5 import os
6 import csv
7
8 #This file compares two folders of pages from the TITAThink camera and
      camera honeypot
9
10 #filename to save the results as
11 filename = sys.argv[3]
12
13 #Open a csv from the filename taken from command line argument
14 with open(str('csv/' + filename + '.csv'), 'w') as csvfile:
15   fieldnames = ['pageScanned', 'iotLen', 'honeyLen', 'iotCharLen', '
       honeyCharLen', 'diffLines', 'diffChars', 'linePercentSim', '
       charPercentSim', 'expecDiffLines', 'expecDiffChars', '
       diffLinePercentSim', 'diffcharPercentSim']
16   #Set up csv and write the headers
17    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
18    writer.writeheader()
19   #Command line parameters. The directory of the folders that contain
       the HTML for the iot and honeypot. 1st is folder of the actual
       device and 2nd is the folder of the honeypot.
20       iotFolder=sys.argv[1]
21       honeyFolder = sys.argv[2]
22       #iterate through the files of both directories.
23       for f1 in os.listdir(iotFolder):
24     #Filenames
25     iotDeviceFilename = f1
26     honeyDeviceFilename = f1
27     #File to be written to csv
28     pageScanned = f1.split("_")[0]
29     #Open the files
30     iotPage = open(str(iotFolder + '/' + iotDeviceFilename), 'r')
31     honeyPage = open(str(honeyFolder + '/' + honeyDeviceFilename), 'r')
32     #arrays containing each line from the html page. Used for contains()
       method
33     iotList = []
34     honeyList = []
35     #Total number of characters
36     iotCharLen = 0
37     honeyCharLen = 0
```

```
38      #Loop through the files and add each line to the list
39      for line in iotPage:
40          #print line
41          iotList.append(line)
42          iotCharLen += len(line)
43      for line in honeyPage:
44          #print line
45          honeyList.append(line)
46          honeyCharLen += len(line)
47      #File length in number of lines
48      iotLen = len(iotList)
49      honeyLen = len(honeyList)
50      #print iotList
51      #print honeyList
52      #Find greater number of lines, this is the max number of lines for
        calculating difference
53      if iotLen > honeyLen:
54          maxLen = iotLen
55      else:
56          maxLen = honeyLen
57      #Number of similar lines
58      numSim = 0
59      #Expected Differences
60      expecDiff = 0
61      #Pointer to current line number
62      lineNum = 0
63      #First fast pass to see if the lines are different
64      for line in iotList:
65          lineNum = lineNum + 1
66
67          if line in honeyList:
68        numSim = numSim + 1
69          else:
70        print lineNum
71        print line
72
73      #calculate number of difference lines and percentage of differences
74      numDifferent = maxLen - numSim
75      percentSimilar = (float(numSim) / float(maxLen))
76
77      expecnumDifferent = maxLen - numSim - expecDiff
78      expecpercentSimilar = (float(numSim + expecDiff) / float(maxLen))
79
80      #Character Comparison
81      #Total Number of characters in the IoT page and Honeypot page
82      iotChars = 0
83      honeyChars = 0
84      #Count number of characters in each file
85      for line in iotList:
86          for c in line:
87        iotChars = iotChars + 1
88      for line in honeyList:
```

```
89          for c in line:
90            honeyChars = honeyChars + 1
91        #Total number of characters that are the same in the IoT page and
          Honeypot page
92        sameChars = 0
93        #Expected Differences
94        expecDiff = 0
95        #Pointer to current character
96        charNum = 0
97        #Loop through each line of both HTML files. It sees which list is
          shorter and stops at the end of that list.
98        for iotLine, honeyLine in zip(iotList, honeyList):
99        #loop through each character in line
100           for c1, c2 in zip(iotLine, honeyLine):
101             charNum = charNum + 1
102
103             if c1 is c2:
104                 sameChars = sameChars + 1
105
106       if len(iotList) > len(honeyList):
107           totalChars = honeyChars
108       else:
109           totalChars = iotChars
110
111       #total number of different characters in the sequence including the
          expected differences
112       diffChars = totalChars - sameChars
113       percentChars = (float(sameChars) / float(totalChars))
114
115       expecdiffChars = totalChars - sameChars - expecDiff
116       expecpercentChars = (float(sameChars + expecDiff) / float(totalChars
          ))
117       #write each item to the csv
118       writer.writerow({'pageScanned': pageScanned, 'iotLen': iotLen, '
          honeyLen': honeyLen, 'iotCharLen': iotChars, 'honeyCharLen':
          honeyChars, 'diffLines': numDifferent, 'diffChars': diffChars, '
          linePercentSim': percentSimilar, 'charPercentSim': percentChars, '
          expecDiffLines': expecnumDifferent, 'expecDiffChars': expecdiffChars
          , 'diffLinePercentSim': expecpercentSimilar, 'diffcharPercentSim':
          expecpercentChars})
```

thermostatHTMLCompare.py

```
#!/usr/bin/python

import sys
import itertools
import os
import os.path
import csv
#This file compares two folders of pages from the ezOutlet2 and power outlet honeypot

#filename to save the results as
filename = sys.argv[3]

#Open a csv from the filename taken from command line argument
with open('csv/' + filename + '.csv', 'w') as csvfile:
  fieldnames = ['pageScanned', 'iotLen', 'honeyLen', 'iotCharLen', 'honeyCharLen', 'diffLines', 'diffChars', 'linePercentSim', 'charPercentSim', 'expecDiffLines', 'expecDiffChars', 'diffLinePercentSim', 'diffcharPercentSim']
  #Set up csv and write the headers
  writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
  writer.writeheader()
  #Command line parameters. The directory of the folders that contain the HTML for the iot and honeypot. 1st is folder of the actual device and 2nd is the folder of the honeypot.
     iotFolder=sys.argv[1]
     honeyFolder = sys.argv[2]
     #iterate through the files of both directories.
     for f1 in os.listdir(honeyFolder):
    #Filenames
    iotDeviceFilename = f1
    honeyDeviceFilename = f1
    #File to be written to csv
    pageScanned = f1.split("_")[0]
    #arrays containing each line from the html page. Used for contains() method
    iotList = []
    honeyList = []
    #Total number of characters
    iotCharLen = 0
    honeyCharLen = 0
    #The thermostat does not allow more than one connection, so some folders are going to be empty. Check if the file exists, if it does not, then the number of differences is the same as the number of lines in the file on the honeypot.
    #print str(ioTFolder + '/' + iotDeviceFilename)
    if not os.path.isfile(str(iotFolder + '/' + iotDeviceFilename)):
      #print str(iotFolder + '/' + iotDeviceFilename + ' Does not exist')
      #Open the files
      honeyPage = open(str(honeyFolder + '/' + honeyDeviceFilename), 'r'
```

```
    )
41      for line in honeyPage:
42            #print line
43            honeyList.append(line)
44            honeyCharLen += len(line)
45        #File length in number of lines
46        iotLen = 0
47        honeyLen = len(honeyList)
48        #
49        writer.writerow({'pageScanned': pageScanned, 'iotLen': 0, '
    honeyLen': honeyLen, 'iotCharLen': 0, 'honeyCharLen': honeyCharLen,
    'diffLines': honeyLen, 'diffChars': honeyCharLen, 'linePercentSim':
    0, 'charPercentSim': 0, 'expecDiffLines': 0, 'expecDiffChars': 0, '
    diffLinePercentSim': 0, 'diffcharPercentSim': 0})
50        continue
51    #Open the files
52    iotPage = open(str(iotFolder + '/' + iotDeviceFilename), 'r')
53    honeyPage = open(str(honeyFolder + '/' + honeyDeviceFilename), 'r')
54    #Loop through the files and add each line to the list
55    for line in iotPage:
56        #print line
57        iotList.append(line)
58        iotCharLen += len(line)
59    for line in honeyPage:
60        #print line
61        honeyList.append(line)
62        honeyCharLen += len(line)
63    #File length in number of lines
64    iotLen = len(iotList)
65    honeyLen = len(honeyList)
66    #print iotList
67    #print honeyList
68    #Find greater number of lines, this is the max number of lines for
    calculating difference
69    if iotLen > honeyLen:
70        maxLen = iotLen
71    else:
72        maxLen = honeyLen
73    #Number of similar lines
74    numSim = 0
75    #Expected Differences
76    expecDiff = 0
77    #Pointer to current line number
78    lineNum = 0
79    #First fast pass to see if the lines are different
80    for line in iotList:
81        lineNum = lineNum + 1
82        #If the line is a line that should be different, such as the
    date/time or temperature setting, this is an expected difference
83        if lineNum == 592 or lineNum == 625 or lineNum == 631 or lineNum
     == 632 or lineNum == 683 or lineNum == 688:
84      expecDiff = expecDiff + 1
```

```
85          elif line in honeyList:
86        numSim = numSim + 1
87          #else:
88        #print lineNum
89        #print line
90
91      #calculate number of difference lines and percentage of differences
92      numDifferent = maxLen - numSim
93      percentSimilar = (float(numSim) / float(maxLen))
94
95      expecnumDifferent = expecDiff
96      expecpercentSimilar = (float(numSim + expecDiff) / float(maxLen))
97
98      #Character Comparison
99      #Total Number of characters in the IoT page and Honeypot page
100     iotChars = 0
101     honeyChars = 0
102     #Count number of characters in each file
103     for line in iotList:
104         for c in line:
105       iotChars = iotChars + 1
106     for line in honeyList:
107         for c in line:
108       honeyChars = honeyChars + 1
109     #Total number of characters that are the same in the IoT page and
        Honeypot page
110     sameChars = 0
111     #Expected Differences
112     expecDiff = 0
113     #Pointer to current character
114     charNum = 0
115     #Loop through each line of both HTML files. It sees which list is
        shorter and stops at the end of that list.
116     for iotLine, honeyLine in zip(iotList, honeyList):
117     #loop through each character in line
118         for c1, c2 in zip(iotLine, honeyLine):
119       charNum = charNum + 1
120       #Check to see if the characters reside within areas that are
        expected to be different on a specific page
121       if charNum == 13232 or (charNum >= 13833 and charNum <= 13842) or
        (charNum >= 13996 and charNum <= 13999) or (charNum >= 14015 and
        charNum <= 14018) or (charNum >= 14955 and charNum <= 14958) or (
        charNum >= 15024 and charNum <= 15027):
122             expecDiff = expecDiff + 1
123       elif c1 is c2:
124             sameChars = sameChars + 1
125       #else:
126             #print charNum
127             #print str(c1 + ' -- ' + c2)
128
129     if len(iotList) > len(honeyList):
130         totalChars = honeyChars
```

```
131     else:
132         totalChars = iotChars
133 
134     #total number of different characters in the sequence including the
        expected differences
135     diffChars = totalChars - sameChars
136     percentChars = (float(sameChars) / float(totalChars))
137 
138     expecdiffChars = expecDiff
139     expecpercentChars = (float(sameChars + expecDiff) / float(totalChars
        ))
140 
141     #Write the data to the file
142     writer.writerow({'pageScanned': pageScanned, 'iotLen': iotLen, '
        honeyLen': honeyLen, 'iotCharLen': iotChars, 'honeyCharLen':
        honeyChars, 'diffLines': numDifferent, 'diffChars': diffChars, '
        linePercentSim': percentSimilar, 'charPercentSim': percentChars, '
        expecDiffLines': expecnumDifferent, 'expecDiffChars': expecdiffChars
        , 'diffLinePercentSim': expecpercentSimilar, 'diffcharPercentSim':
        expecpercentChars})
```

outletHTMLCompare.py

```
1 #!/usr/bin/python
2 
3 import sys
4 import itertools
5 import os
6 import csv
7 
8 #This file compares two folders of pages from the ezOutlet2 and power
      outlet honeypot
9 
10 #filename to save the results as
11 filename = sys.argv[3]
12 
13 #Open a csv from the filename taken from command line argument
14 with open('csv/' + filename + '.csv', 'w') as csvfile:
15   fieldnames = ['pageScanned', 'iotLen', 'honeyLen', 'iotCharLen', '
       honeyCharLen', 'diffLines', 'diffChars', 'linePercentSim', '
       charPercentSim', 'expecDiffLines', 'expecDiffChars', '
       diffLinePercentSim', 'diffcharPercentSim']
16   #Set up csv and write the headers
17   writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
18   writer.writeheader()
19   #Command line parameters. The directory of the folders that contain
       the HTML for the iot and honeypot. 1st is folder of the actual
       device and 2nd is the folder of the honeypot.
20      iotFolder=sys.argv[1]
21      honeyFolder = sys.argv[2]
22      #iterate through the files of both directories.
23      for f1 in os.listdir(honeyFolder):
24     #Filenames
```

```
25      iotDeviceFilename = f1
26      honeyDeviceFilename = f1
27      #File to be written to csv
28      pageScanned = f1.split("_")[0]
29      #print pageScanned
30      #arrays containing each line from the html page. Used for contains()
         method
31      iotList = []
32      honeyList = []
33      #Total number of characters
34      iotCharLen = 0
35      honeyCharLen = 0
36      #The outlet sometimes does not accept every simultaneous connection,
         so some folders are going to be empty. Check if the file exists, if
         it does not, then the number of differences is the same as the
        number of lines in the file on the honeypot.
37      #print str(ioTFolder + '/' + iotDeviceFilename)
38      if not os.path.isfile(str(iotFolder + '/' + iotDeviceFilename)):
39        #print str(iotFolder + '/' + iotDeviceFilename + ' Does not exist
        ')
40        #Open the files
41        honeyPage = open(str(honeyFolder + '/' + honeyDeviceFilename), 'r'
        )
42        for line in honeyPage:
43              #print line
44              honeyList.append(line)
45              honeyCharLen += len(line)
46        #File length in number of lines
47        iotLen = 0
48        honeyLen = len(honeyList)
49        #Write to file with 0 percent similarity, because its a comparison
         to an empty page
50        writer.writerow({'pageScanned': pageScanned, 'iotLen': 0, '
        honeyLen': honeyLen, 'iotCharLen': 0, 'honeyCharLen': honeyCharLen,
        'diffLines': honeyLen, 'diffChars': honeyCharLen, 'linePercentSim':
        0, 'charPercentSim': 0, 'expecDiffLines': 0, 'expecDiffChars': 0, '
        diffLinePercentSim': 0, 'diffcharPercentSim': 0})
51        continue
52      #Open the files
53      iotPage = open(str(iotFolder + '/' + iotDeviceFilename), 'r')
54      honeyPage = open(str(honeyFolder + '/' + honeyDeviceFilename), 'r')
55      #Loop through the files and add each line to the list
56      for line in iotPage:
57          #print line
58          iotList.append(line)
59          iotCharLen += len(line)
60      for line in honeyPage:
61          #print line
62          honeyList.append(line)
63          honeyCharLen += len(line)
64      #File length in number of lines
65      iotLen = len(iotList)
```

```
66      honeyLen = len(honeyList)
67      #print iotList
68      #print honeyList
69      #Find greater number of lines, this is the max number of lines for
        calculating difference
70      if iotLen > honeyLen:
71          maxLen = iotLen
72      else:
73          maxLen = honeyLen
74      #Number of similar lines
75      numSim = 0
76      #Expected Differences
77      expecDiff = 0
78      #Pointer to current line number
79      lineNum = 0
80      #First fast pass to see if the lines are different
81      for line in iotList:
82          lineNum = lineNum + 1
83          #If the line is a line that should be different, such as the
        date/time or temperature setting, this is an expected difference
84          if line in honeyList:
85        numSim = numSim + 1
86          else:
87        #print lineNum
88        #print line
89        #Check to see if specific lines lie on a line that is expected to
        be different on a specific page
90              if pageScanned == 'network':
91            if lineNum == 59 or lineNum == 60 or lineNum == 61 or lineNum
        == 63:
92                expecDiff = expecDiff + 1
93        elif pageScanned == 'status':
94            if lineNum == 31 or lineNum == 36 or lineNum == 37 or lineNum
        == 38 or lineNum == 42 or lineNum == 43:
95          expecDiff = expecDiff + 1
96
97      #calculate number of difference lines and percentage of differences
98      numDifferent = maxLen - numSim
99      percentSimilar = (float(numSim) / float(maxLen))
100     #print expecDiff
101     expecnumDifferent = expecDiff
102     expecpercentSimilar = (float(numSim + expecDiff) / float(maxLen))
103
104     #Character Comparison
105     #Total Number of characters in the IoT page and Honeypot page
106     iotChars = 0
107     honeyChars = 0
108     #Count number of characters in each file
109     for line in iotList:
110         for c in line:
111       iotChars = iotChars + 1
112     for line in honeyList:
```

```
113         for c in line:
114       honeyChars = honeyChars + 1
115     #Total number of characters that are the same in the IoT page and
    Honeypot page
116     sameChars = 0
117     #Expected Differences
118     expecDiff = 0
119     #Pointer to current character
120     charNum = 0
121     #Loop through each line of both HTML files. It sees which list is
    shorter and stops at the end of that list.
122     for iotLine, honeyLine in zip(iotList, honeyList):
123     #loop through each character in line
124         for c1, c2 in zip(iotLine, honeyLine):
125       charNum = charNum + 1
126       if c1 is c2:
127           sameChars = sameChars + 1
128       else:
129           #print charNum
130           #print str(c1 + ' -- ' + c2)
131           #Check to see if the characters reside within areas that are
    expected to be different on a specific page
132           if pageScanned == 'network':
133               if (charNum >= 2362 and charNum <= 2374) or (charNum >=
    2476 and charNum <= 2488) or (charNum >= 2591 and charNum <= 2601)
    or (charNum >= 2834 and charNum <= 2844):
134                   expecDiff = expecDiff + 1
135           elif pageScanned == 'status':
136               if (charNum >= 1219 and charNum <= 1231) or (charNum >=
    1417 and charNum <= 1429) or (charNum >= 1572 and charNum <= 1582)
    or (charNum >= 1718 and charNum <= 1734) or (charNum >= 1903 and
    charNum <= 1907) or (charNum >= 1914 and charNum <= 1915) or (
    charNum >= 1993 and charNum <= 2006) or (charNum >= 2038 and charNum
     <= 2051):
137                   expecDiff = expecDiff + 1
138     #Determine the min number of chars
139     if len(iotList) > len(honeyList):
140         totalChars = honeyChars
141     else:
142         totalChars = iotChars
143
144     #total number of different characters in the sequence including the
    expected differences
145     diffChars = totalChars - sameChars
146     percentChars = (float(sameChars) / float(totalChars))
147
148     expecdiffChars = expecDiff
149     expecpercentChars = (float(sameChars + expecDiff) / float(totalChars
    ))
150
151     #Write the data to the file
152     writer.writerow({'pageScanned': pageScanned, 'iotLen': iotLen, '
```

```
    honeyLen': honeyLen, 'iotCharLen': iotChars, 'honeyCharLen':
    honeyChars, 'diffLines': numDifferent, 'diffChars': diffChars, '
    linePercentSim': percentSimilar, 'charPercentSim': percentChars, '
    expecDiffLines': expecnumDifferent, 'expecDiffChars': expecdiffChars
    , 'diffLinePercentSim': expecpercentSimilar, 'diffcharPercentSim':
    expecpercentChars})
```

headerCompare.py

```
#!/usr/bin/python

import sys
import csv

#This program will take two TCP/IP header files and HTTP header files
    and compare them and calculate the differenes
#Because the TCP/IP header fields are always the same, we can get the
    exact differences for each header
#But the HTTP header is not always the same, so we just count the number
     of differences
#Comand line parameters. 1st is the TCP/IP Headers of the IoT device and
     2nd is the TCP/IP Headers of the Honeypot device. 3rd is the HTTP
    headers of the IoT device and 4th is the HTTP headres of the
    honeypot device.
iotTCPHeaderFile = sys.argv[1]
honeyTCPHeaderFile = sys.argv[2]
iotHTTPHeader = sys.argv[3]
honeyHTTPHeader = sys.argv[4]
filename = sys.argv[5]

#Create csv file
#Open csv file to store data
with open('csv/' + filename + '.csv', 'w') as csvfile:
  fieldnames = ['iotNumTCPIP', 'honeyNumTCPIP', 'maxNumTCPIP', '
    diffIPver', 'diffIPheadLen' , 'diffipttl', 'diffipprotocol', '
    difftcpsrcport', 'difftcpheadlen', 'diffhttphead', 'totalhttp']
  #set up csv and write the headers
  writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
  writer.writeheader()

  #Open the files
  #iotPage = open(iotTCPHeaderFile, 'r')
  #honeyPage = open(honeyTCPHeaderFile, 'r')
  #Number of TCP and IP packets for the iot device and honeypot
  iotNumTCPIP = 0
  honeyNumTCPIP = 0
  #Lists that hold the values of each of the fields in the TCP header
    that we are comparing for the iot device and honeypot
  iotIPver = []
  iotIPheadLen = []
  iotIPttl = []
  iotIPprotocol = []
  iotTCPsrcPort = []
```

```
iotTCPheadLen = []
honeyIPver = []
honeyIPheadLen = []
honeyIPttl = []
honeyIPprotocol = []
honeyTCPsrcPort = []
honeyTCPheadLen = []

#Loop through the files and add each line to the respective list
  according to the header field it is part of
with open(iotTCPHeaderFile) as f:
  for line in f:
    #print line
    #print nextline
    #IP designates the beginning of a new Packet.
    if line.strip() == "IP":
          iotNumTCPIP = iotNumTCPIP + 1
        elif line.strip() == 'Version:':
          nextline = next(f)
          iotIPver.append(int(nextline))
        elif line.strip() == 'IP Header Length:':
          nextline = next(f)
          iotIPheadLen.append(int(nextline))
        elif line.strip() == 'TTL:':
          nextline = next(f)
          iotIPttl.append(int(nextline))
        elif line.strip() == 'Protocol:':
          nextline = next(f)
          iotIPprotocol.append(int(nextline))
      elif line.strip() == 'Source Port:':
          nextline = next(f)
          iotTCPsrcPort.append(int(nextline))
        elif line.strip() == 'TCP header length:':
          nextline = next(f)
          iotTCPheadLen.append(int(nextline))
#Do the same for the HTTP IP/TCP header file
with open(honeyTCPHeaderFile) as f:
      for line in f:
        #IP designates the beginning of a new Packet.
        if line.strip() == "IP":
        honeyNumTCPIP = honeyNumTCPIP + 1
        elif line.strip() == 'Version:':
          nextline = next(f)
          honeyIPver.append(int(nextline))
        elif line.strip() == 'IP Header Length:':
          nextline = next(f)
          honeyIPheadLen.append(int(nextline))
        elif line.strip() == 'TTL:':
          nextline = next(f)
          honeyIPttl.append(int(nextline))
        elif line.strip() == 'Protocol:':
          nextline = next(f)
```

```
87              honeyIPprotocol.append(int(nextline))
88            elif line.strip() == 'Source Port:':
89              nextline = next(f)
90              honeyTCPsrcPort.append(int(nextline))
91          elif line.strip() == 'TCP header length:':
92              nextline = next(f)
93              honeyTCPheadLen.append(int(nextline))
94
95  #counters of the number of similarities in each field
96  diffipver = 0
97  diffipheadlen = 0
98  diffipttl = 0
99  diffipprotocol = 0
100 difftcpsrcport = 0
101 difftcpheadlen = 0
102
103 for x, y in zip(iotIPver, honeyIPver):
104       if x != y:
105     diffipver = diffipver + 1
106 for x, y in zip(iotIPheadLen, honeyIPheadLen):
107       if x != y:
108     diffipheadlen = diffipheadlen + 1
109 for x, y in zip(iotIPttl, honeyIPttl):
110       if x != y:
111     diffipttl = diffipttl + 1
112 for x, y in zip(iotIPprotocol, honeyIPprotocol):
113       if x != y:
114     diffipprotocol = diffipprotocol + 1
115 for x, y in zip(iotTCPsrcPort, honeyTCPsrcPort):
116       if x != y:
117     difftcpsrcport = difftcpsrcport + 1
118 for x, y in zip(iotTCPheadLen, honeyTCPheadLen):
119       if x != y:
120     difftcpheadlen = difftcpheadlen + 1
121
122 #Number of headers for the iot device and honeypot
123 iotNumHTTP = 0
124 honeyNumHTTP = 0
125 #Lists holding the number of times that a certain item appeared in a
      list of HTTP headers
126 iotHTTPhead = {}
127 honeyHTTPhead = {}
128
129 #Go through the IoT HTTP header
130 with open(iotHTTPHeader) as f:
131       for line in f:
132     #"HTTP Header" denotes the start of a set of headers
133     if line.strip() == 'Root HTTP Header:' or line.strip() == 'Login
      HTTP Header:' or line.strip() == 'Main HTTP Header:' or line.strip()
       == 'Settings HTTP Header:' or line.strip() == 'DNE HTTP Header:' or
       line.strip() == 'Status HTTP Header:':
134             iotNumHTTP = iotNumHTTP + 1
```

```
135             else:
136           #if it already is in the list, increment the number by 1
137           if line.strip() in iotHTTPhead:
138             iotHTTPhead[line.strip()] = iotHTTPhead[line.strip()] + 1
139           #If it has not been recorded in the list, start the item at 1
140           else:
141             x = line.strip()
142             #Ignore the date field. It will always be different
143             if x[:3] != 'Mon' and x[:3] != 'Tue' and x[:3] != 'Wed' and x
      [:3] != 'Thu' and x[:3] != 'Fri' and x[:3] != 'Sat' and x[:3] != '
      Sun':
144               iotHTTPhead[line.strip()] = 1
145   #Go through the honeypot HTTP header
146   with open(honeyHTTPHeader) as f:
147         for line in f:
148       #"HTTP Header" denotes the start of a set of headers
149       if line.strip() == 'Root HTTP Header:' or line.strip() == 'Login
       HTTP Header:' or line.strip() == 'Main HTTP Header:' or line.strip()
       == 'Settings HTTP Header:' or line.strip() == 'DNE HTTP Header:' or
       line.strip() == 'Status HTTP Header:':
150             honeyNumHTTP = honeyNumHTTP + 1
151           else:
152           #if it already is in the list, increment the number by 1
153           if line.strip() in honeyHTTPhead:
154             honeyHTTPhead[line.strip()] = honeyHTTPhead[line.strip()] + 1
155           #If it has not been recorded in the list, start the item at 1
156           else:
157             x = line.strip()
158             #Ignore the date field. It will always be different
159             if x[:3] != 'Mon' and x[:3] != 'Tue' and x[:3] != 'Wed' and x
      [:3] != 'Thu' and x[:3] != 'Fri' and x[:3] != 'Sat' and x[:3] != '
      Sun':
160               honeyHTTPhead[line.strip()] = 1
161
162   #get number of differences in HTTP header of each packet. Ignore the
       date, that will be different.
163   diffhttphead = 0
164   #Total number of HTTP items
165   totalHTTP = 0
166   for x in iotHTTPhead:
167     totalHTTP = totalHTTP + iotHTTPhead[x]
168     if not x in honeyHTTPhead:
169       diffhttphead = diffhttphead + iotHTTPhead[x]
170       #print x
171       #print iotHTTPhead[x]
172     else:
173       if iotHTTPhead[x] != honeyHTTPhead[x]:
174         diffhttphead = diffhttphead + abs(iotHTTPhead[x] - honeyHTTPhead
      [x])
175   for y in honeyHTTPhead:
176     if not y in iotHTTPhead:
177       diffhttphead = diffhttphead + honeyHTTPhead[x]
```

```
178     else:
179       if iotHTTPhead[x] != honeyHTTPhead[x]:
180         diffhttphead = diffhttphead + abs(iotHTTPhead[x] - honeyHTTPhead
        [x])
181
182   #Write data to file
183   writer.writerow({'iotNumTCPIP': iotNumTCPIP, 'honeyNumTCPIP':
       honeyNumTCPIP, 'maxNumTCPIP': max(iotNumTCPIP, honeyNumTCPIP), '
       diffIPver': diffipver, 'diffIPheadLen': diffipheadlen, 'diffipttl':
       diffipttl, 'diffipprotocol': diffipprotocol, 'difftcpsrcport':
       difftcpsrcport, 'difftcpheadlen': difftcpheadlen, 'diffhttphead':
       diffhttphead, 'totalhttp': totalHTTP})
```

Appendix F. Experimentation Data

Table 5. TITAThink camera average query response time

Trial (#Queries - #Users)	IoT Device Time (s)	Honeypot Time (s)
100-1	0.01846	0.09193
500-1	0.0178	0.09504
1000-1	0.01829	0.09224
100-10	0.36583	0.95157
500-10	0.2124	0.83672
1000-10	0.37617	0.83545
100-20	0.51574	1.71654
500-20	0.51866	1.78142
1000-20	0.52411	1.78191

Table 6. Proliphix thermostat average query response time

Trial (#Queries - #Users)	IoT Device Time (s)	Honeypot Time (s)
100-1	1.22311	1.19968
500-1	1.59755	1.19986
1000-1	1.44012	1.19866
100-10	2.47040	1.32238
500-10	1.48113	1.22359
1000-10	1.55390	1.21607
100-20	2.21184	1.34421
500-20	1.30939	1.31993
1000-20	1.66879	1.32633

Table 7. ezOutlet2 average query response time

Trial (#Queries - #Users)	IoT Device Time (s)	Honeypot Time (s)
100-1	0.00376	0.02710
500-1	0.00359	0.02784
1000-1	0.00362	0.02891
100-10	0.07662	0.24773
500-10	0.162	0.26984
1000-10	0.14008	0.31739
100-20	0.7769	0.69839
500-20	0.82573	0.66962
1000-20	0.74922	0.65482

Table 8. TITAThink camera HTML percent similarity

Trial	Line Sim	Char Sim	Line Sim w/o Expected Difference	Char Sim w/o Expected Difference
100-1	100	100	100	100
500-1	100	100	100	100
1000-1	100	100	100	100
100-10	100	100	100	100
500-10	100	100	100	100
1000-10	100	100	100	100
100-20	100	100	100	100
500-20	100	100	100	100
1000-20	100	100	100	100

Table 9. Proliphix thermostat HTML percent similarity

Trial	Line Sim	Char Sim	Line Sim w/o Expected Difference	Char Sim w/o Expected Difference
100-1	99.59	99.92	99.78	99.96
500-1	99.59	99.92	99.78	99.96
1000-1	99.72	99.94	99.91	99.98
100-10	22.10	22.14	22.14	22.15
500-10	20.09	20.16	20.13	20.17
1000-10	19.93	20.00	19.97	20.01
100-20	11.37	11.40	11.39	11.40
500-20	9.97	9.99	9.99	9.99
1000-20	10.15	10.19	10.17	10.19

Table 10. ezOutlet2 HTML percent similarity

Trial	Line Sim	Char Sim	Line Sim w/o Expected Difference	Char Sim w/o Expected Difference
100-1	99.26	99.87	100	100
500-1	99.26	99.88	100	100
1000-1	99.26	99.88	100	100
100-10	99.26	99.88	100	100
500-10	99.26	99.88	100	100
1000-10	99.26	99.88	100	100
100-20	94.25	94.83	94.95	94.95
500-20	92.30	92.88	92.99	92.99
1000-20	85.59	86.12	86.23	86.23

Table 11. TITAThink camera header information

Trial	IoT # of Packets	Honeypot # of Packets	IP/TCP # of Differences	HTTP # of Differences
100-1	8865	4635	0	0
500-1	43170	23160	0	0
1000-1	87787	46029	0	0
100-10	82195	47216	0	0
500-10	401874	236502	0	0
1000-10	805346	469528	0	0
100-20	149843	94038	0	0
500-20	747944	469171	0	0
1000-20	1488642	938891	0	0

Table 12. Proliphix thermostat header information

Trial	IoT # Packets	Honeypot # Packets	IP/TCP # Diffs	HTTP # Diffs
100-1	8300	3739	0	0
500-1	41501	18553	0	0
1000-1	83011	37168	0	0
100-10	9336	19581	0	2
500-10	42010	96118	0	0
1000-10	83126	192175	0	1
100-20	9571	38611	0	4
500-20	41502	191792	0	1
1000-20	84874	388779	0	8

Table 13. ezOutlet2 header information

Trial	IoT # Packets	Honeypot # Packets	IP/TCP # Diffs	HTTP # Diffs
100-1	7500	7274	7274	0
500-1	37500	35571	35571	0
1000-1	74997	71482	71482	0
100-10	75476	72418	72418	0
500-10	378415	362198	362198	0
1000-10	756326	729212	729212	0
100-20	143527	145042	143527	0
500-20	702103	725897	702103	0
1000-20	1302931	1460350	1302931	0

Table 14. Nmap scan times for TITAThink camera and camera honeypot

Trial	IoT SYN Time (s)	IoT UDP Time (s)	IoT FIN Time (s)
1	158.78	1187.76	156.36
2	128.94	1187.99	151.54
3	152.29	1193.92	129.05
4	152.45	1193.13	157.03
5	129.07	1192.85	156.57

Trial	Honey SYN Time (s)	Honey UDP Time (s)	Honey FIN Time (s)
1	214.01	131.10	217.90
2	211.91	306.93	214.77
3	210.94	131.09	214.36
4	211.36	292.42	226.16
5	191.78	307.36	190.80

Table 15. Nmap scan times for Proliphix thermostat and thermostat honeypot

Trial	IoT SYN Time (s)	IoT UDP Time (s)	IoT FIN Time (s)
1	50.24	323.12	68.33
2	37.15	323.11	71.29
3	44.71	319.66	68.46
4	17.85	323.43	59.27
5	89.1	323.82	69.48
Trial	Honey SYN Time (s)	Honey UDP Time (s)	Honey FIN Time (s)
1	171.72	4325.40	181.75
2	147.96	4325.74	181.03
3	169.50	4325.82	173.00
4	169.44	4325.63	150.38
5	169.31	4325.80	169.31

Table 16. Nmap scan times for ezOutlet2 and outlet honeypot

Trial	IoT SYN Time (s)	IoT UDP Time (s)	IoT FIN Time (s)
1	129.27	4326.17	2216.31
2	130.42	4325.4	2080.48
3	130.72	4325.85	2033.43
4	132.79	4325.9	2358.04
5	130.39	4325.24	2030.31
Trial	Honey SYN Time (s)	Honey UDP Time (s)	Honey FIN Time (s)
1	181.00	188.62	185.05
2	178.51	188.72	179.54
3	178.49	188.69	186.30
4	156.98	208.57	178.35
5	178.49	188.67	179.74

Appendix G. Statistical Tests

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.018456	0.091929
Variance	1.48E-05	4.55E-05
Observations	100	100
Pooled Variance	3.02E-05	
Hypothesized Mean Difference	0	
df	198	
t Stat	-94.57984	
P(T<=t) one-tail	4.8E-167	
t Critical one-tail	1.652586	
P(T<=t) two-tail	9.5E-167	
t Critical two-tail	1.972017	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 75. TITAThink camera query response time T-test 100 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.365829	0.951572
Variance	0.006547	0.018545
Observations	1000	1000
Pooled Variance	0.012546	
Hypothesized Mean Difference	0	
df	1998	
t Stat	-116.9333	
P(T<=t) one-tail	0	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0	
t Critical two-tail	1.961152	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 76. TITAThink camera query response time T-test 100 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.515743	1.716539
Variance	0.015996	0.060999
Observations	2000	2000
Pooled Variance	0.038497	
Hypothesized Mean Difference	0	
df	3998	
t Stat	-193.5323	
P(T<=t) one-tail	0	
t Critical one-tail	1.645235	
P(T<=t) two-tail	0	
t Critical two-tail	1.960558	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 77. TITAThink camera query response time T-test 100 queries - 20 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.017804	0.095045
Variance	2.13E-05	0.00197
Observations	500	500
Pooled Variance	0.000996	
Hypothesized Mean Difference	0	
df	998	
t Stat	-38.70125	
P(T<=t) one-tail	3.7E-201	
t Critical one-tail	1.646382	
P(T<=t) two-tail	7.5E-201	
t Critical two-tail	1.962344	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 78. TITAThink Camera query response time T-test 500 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.212401	0.836725
Variance	0.001874	0.003781
Observations	5000	5000
Pooled Variance	0.002828	
Hypothesized Mean Difference	0	
df	9998	
t Stat	-587.0502	
P(T<=t) one-tail	0	
t Critical one-tail	1.645006	
P(T<=t) two-tail	0	
t Critical two-tail	1.960201	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 79. TITAThink camera query response time T-test 500 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.518661	1.781423
Variance	0.008449	0.049953
Observations	10000	10000
Pooled Variance	0.029201	
Hypothesized Mean Difference	0	
df	19998	
t Stat	-522.527	
P(T<=t) one-tail	0	
t Critical one-tail	1.64493	
P(T<=t) two-tail	0	
t Critical two-tail	1.960083	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 80. TITAThink camera query response time T-test 500 queries - 20 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.018292	0.092244
Variance	2.24E-05	3.51E-05
Observations	1000	1000
Pooled Variance	2.87E-05	
Hypothesized Mean Difference	0	
df	1998	
t Stat	-308.4981	
P(T<=t) one-tail	0	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0	
t Critical two-tail	1.961152	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 81. TITAThink camera query response time T-test 1000 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.37617	0.835445
Variance	0.004719	0.00701
Observations	10000	10000
Pooled Variance	0.005865	
Hypothesized Mean Difference	0	
df	19998	
t Stat	-424.0638	
P(T<=t) one-tail	0	
t Critical one-tail	1.64493	
P(T<=t) two-tail	0	
t Critical two-tail	1.960083	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 82. TITAThink camera query response time T-test 1000 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.524106	1.78191
Variance	0.009575	0.08205
Observations	20000	20000
Pooled Variance	0.045812	
Hypothesized Mean Difference	0	
df	39998	
t Stat	-587.6551	
P(T<=t) one-tail	0	
t Critical one-tail	1.644892	
P(T<=t) two-tail	0	
t Critical two-tail	1.960023	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 83. TITAThink camera query response time T-test 1000 queries - 20 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	1.223108	1.199676
Variance	0.008654	9.19E-05
Observations	100	100
Pooled Variance	0.004373	
Hypothesized Mean Difference	0	
df	198	
t Stat	2.505633	
P(T<=t) one-tail	0.006514	
t Critical one-tail	1.652586	
P(T<=t) two-tail	0.013029	
t Critical two-tail	1.972017	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 84. Proliphix thermostat query response time T-test 100 queries - 1 user

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	2.470397	1.322377
Variance	21.74567	0.01135
Observations	112	500
df	111	499
F	1915.961	
P(F<=f) one-tail	0	
F Critical one-tail	1.263551	
F > F Critical	TRUE	
Because F > F Ctirical, we reject the null hypothesis		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	2.470397	1.322377
Variance	21.74567	0.01135
Observations	112	500
Hypothesized Mean Difference	0	
df	111	
t Stat	2.605234	
P(T<=t) one-tail	0.00522	
t Critical one-tail	1.658697	
P(T<=t) two-tail	0.010439	
t Critical two-tail	1.981567	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 85. Proliphix thermostat query response time F-test and T-test 100 queries - 5 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	2.211838	1.344205
Variance	34.26673	0.026539
Observations	115	1000
df	114	999
F	1291.193	
P(F<=f) one-tail	0	
F Critical one-tail	1.244246	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	2.211838	1.344205
Variance	34.26673	0.026539
Observations	115	1000
Hypothesized Mean Difference	0	
df	114	
t Stat	1.589385	
P(T<=t) one-tail	0.057372	
t Critical one-tail	1.65833	
P(T<=t) two-tail	0.114743	
t Critical two-tail	1.980992	
p-val < 0.05?	Fail to Reject Null	
Because we fail to reject the null, we can not prove that there is no significant difference between the means		

Figure 86. Proliphix thermostat query response time F-test and T-test 100 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	1.597555	1.19986
Variance	0.003941	0.001916
Observations	500	500
Pooled Variance	0.002929	
Hypothesized Mean Difference	0	
df	998	
t Stat	116.1947	
P(T<=t) one-tail	0	
t Critical one-tail	1.646382	
P(T<=t) two-tail	0	
t Critical two-tail	1.962344	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 87. Proliphix thermostat query response time T-test 500 queries - 1 user

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	1.48113	1.223587
Variance	0.538198	0.004859
Observations	506	2500
df	505	2499
F	110.7659	
P(F<=f) one-tail	0	
F Critical one-tail	1.117504	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis.		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	1.48113	1.223587
Variance	0.538198	0.004859
Observations	506	2500
Hypothesized Mean Difference	0	
df	507	
t Stat	7.889649	
P(T<=t) one-tail	9.4E-15	
t Critical one-tail	1.647865	
P(T<=t) two-tail	1.88E-14	
t Critical two-tail	1.964654	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 88. Proliphix thermostat query response time F-test and T-test 500 queries - 5 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	1.319928	1.309388
Variance	0.015856	0.011078
Observations	5000	500
df	4999	499
F	1.431365	
P(F<=f) one-tail	1.47E-07	
F Critical one-tail	1.118656	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis.		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	1.309388	1.319928
Variance	0.011078	0.015856
Observations	500	5000
Hypothesized Mean Difference	0	
df	651	
t Stat	-2.094299	
P(T<=t) one-tail	0.01831	
t Critical one-tail	1.647198	
P(T<=t) two-tail	0.036619	
t Critical two-tail	1.963615	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 89. Proliphix thermostat query response time F-test and T-test 500 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	1.440122	1.198659
Variance	0.003961	0.000967
Observations	1000	1000
Pooled Variance	0.002464	
Hypothesized Mean Difference	0	
df	1998	
t Stat	108.7669	
P(T<=t) one-tail	0	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0	
t Critical two-tail	1.961152	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 90. Proliphix thermostat query response time T-test 1000 queries - 1 user

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	1.553902	1.216067
Variance	8.36048	0.002808
Observations	1002	5000
df	1001	4999
F	2977.887	
P(F<=f) one-tail	0	
F Critical one-tail	1.082557	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis. The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	1.553902	1.216067
Variance	8.36048	0.002808
Observations	1002	5000
Hypothesized Mean Difference	0	
df	1001	
t Stat	3.698347	
P(T<=t) one-tail	0.000114	
t Critical one-tail	1.646377	
P(T<=t) two-tail	0.000229	
t Critical two-tail	1.962337	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 91. Proliphix thermostat query response time F-test and T-test 1000 queries - 5 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	1.66879	1.326329
Variance	10.67762	0.011678
Observations	1023	10000
df	1022	9999
F	914.3176	
P(F<=f) one-tail	0	
F Critical one-tail	1.077931	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis.		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	1.66879	1.326329
Variance	10.67762	0.011678
Observations	1023	10000
Hypothesized Mean Difference	0	
df	1022	
t Stat	3.351876	
P(T<=t) one-tail	0.000416	
t Critical one-tail	1.646346	
P(T<=t) two-tail	0.000832	
t Critical two-tail	1.962288	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 92. Proliphix thermostat query response time F-test and T-test 1000 queries - 10 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.003757	0.027105
Variance	3.96E-07	5.71E-06
Observations	100	100
Pooled Variance	3.05E-06	
Hypothesized Mean Difference	0	
df	198	
t Stat	-94.48255	
P(T<=t) one-tail	5.8E-167	
t Critical one-tail	1.652586	
P(T<=t) two-tail	1.2E-166	
t Critical two-tail	1.972017	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have		
enough certaintly to say that they are not the same		

Figure 93. ezOutlet2 query response time T-test 100 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.076623	0.247729
Variance	0.096036	0.003936
Observations	1000	1000
Pooled Variance	0.049986	
Hypothesized Mean Difference	0	
df	1998	
t Stat	-17.11309	
P(T<=t) one-tail	1.13E-61	
t Critical one-tail	1.645617	
P(T<=t) two-tail	2.26E-61	
t Critical two-tail	1.961152	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have		
enough certaintly to say that they are not the same		

Figure 94. ezOutlet2 query response time T-test 100 queries - 10 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	0.776901	0.698385
Variance	6.683718	0.006755
Observations	1899	2000
df	1898	1999
F	989.4045	
P(F<=f) one-tail	0	
F Critical one-tail	1.077375	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	0.776901	0.698385
Variance	6.683718	0.006755
Observations	1899	2000
Hypothesized Mean Difference	0	
df	1902	
t Stat	1.322823	
P(T<=t) one-tail	0.093027	
t Critical one-tail	1.645655	
P(T<=t) two-tail	0.186053	
t Critical two-tail	1.961212	
p-val < 0.05?	Fail to Reject Null	
Because we fail to reject the null, we do not have		
enough certaintly to say that they are not the same		

Figure 95. ezOutlet2 query response time F-test and T-test 100 queries - 20 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.003594	0.027841
Variance	1.74E-07	1.63E-05
Observations	500	500
Pooled Variance	8.24E-06	
Hypothesized Mean Difference	0	
df	998	
t Stat	-133.5619	
P(T<=t) one-tail	0	
t Critical one-tail	1.646382	
P(T<=t) two-tail	0	
t Critical two-tail	1.962344	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have enough certaintly to say that they are not the same		

Figure 96. ezOutlet2 query response time T-test 500 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.162003	0.26984
Variance	0.2774	0.003602
Observations	5000	5000
Pooled Variance	0.140501	
Hypothesized Mean Difference	0	
df	9998	
t Stat	-14.38457	
P(T<=t) one-tail	9.39E-47	
t Critical one-tail	1.645006	
P(T<=t) two-tail	1.88E-46	
t Critical two-tail	1.960201	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have enough certaintly to say that they are not the same		

Figure 97. ezOutlet2 query response time T-test 500 queries - 10 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	0.825734	0.669619
Variance	9.491682	0.008541
Observations	9299	10000
df	9298	9999
F	1111.361	
P(F<=f) one-tail	0	
F Critical one-tail	1.03407	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis.		
The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	0.825734	0.669619
Variance	9.491682	0.008541
Observations	9299	10000
Hypothesized Mean Difference	0	
df	9314	
t Stat	4.884383	
P(T<=t) one-tail	5.27E-07	
t Critical one-tail	1.645017	
P(T<=t) two-tail	1.05E-06	
t Critical two-tail	1.960219	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a		
significant difference between the means		

Figure 98. ezOutlet2 query response time T-test 500 queries - 20 users

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.00362	0.028914
Variance	6.87E-07	1.45E-05
Observations	1000	1000
Pooled Variance	7.58E-06	
Hypothesized Mean Difference	0	
df	1998	
t Stat	-205.4219	
P(T<=t) one-tail	0	
t Critical one-tail	1.645617	
P(T<=t) two-tail	0	
t Critical two-tail	1.961152	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have		
enough certaintly to say that they are not the same		

Figure 99. ezOutlet2 query response time T-test 1000 queries - 1 user

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.140081	0.317391
Variance	0.346186	0.00527
Observations	10000	10000
Pooled Variance	0.175728	
Hypothesized Mean Difference	0	
df	19998	
t Stat	-29.90868	
P(T<=t) one-tail	1.3E-192	
t Critical one-tail	1.64493	
P(T<=t) two-tail	2.6E-192	
t Critical two-tail	1.960083	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have enough certaintly to say that they are not the same		

Figure 100. ezOutlet2 query response time T-test 1000 queries - 10 users

F-Test Two-Sample for Variances		
	Variable 1	*Variable 2*
Mean	0.749225	0.654824
Variance	6.82387	0.006429
Observations	17245	20000
df	17244	19999
F	1061.41	
P(F<=f) one-tail	0	
F Critical one-tail	1.024456	
F > F Critical	TRUE	
Because F > F Critical, we reject the null hypothesis. The variances of the two populations are unequal		

t-Test: Two-Sample Assuming Unequal Variances		
	Variable 1	*Variable 2*
Mean	0.749225	0.654824
Variance	6.82387	0.006429
Observations	17245	20000
Hypothesized Mean Difference	0	
df	17272	
t Stat	4.74371	
P(T<=t) one-tail	1.06E-06	
t Critical one-tail	1.644942	
P(T<=t) two-tail	2.12E-06	
t Critical two-tail	1.960101	
p-val < 0.05?	Reject Null	
Because we reject the null, we know that there is a significant difference between the means		

Figure 101. ezOutlet2 query response time T-test 1000 queries - 20 users

Mann-Whitney Test for Two Independent Samples		
	Camera	Honeypot
count	5	5
median	152.29	211.36
rank sum	15	40
U	25	0
	one tail	two tail
U	0	
mean	12.5	
std dev	4.7871355	
z-score	2.5067182	
effect r	0.7926939	
p-value	0.0060929	0.012186
p-value < 0.05	Reject Null	

Figure 102. TITAThink camera Nmap SYN times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples		
	Camera	Honeypot
count	5	5
median	1192.85	292.42
rank sum	40	15
U	0	25
	one tail	two tail
U	0	
mean	12.5	
std dev	4.787136	
z-score	2.506718	
effect r	0.792694	
p-norm	0.006093	0.012186
p-value < 0.05	Reject Null	

Figure 103. TITAThink camera Nmap UDP times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples		
	Camera	Honeypot
count	5	5
median	156.36	214.77
rank sum	15	40
U	25	0
	one tail	two tail
U	0	
mean	12.5	
std dev	4.7871355	
z-score	2.5067182	
effect r	0.7926939	
p-norm	0.0060929	0.012186
p-value < 0.05	Reject Null	

Figure 104. TITAThink camera Nmap FIN times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples		
	Thermostat	Honeypot
count	5	5
median	44.71	169.44
rank sum	15	40
U	25	0
	one tail	two tail
U	0	
mean	12.5	
std dev	4.7871355	
z-score	2.5067182	
effect r	0.7926939	
p-norm	0.0060929	0.012186
p-value < 0.05	Reject Null	

Figure 105. Proliphix thermostat Nmap SYN times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples		
	Thermostat	Honeypot
count	5	5
median	323.12	4325.74
rank sum	15	40
U	25	0
	one tail	two tail
U	0	
mean	12.5	
std dev	4.7871355	
z-score	2.5067182	
effect r	0.7926939	
p-norm	0.0060929	0.012186
p-value < 0.05	Reject Null	

Figure 106. Proliphix thermostat Nmap UDP times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples		
	Thermostat	Honeypot
count	5	5
median	68.46	173
rank sum	15	40
U	25	0
	one tail	two tail
U	0	
mean	12.5	
std dev	4.7871355	
z-score	2.5067182	
effect r	0.7926939	
p-norm	0.0060929	0.012186
p-value < 0.05	Reject Null	

Figure 107. Proliphix thermostat Nmap FIN times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples			
	Outlet	Honeypot	
count	5	5	
median	130.42	178.49	
rank sum	15	40	
U	25	0	
	one tail	two tail	
U	0		
mean	12.5		
std dev	4.787136		
z-score	2.506718	yates	
effect r	0.792694		
p-norm	0.006093	0.012186	
p-value < 0.05	Reject Null		

Figure 108. ezOutlet2 Nmap SYN times Mann-Whitney U test

Mann-Whitney Test for Two Independent Samples			
	Outlet	Honeypot	
count	5	5	
median	4325.85	188.69	
rank sum	40	15	
U	0	25	
	one tail	two tail	
U	0		
mean	12.5		
std dev	4.787136		
z-score	2.506718	yates	
effect r	0.792694		
p-norm	0.006093	0.012186	
p-value < 0.05	Reject Null		

Figure 109. ezOutlet2 Nmap UDP times Mann-Whitney U test

t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	2143.714	181.796
Variance	20056.845	13.01653
Observations	5	5
Pooled Variance	10034.9308	
Hypothesized Mean Difference	0	
df	8	
t Stat	30.9666101	
P(T<=t) one-tail	6.4276E-10	
t Critical one-tail	1.85954804	
P(T<=t) two-tail	1.2855E-09	
t Critical two-tail	2.30600414	
p-val < 0.05?	Reject Null	
Because we fail to reject the null, we do not have enough certaintly to say that they are not the same		

Figure 110. ezOutlet2 Nmap FIN times T-test

Bibliography

1. H. Suo, J. Wan, C. Zou, and J. Liu, "Security in the internet of things: A review," in *International Conference on Computer Science and Electronics Engineering*, vol. 3, 2012, pp. 648–651.

2. W. Sun, M. Choi, and S. Choi, "IEEE 802.11ah: A Long Range 802.11 WLAN at Sub 1 GHz," *Journal of ICT Standardization*, vol. 2, no. 2, pp. 83–108, 2014.

3. A. Stachowicz, "ZigBee Wireless Networks," 2010 [Online]. Available: http://zigbee.pbworks.com/w/page/25465049/ZigBee [Accessed: 2019-11-01].

4. L. Spitzner, *Honeypots: Tracking Hackers*. Boston: Pearson Education, 2002.

5. N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, 1st ed. Boston: Pearson Education, 2008.

6. Insecure.COM LLC, "Nmap OS Fingerprinting 2nd Generation DB," 2017 [Online]. Available: https://svn.nmap.org/nmap/nmap-os-db [Accessed: 2019-11-01].

7. N. Provos, "test.sh," 2008 [Online]. Available: https://searchcode.com/codesearch/view/19216596/ [Accessed: 2019-11-01].

8. C. Hock-Chuan, "HTTP (HyperText Transfer Protocol)," 2009 [Online]. Available: http://www.ntu.edu.sg/home/ehchua/programming/webprogramming/http_basics.html [Accessed: 2019-11-01].

9. K. W. Ching and M. M. Singh, "Wearable Technology Devices Security and Privacy Vulnerability Analysis," *International Journal of Network Security & Its Applications*, vol. 8, no. 3, pp. 19–30, 2016.

10. T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices," *Proceedings of the 14th ACM Workshop on Hot Topics in Networks - HotNets-XIV*, pp. 1–7, 2015.

11. N. Provos, "Honeyd: A Virtual Honeypot Daemon," in *Proceedings of the 10th DFNCERT Workshop*, Hamburg, Germany, 2003, pp. 1–7.

12. Nest Support, "How to add your Nest thermostat to the Nest app," 2019 [Online]. Available: https://nest.com/support/article/How-do-I-pair-my-Nest-Learning-Thermostat-with-my-Nest-Account#section-4 [Accessed: 2019-06-01].

13. Ring, "Ring Setup Guide," 2018 [Online]. Available: https://images-na.ssl-images-amazon.com/images/I/E1I-CD2BeQS.pdf [Accessed: 2019-06-01].

14. T. Salman and R. Jain, "A Survey of Protocols and Standards for Internet of Things," *Advanced Computing and Communications*, vol. 1, no. 1, 2017.

15. P. McDermott-Wells, "What is Bluetooth?" *Potentials, IEEE*, vol. 23, no. 5, pp. 33–35, 2005.

16. Bluetooth SIG, "Specification of the Bluetooth System: Core System Package [BR/EDR Controller volume]," *Bluetooth Specification Version 4.0*, vol. 2, p. 36, 2010.

17. Bluetooth SIG, "Specification of the Bluetooth System: Core System Package [Low Energy Controller volume]," *Bluetooth Specification Version 4.0*, vol. 6, p. 17, 2010.

18. J. S. Lee, Y. W. Su, and C. C. Shen, "A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi," *IECON Proceedings (Industrial Electronics Conference)*, pp. 46–51, 2007.

19. D. E. Zheng and W. A. Carter, *Leveraging the Internet of Things for a More Efficient and Effective Military*. Washington D.C.: Center for Strategic & International Studies, 2015.

20. K. Rawlinson, "HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack," 2014 [Online]. Available: http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676 [Accessed: 2018-04-06].

21. M. Patton, E. Gross, R. Chinn, S. Forbis, L. Walker, and H. Chen, "Uninvited connections: A study of vulnerable devices on the internet of things (IoT)," *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014*, pp. 232–235, 2014.

22. M. Ryan, "Bluetooth: With Low Energy Comes Low Security," in *Proceedings of the 7th USENIX Conference on Offensive Technologies*, Washington D.C., 2013, p. 7 [Online]. Available: https://www.usenix.org/system/files/conference/woot13/woot13-ryan.pdf [Accessed: January 25, 2019].

23. Nest, "What is Bluetooth Low Energy (BLE), and do I need it to use Nest Products?" 2018 [Online]. Available: https://nest.com/support/article/What-is-Bluetooth-Low-Energy-BLE-and-do-I-need-it-to-use-Nest-Products [Accessed: 2018-07-23].

24. B. Cyr, W. Horn, D. Miao, and M. Specter, "Security Analysis of Wearable Fitness Devices (Fitbit)," pp. 1–14, 2014 [Online]. Available: https://courses.csail.mit.edu/6.857/2014/files/17-cyrbritt-webbhorn-specter-dmiao-hacking-fitbit.pdf [Accessed: 2018-07-23].

25. Tile, "What's Tile's range?" 2018 [Online]. Available: https://support.thetileapp.com/hc/en-us/articles/200991837-What-s-Tile-s-range- [Accessed: 2018-07-23].

26. T. DiCola, "Reverse Engineering a Bluetooth Low Energy Light Bulb," 2015 [Online]. Available: https://learn.adafruit.com/reverse-engineering-a-bluetooth-low-energy-light-bulb/overview [Accessed: January 25, 2019].

27. B. Scottberg, W. Yurcik, and D. Doss, "Internet honeypots: protection or entrapment?" in *IEEE 2002 International Symposium on Technology and Society (ISTAS'02). Social Implications of Information and Communication Technology. Proceedings*, no. 2, Raleigh, NC, 2002, pp. 387–391.

28. N. Provos, "A Virtual Honeypot Framework," in *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, 2004, pp. 1–14 [Online]. Available: http://static.usenix.org/event/sec04/tech/full_papers/provos/provos_html/ [Accessed: January 25, 2019].

29. M. Masters, "Understanding Intrusion Detection Systems," 2001 [Online]. Available: https://www.sans.org/reading-room/whitepapers/detection/understanding-intrusion-detection-systems-337 [Accessed: 2018-07-23].

30. A. Orebaugh and B. Pinkard, *Nmap in the Enterprise: Your Guide to Network Scanning*. Burlington: Syngress, 2011.

31. G. F. Lyon, *Nmap network scanning : official Nmap project guide to network discovery and security scanning*, 2008 [Online]. Available: https://nmap.org/book/ [Accessed: January 25, 2019].

32. Y. M. Pa Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow, "IoTPOT: Analysing the Rise of IoT Compromises," in *USENIX Workshop on Offensive Technologies*, 2015.

33. A. G. Manzanares, "HoneyIo4 The construction of a virtual , low- interaction IoT Honeypot," Ph.D. dissertation, Universitat Politècnica de Catalunya, 2017 [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/108166/Alejandro_Guerra_Manzanares.pdf [Accessed: January 25, 2019].

34. K. P, "Capturing attacks on IoT devices with a multi-purpose IoT honeypot," Ph.D. dissertation, Indian Institute of Technology Kanpur, 2017 [Online]. Available: https://security.cse.iitk.ac.in/node/155 [Accessed: February 20, 2018].

35. M. Freeman and A. Woodward, "SmartPot - Creating a 1 st Generation Smartphone Honeypot," in *Australian Digital Forensics Conference*, Perth, Western Australia, 2009, pp. 24–31.

36. C. Kreibich and J. Crowcroft, "Honeycomb Creating Intrusion Detection Signatures Using Honeypots," *ACM SIGCOMM Computer Communications Review*, vol. 34, no. 1, pp. 51–56, 2004.

37. L. Spitzner, "Honeypots: Catching the insider threat," in *Proceedings - Annual Computer Security Applications Conference, ACSAC*, Las Vegas, NV, 2003, pp. 170–179.

38. H. Zhang, S. Wei, L. Ge, D. Shen, W. Yu, E. P. Blasch, K. D. Pham, and G. Chen, "Towards An Integrated Defense System for Cyber Security Situation Awareness Experiment," *Sensors and Systems for Space Applications*, vol. 8, 2015.

39. TITAThink, "TITAThink Store," 2018 [Online]. Available: https://titathink.com/shop/ [Accessed: 2019-02-01].

40. Proliphix Inc, *Professional Series Network Thermostat Configuration Guide (NT100e/h, NT120e/h, NT130e/h, NT150e/h, and NT160)*, 2007 [Online]. Available: http://www.proliphix.com/Collateral/Documents/English-US/ProSeriesConfigurationGuide.pdf [Accessed: January 25, 2019].

41. Mega System Technologies, "Product Introductions & Information," 2017 [Online]. Available: http://www.megatec.com.tw/info.htm#ezOutlet [Accessed: January 25, 2019].

42. IANA, "Hypertext Transfer Protocol (HTTP) Status Code Registry," 2018 [Online]. Available: https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml [Accessed: 2019-11-01].

43. S. Moon, "Basic Sniffer," 2011 [Online]. Available: https://www.binarytides.com/python-packet-sniffer-code-linux [Accessed: January 25, 2019].

44. D. Siegle, "T Test" [Online]. Available: https://researchbasics.education.uconn.edu/t-test/ [Accessed: 2019-09-01].

45. T. Levine, "T-test for non normal when N>50?" 2011 [Online]. Available: https://stats.stackexchange.com/questions/9573/t-test-for-non-normal-when-n50 [Accessed: 2019-03-01].

46. B. McNeese, "Anderson Darling Test for Normality," 2011 [Online]. Available: https://www.spcforexcel.com/knowledge/basic-statistics/anderson-darling-test-for-normality [Accessed: 2019-09-01].

47. C. Zaiontz, "Wilcoson Rank Sum Test," 2014 [Online]. Available: https://www.real-statistics.com/non-parametric-tests/wilcoxon-rank-sum-test/ [Accessed: 2019-11-01].

48. C. Zaiontz, "Mann-Whitney Test for Independent Samples," 2014 [Online]. Available: https://www.real-statistics.com/non-parametric-tests/mann-whitney-test/ [Accessed: 2019-11-01].

49. J. Jones, "Stats: F-Test," 1996 [Online]. Available: https://people.richland.edu/james/lecture/m170/ch13-f.html [Accessed: 2019-09-01].

50. Nest, "Nest Home Page," 2019 [Online]. Available: https://nest.com/ [Accessed: 2019-06-01].

51. Ring, "Ring Home Page," 2019 [Online]. Available: https://ring.com/ [Accessed: 2019-06-01].

52. L. J. Flynn, "Poor Nations Are Littered With Old PC's, Report Says," 2005 [Online]. Available: https://www.nytimes.com/2005/10/24/technology/poor-nations-are-littered-with-old-pcs-report-says.html [Accessed: 2019-06-01].

53. N. Provos, "Honeyd Downloads and Releases," 2009 [Online]. Available: http://www.honeyd.org/release.php [Accessed: 2019-06-01].

54. DataSoft, "Honeyd 1.6d GitHub," 2013 [Online]. Available: https://github.com/DataSoft/Honeyd [Accessed: 2019-06-01].

55. Cymmetria, "Honeycomb GitHub," 2018 [Online]. Available: https://github.com/Cymmetria/honeycomb [Accessed: 2019-06-01].

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
21–03–2019	Master's Thesis	Sept 2017 — Mar 2019

4. TITLE AND SUBTITLE

Examining Effectiveness of Web Based Internet of Things Honeypots

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Stafira, Lukas A, 2d Lt

5d. PROJECT NUMBER

19G437

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-19-M-057

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

Joseph A. Misher
Department of Homeland Security
Advanced Technology Security Division, Federal Protective Service
800 North Capitol Street NW, Washington D.C. 20001
COMM 202-658-8806
Email: Joseph.misher@hq.dhs.gov

10. SPONSOR/MONITOR'S ACRONYM(S)

DHS

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

12. DISTRIBUTION / AVAILABILITY STATEMENT

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Internet of Things (IoT) is growing at an alarming rate. It is estimated that there will be over 25 billion IoT devices by 2020. The simplicity of their function usually means that IoT devices have low processing power, which prevent them from having intricate security features, leading to vulnerabilities for attackers. Honeyd is popular open-source software written by Niels Provos that creates low-interaction virtual honeypots. It is able to simulate everything on the network level, allow the user to create various Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) services, and allow Operating System (OS) simulation for scanning tools such as Nmap. Three IoT devices are simulated in Honeyd: a TITAThink camera, a Proliphix thermostat, and an ezOutlet2 power outlet. The common theme among all the devices is that that they utilize the Hyptertext Transfer Protocol (HTTP) to display their information to the user. This research seeks to determine if Honeyd is capable of producing convincing web based IoT honeypots.

15. SUBJECT TERMS

Internet of Things, Honeypots, Honeyd, Smart Devices, IoT Honeypots

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. B. E. Mullins, AFIT/ENG
U	U	U	UU	223	19b. TELEPHONE NUMBER *(include area code)* (937)-255-3636 x7979; Barry.Mullins@afit.edu