



**PROACTIVE HOST MUTATION IN
SOFTWARE-DEFINED NETWORKING**

THESIS

Matthew E. Aust, 2nd Lieutenant, USAF
AFIT-ENG-MS-17-M-003

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

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

AFIT-ENG-MS-17-M-003

PROACTIVE HOST MUTATION IN SOFTWARE-DEFINED NETWORKING

THESIS

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

Matthew E. Aust, B.S.C.S.
2nd Lieutenant, USAF

March 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

PROACTIVE HOST MUTATION IN SOFTWARE-DEFINED NETWORKING

THESIS

Matthew E. Aust, B.S.C.S.
2nd Lieutenant, USAF

Committee Membership:

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

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

Michael R. Grimaila, Ph.D., CISM, CISSP
Member

Abstract

Attacks against computer networks are on the rise as adversaries find new and creative ways to penetrate systems. Some leverage the use of static IP Addresses, slowly scanning the network until enough information is available for infiltration. A novel defense method utilizes Moving Target Defense (MTD) to change visible host IP Addresses fast enough that an attacker is forced to increase their scan rate to maintain a relevant map of a network, raising their visibility on the network.

Until recently, this type of MTD would not have been possible without additional hardware and countless manhours of reconfiguration. However, with the development of Software-Defined Networking (SDN), hosts can be assigned a virtual IP Address at the network level that changes at random intervals, allowing hosts to maintain their real Dynamic Host Configuration Protocol (DHCP) assigned address. The original concept for host mutation was proposed in 2012 through research conducted at the University of North Carolina and although a promising proof of concept was developed, testing was performed in an SDN emulator without any statistical analysis. This does not negate the research, but does leave room for doubt about how such a system would perform on a real network.

To further this field of research, a testbed with fifty hosts and one attack machine (virtual machines running on ESXi) connected through an SDN switch and controlled by an SDN controller is established on physical servers. Numerous experiments are conducted with different possible configurations (i.e., 5, 10, 20, 30, 40, 50 hosts; and 30 sec, 1 min, 5 min, and 15 min mutation rates) to compare how the system performs under different conditions. Based on the results from initial testing and the original proof of concept, it is hypothesized that Proactive Host Mutation (PHM) will provided a significant difference, in terms of scan time and accuracy, over the same configuration in a traditional network. Further analysis is performed to explain metrics and make recommendations on system implementation.

Results show there is a statistically measurable difference between a traditional network and a software-defined network running host mutation software, with the same configuration. Comparing the scan times and number of hosts found from an Intense scan and a Quick scan of both networks with a t-test (null hypothesis = the networks have the same effect on scan time/hosts found), results in low P-Values, approximately 0.05 or lower. In addition to the results of the t-test, as the number of hosts increases from five to fifty, the disparity between the number of hosts found in the traditional network versus the SDN increases: beginning with a difference of only 2.1 and 2.05 for the intense and quick scans respectively and increasing to a difference of 34.74 and 25.1. This secondary validation of PHM substantiates it as a valid line of defense against network scans and should be expanded upon and studied more so as to become a full fledged application implemented in an enterprise setting.

Table of Contents

	Page
Abstract	iv
List of Figures	ix
List of Tables	xi
List of Abbreviations	xiii
I. Introduction	1
1.1 Background	1
1.2 Problem Statement	2
1.3 Goals and Hypothesis	2
1.4 Approach	3
1.5 Assumptions and Limitations	3
1.6 Contributions	4
1.7 Thesis Overview	4
II. Background and Related Research	5
2.1 Software-Defined Networking	5
Evolution of Software-Defined Networking	5
The Control and Data Planes	7
OpenFlow	8
Network Programmability and SDN Controllers	11
2.2 Network Attacks	12
Attack Structure	12
Scanning in Use in Attacks	13
2.3 Moving Target Defense	13
2.4 Random Host Mutation	15
Implementations and Goals	15
Virtual IP Address and Address Translation	16
OpenFlow Random Host Mutation	17
Spatio-Temporal Mutation	18
Proactive Defense	19
Results and Evaluation	20
2.5 Chapter Summary	21
III. Framework Design	22
3.1 Overview	22
3.2 Design Goals	22
3.3 Hosts	23

	Page
3.4 Attack Machine	23
3.5 SDN Controller	24
3.6 SDN Switch	24
3.7 Network Design	24
3.8 Framework Design	25
Example	26
3.9 Design Summary	28
IV. Methodology	29
4.1 Problem Statement	29
4.2 Experimental Design	29
Metrics	29
Factors	30
Parameters	32
Experimental Process	33
Determining Sample Size	34
Testing for Accessibility	35
Data Processing	36
4.3 Methodology Summary	36
V. Results and Analysis	38
5.1 Summary of Results	38
5.2 Intense Scan	38
Scan Time	38
Hosts Found	42
Penetration Length	45
Results - 5 Hosts	48
Results - 10 Hosts	48
Results - 20 Hosts	48
Results - 30 Hosts	49
Results - 40 Hosts	50
Results - 50 Hosts	51
Intense Scan Summary	51
5.3 Quick Scan	51
Scan Time	51
Hosts Found	55
Penetration Length	58
Results - 5 Hosts	61
Results - 10 Hosts	61
Results - 20 Hosts	61
Results - 30 Hosts	62
Results - 40 Hosts	62
Results - 50 Hosts	62

	Page
Quick Scan Summary	63
VI. Conclusions and Recommendations	65
6.1 Introduction	65
6.2 Research Conclusions	65
Stability	65
Effectiveness	66
Ease of Implementation	66
6.3 Significance of Research	66
Contributions	66
Applications	67
6.4 Future Work	67
6.5 Chapter Summary	68
Appendix A. Pilot Study Results	69
Appendix B. Experiment Results	73
2.1 5 Hosts	73
2.2 10 Hosts	75
2.3 20 Hosts	77
2.4 30 Hosts	79
2.5 40 Hosts	81
2.6 50 Hosts	83
Appendix C. R Code	85
Appendix D. Scripts	106
Appendix E. Proactive Host Mutation - Controller	108
Bibliography	117

List of Figures

Figure		Page
1	Differences in Traditional and Software-Defined Networks [1]	6
2	Software-Defined Network Architecture [2]	8
3	OpenFlow Switch [3]	9
4	Components of an OpenFlow Switch [4]	10
5	Packet Flow through Processing Pipeline [4]	11
6	The Architecture of an OF-RHM Network [5]	17
7	The Architecture of Moving Target Network [6]	18
8	Algorithm Used with NOX Controller [5]	19
9	Spatio-Temporal IP Assignment [7]	20
10	Network Diagram for PHM Framework	24
11	PHM Framework Flowchart	26
12	Example of vIP Translation	27
13	PHM System Diagram	29
14	Scan Time Observed During Intense Scans	40
15	Confidence Intervals of Mean Intense Scan Time Difference between Traditional and SDN Networks	41
16	Number of Hosts Found During Intense Scans	43
17	Confidence Intervals of Mean Intense Scan Host Difference between Traditional and SDN Networks	44
18	Length of Penetration Attack during Intense Scans	46
19	Confidence Intervals of Mean Intense Scan Penetration Length Difference between Traditional and SDN Networks	47
20	Scan Time Observed During Quick Scans	53
21	Confidence Intervals of Mean Quick Scan Time Difference between Traditional and SDN Networks	54

Figure		Page
22	Number of Hosts Found During Quick Scans	56
23	Confidence Intervals of Mean Quick Scan Host Difference between Traditional and SDN Networks	57
24	Length of Penetration Attack during Quick Scans	59
25	Confidence Intervals of Mean Quick Scan Penetration Length Difference between Traditional and SDN Networks	60

List of Tables

Table	Page
1 Metrics Summary	30
2 Factors Summary	33
3 Sample Test Matrix	34
4 Sample Test Matrix	35
5 Pilot Study Results	36
6 Intense Scan - 5 Hosts	48
7 Intense Scan - 10 Hosts	49
8 Intense Scan - 20 Hosts	49
9 Intense Scan - 30 Hosts	50
10 Intense Scan - 40 Hosts	50
11 Intense Scan - 50 Hosts	51
12 Quick Scan - 5 Hosts	61
13 Quick Scan - 10 Hosts	62
14 Quick Scan - 20 Hosts	62
15 Quick Scan - 30 Hosts	63
16 Quick Scan - 40 Hosts	63
17 Quick Scan - 50 Hosts	63
18 5 Hosts - Control	73
19 5 Hosts - 30 Second	73
20 5 Hosts - 1 Minute	74
21 5 Hosts - 5 Minutes	74
22 5 Hosts - 15 Minutes	74
23 10 Hosts - Control	75

Table		Page
24	10 Hosts - 30 Second	75
25	10 Hosts - 1 Minute	75
26	10 Hosts - 5 Minutes	76
27	10 Hosts - 15 Minutes	76
28	20 Hosts - Control	77
29	20 Hosts - 30 Second	77
30	20 Hosts - 1 Minute	77
31	20 Hosts - 5 Minutes	78
32	20 Hosts - 15 Minutes	78
33	30 Hosts - Control	79
34	30 Hosts - 30 Second	79
35	30 Hosts - 1 Minute	79
36	30 Hosts - 5 Minute	80
37	30 Hosts - 15 Minute	80
38	40 Hosts - Control	81
39	40 Hosts - 30 Second	81
40	40 Hosts - 1 Minute	81
41	40 Hosts - 5 Minute	82
42	40 Hosts - 15 Minute	82
43	50 Hosts - Control	83
44	50 Hosts - 30 Second	83
45	50 Hosts - 1 Minute	83
46	50 Hosts - 5 Minute	84
47	50 Hosts - 15 Minute	84

List of Abbreviations

Abbreviation	Page
SDN	Software-Defined Networking 1
UNC	University of North Carolina 1
IDS	Intrusion Detection Systems 5
ONF	Open Networking Foundation 6
MIT	Massachusetts Institute of Technology 6
API	Application Programming Interface 7
GENI	Global Environment for Networking Innovations 8
NAT	Network Address Translation 9
QoS	Quality of Service 9
DCAN	Devolved Control of ATM Networks 11
IETF	Internet Engineering Task Force 11
OS	Operating System 13
MTD	Moving Target Defense 13
NAS	Network Address Shuffling 14
ASLR	Address Space Layout Randomization 14
DoS	Denial of Service 15
RHM	Random Host Mutation 15
OF-RHM	OpenFlow Random Host Mutation 15
vIP	virtual IP Address 16
rIP	real IP Address 16
SMT	Satisfiability Modulo Theories 17
PHM	Proactive Host Mutation 22
MOE	Margin of Error 35

I. Introduction

This research implements a network level security countermeasure against network scanning using a Software-Defined Networking (SDN) controller (RYU) and an OpenFlow enabled switch. This chapter introduces and provides context to the problem, and an approach to addressing it. The goals and expected results are then presented as the foundation of testing the developed framework, with documented assumptions and limitations.

1.1 Background

Computer systems have become the fundamental backbone of our society. Critical for our financial operations, business transactions, and defense of our nation, it is not surprising to also see a corresponding rise in network attacks from our adversaries. Successful enemies gain an intimate knowledge of the targeted network and computer systems through research of public and private information until they know enough to escalate their efforts to network scanning. To the frustration of protectors, adversaries will often slow down the rate of their network scans, sometimes taking days or weeks to complete, to avoid detection by network scan defenses.

SDN is a relatively new construct, challenging the traditional architecture of computer networks by arguing that the control and data planes of a network should be separated to allow for network intelligence and state to be logically centralized while the underlying network infrastructure is abstracted from the applications [2]. Utilizing the OpenFlow protocol [2], SDN can allow for an almost endless number of possible applications to be engaged with the network as if the network were a single object, allowing the controller to direct the flow of data across the network.

Harnessing the power of SDN, researchers from the University of North Carolina (UNC)

proposed a proof of concept in [5] that would allow for an SDN controller to create and provide a virtual IP Address to all hosts on a network so as to hide their real identity, without affecting the hosts real IP Address or compromising on network throughput. Their tests were conducted with an SDN network simulator called Mininet [8]. While their results were promising, it lacked the authenticity of a true experiment and rather relied on empirical data from a limited environment.

1.2 Problem Statement

Given the crucial nature of network scans in attacking a network, and the difficulty of identifying and preventing them, OpenFlow Random Host Mutation was proposed as an innovative method to combat network scans in [5]. Unfortunately, because of the limited adoption of SDN in the industry at the time, the follow-on research shifted back to a traditional network architectures. Today there is an increased momentum toward implementing SDN solutions to real world problems, built on several years of SDN application history. The OpenFlow Random Host Mutation proof of concept proposed by UNC researchers [5] is still relevant today, but their experimentation lacks statistical substance. By using a simulated environment, they were limited to pinging hosts to test their framework. This research aims to address this limitation by implementing a similar framework in a more real-world environment, with physical hardware and actual networks.

1.3 Goals and Hypothesis

The goal of this thesis is to create an SDN application capable of running a host mutation framework and then test that system against a traditional network to statistically show whether or not the mutation actually increases the difficulty and accuracy of scanning a network. As part of the framework's design, it should also be stable, effective, and easy to implement. It is hypothesized that a comparison between a traditional network and a software-defined network running the host mutation application will show a statistically significant difference in network scan times and the accuracy of the scan, measured by the

number of hosts the scan finds.

1.4 Approach

A network consisting of fifty hosts, one switch, one attack machine, and one SDN controller is deployed. Time for each experiment is measured by the internal clock on the attack machine, annotating when each experiment begins, when the scan completes, when the exploit is launched, and when the exploited connection between the attack machine and a victim host is severed. The number of hosts found from each scan is also recorded. The time measurements and number of hosts are then used to compare the two networks to determine if the difference in the amount of information scanning reveals is significant.

1.5 Assumptions and Limitations

To limit the scope of the experimentation and to ensure proper interpretation of the results, some assumptions must be made. The following operating rules apply:

- The adversary is scanning from inside the network
- The IP Address translation only occurs for clients, not servers
- All clients are running Windows XP Service Pack 2 to ensure they can be penetrated
- Because all hosts are identical, it is not necessary to penetrate the same box each run, but rather pick a random one
- A dedicated attacker is able to identify hosts based on their MAC address, but because they will have to rescan the network to rediscover a hosts IP Address the system remains valid

The most notably limitations of these experiments is the lack of server support and the continued ability to fingerprint a host. Despite the promise of obfuscating scans, host mutation still suffers from the same flaw as DHCP, namely that if an IP Address is changed too frequently, then lengthy connections cannot exist. While this might be manageable when

users are connecting to websites that can easily re-establish a connection through cookies, it is more difficult with an internal FTP server. The second limitation of host mutation is the continued ability to fingerprint a host based on open ports and MAC address. These limitations are presented in the future work section in Chapter 6.

1.6 Contributions

This thesis contributes to the overall body of SDN research. Specific contributions include independent verification of results from prior research conducted on OpenFlow Random Host Mutation [5], and providing statistical evidence of the validity of host mutation as being an effective defense against network scans.

1.7 Thesis Overview

This thesis is organized into six chapters. Chapter 2 defines Software-Defined Networking, network attacks, Moving Target Defense, Random Host Mutation, and relevant research needed to understand the proposed framework being developed. Chapter 3 provides the approach to designing the framework as well as a detailed description of how each component in the framework is used and how they are combined to build the framework. Chapter 4 presents the methodology used to test and evaluated the framework designed in Chapter 3. The results for each experiment and an analysis of the data collected is presented in Chapter 5. Finally, Chapter 6 summarizes the research conducted and provides an explanation of the significance of this research and future work that could build upon it.

II. Background and Related Research

This chapter explains the background needed to understand Proactive Host Mutation in Software-Defined Networking. It begins with Section 2.1, which examines Software-Defined Networking, why it exists, how it evolved into its current state, and how it works. Section 2.2 defines how network attacks are typically conducted. Section 2.3 discusses a network defense strategy called Moving Target Defense. This chapter concludes with the related research for this thesis. Random Host Mutation and the different methods in which it is implemented is the basis upon which this thesis is built and is described in Section 2.4.

2.1 Software-Defined Networking

Software-Defined Networking is defined as an architecture where “the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications” [2]. This differs from a traditional network architecture, which utilizes distributed hardware with each device making decisions about the network, i.e., how packets flow, as well as acting as the infrastructure to carry out network operations [9]. SDN limits the role of network devices to solely performing data transfer, while the logic to control the flow of data stems from a central controlling unit. This section provides a basic outline of SDN, how it evolved, and how it works.

Evolution of Software-Defined Networking

Traditional computer networks are typically built using a large number of different network devices, such as switches, routers, and middleboxes (devices that interact with traffic for reasons other than packet forwarding such as a firewalls or Intrusion Detection Systems (IDS)) [1]. Initially this design was sufficient; it allowed for relatively easy network expansion, especially over large areas. As networks grew larger, severe limitations arose in the ability of a distributed architecture to scalably cope with new requirements [10]. Network operators found it difficult to successfully configure the plethora of devices on the

network, all with different operating procedures, to be in compliance with a multitude of high-level policies while still allowing for full network interoperability [1]. These limitations led to the search for a new architecture with SDN satisfying the requirements. Figure 1 illustrates the difference between a traditional network and a software-defined network. A software-defined network ”decouples control logic from the forwarding hardware and enables the consolidation of middleboxes, simpler policy management, and new functionalities. The solid lines define the data-plane links and the dashed lines the control-plane links.” [1]

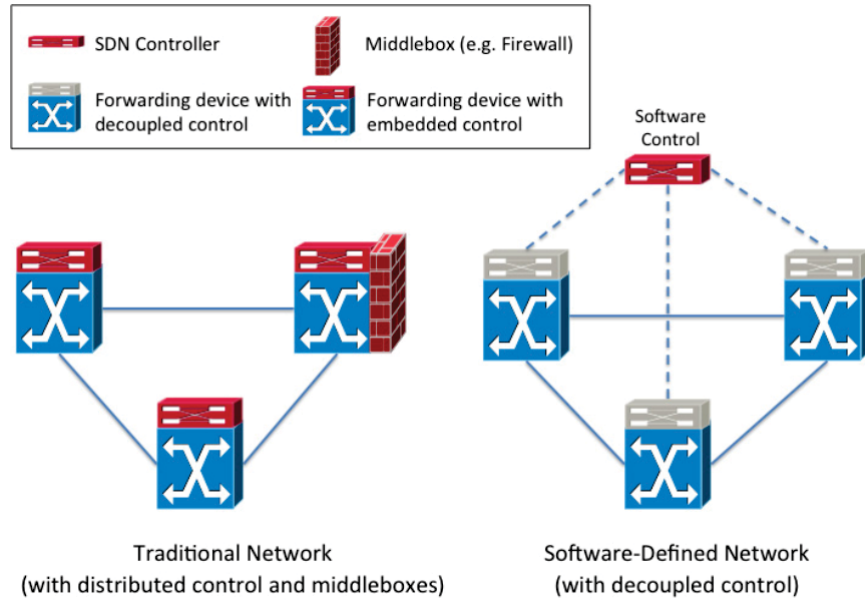


Figure 1. Differences in Traditional and Software-Defined Networks [1]

Envisioning the potential of SDN, the non-profit organization Open Networking Foundation (ONF) was formed to be “a user-driven organization dedicated to the promotion and adoption of Software-Defined Networking (SDN) through open standards development” [11]. In their defining white paper on SDN, ONF discusses four limitations of current networking technologies: complexity that leads to stasis, inconsistent policies, inability to scale, and vendor dependence [2]. These four limitations are cited by numerous researchers as the primary driving force behind adopting a new industry standard [2], [5], [9], [10], [12].

As early as 2004, researchers from Massachusetts Institute of Technology (MIT) and AT&T [10], as well as researchers from Carnegie Mellon University, Princeton University, Rutgers University, and the Naval Postgraduate School [9], were laying the groundwork for

what would become SDN. Both groups introduced new architectures in which the decision plane would be separated from the control plane alleviating the stress placed upon a vast array of distributed switches to simultaneously determine the fastest route to forward packets on, without a global view of the network, and actually sending the packets. While the proposed architectures differed from each other, the underlying idea was the same: to create a programmable network.

The Control and Data Planes.

The defining principle of SDN is the separation of the control layer from the data layer. By removing network control from individual network devices and into more readily accessible computing devices, it is possible to abstract the underlying infrastructure of a network as a single logical or virtual entity [2]. This abstraction allows applications and network services to interact with the infrastructure as if the network is one entity. Network operators can treat the network as a single, logical switch; rather than configuring hundreds or thousands of devices individually. This allows for a simplified, more reliable configuration and vendor-independent control of the entire network from a single logical point [2].

An SDN architecture is broken into three layers: application, control, and infrastructure and is shown in Figure 2. The application layer depicts how business applications view the network. By utilizing an Application Programming Interface (API), applications can interact with an entire network to implement new policies network-wide or manage multiple devices without having to wait for vendors to implement the same features in their proprietary and closed source systems. Because of this abstraction, similar to what is seen in object oriented programming, the control layer can be implemented in the easiest manner, so long as it abides by the API for the application layer. The central view provided by the control layer also opens up new opportunities for improvement in the infrastructure layer in terms of reliability and network flow. By removing the control from the data plane, the central controller(s) now have greater knowledge and can dynamically modify the flow tables used in the infrastructure layer to better route traffic, implement policies, or enforce security standards. The infrastructure layer performs the duties of the data plane, meaning

that it moves traffic where it is told to move it.

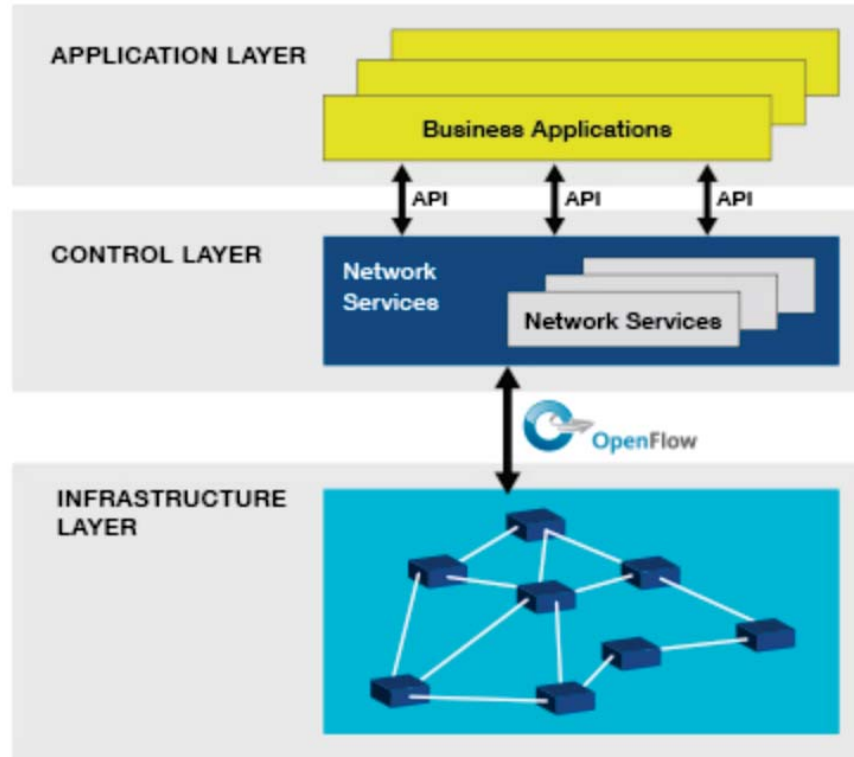


Figure 2. Software-Defined Network Architecture [2]

OpenFlow.

After several years of research and different attempts at a new design for network architecture, such as the 4D approach [9], MPLS [12], Ethane [13], and SANE [14], a protocol called OpenFlow emerged as the standard communications interface between the control and forwarding layers in an SDN architecture [2]. Initially OpenFlow was not designed with SDN in mind. Researchers at Stanford University, and across the country, wanted a practical way to experiment with new network protocols. At the time, there was no easy way to experiment with new ideas, and the Global Environment for Networking Innovations (GENI) [15], a nationwide research facility for experimenting with new networking proposals, at the time was not yet complete. This lack of a platform to experiment on, led researchers to question how to run experiments on their campus networks without risking outages [3]. Wanting a short-term solution, without having to convince commercial com-

panies to provide details on the internal workings of their switches, the researchers created OpenFlow. OpenFlow was built on the idea that modern Ethernet switches and routers contain flow tables, which are used to implement Network Address Translation (NAT), firewalls, Quality of Service (QoS), and to collect statistics. Most vendors implement their flow tables in different manners, but the team at Stanford built OpenFlow to exploit a set of common functions that all switches use. Using these functions, OpenFlow enables users to program the individual flow tables on switches and routers [3]. In contrast, to modify a flow table on a traditional switch would require an administrator to log onto to a switch, modify a table, and then save the changes, taking increasingly more time, depending on the size of the network.

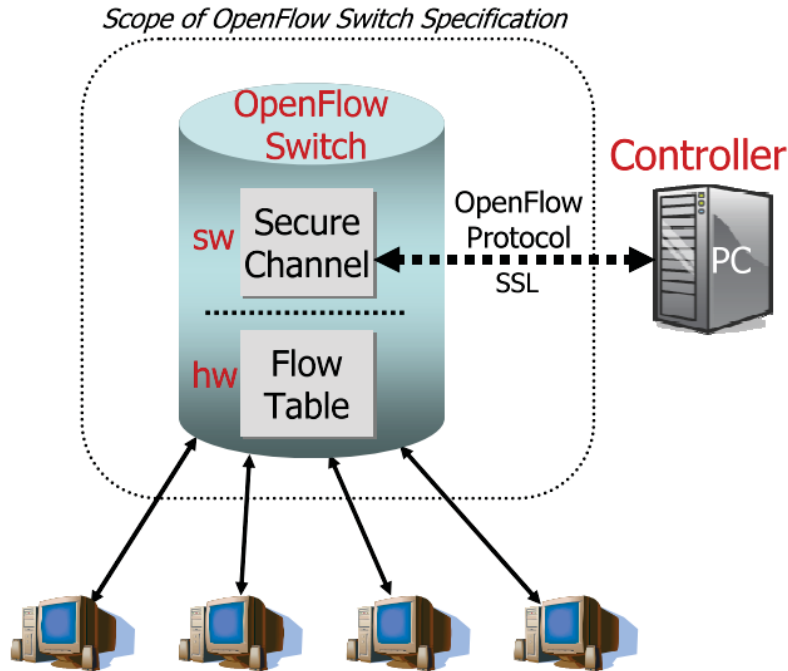


Figure 3. OpenFlow Switch [3]

An OpenFlow switch is made up of three main components, as outlined in Figure 3. The first component is a flow table, with an action associated with each flow to tell the switch how to handle the flow. The second is a secure channel that connects the switch to a controller. The third is the OpenFlow protocol, which provides the open standard on how controllers and switches communicate, allowing commands and packets to be sent between

the controller and switches [3].

The key benefit of OpenFlow is that it allows for direct access and manipulation of the forwarding plane of network devices. Using flows, OpenFlow identifies network traffic based on predefined match rules (i.e., *match*: IP Address x , *action*: forward to port y) which can be configured either statically or dynamically by the SDN controller. As seen in Figure 4, as each flow travels through the flow table pipeline it is either matched with a predefined flow, or passed on to the next table. If no match is found, the flow is processed with the default rule, which could be to forward, drop, flood, or send a request for an updated flow table from the controller. Figure 5 shows how, if the flow table matches the packet, the instructions for that flow table are performed. Depending on the rule installed by a controller, an OpenFlow switch can act like a router, switch, firewall, network address translator, or anything in-between [16]. OpenFlow also allows the network to be configured on a per-flow basis, which allows the network to react to real-time changes at higher levels [2]. Currently, Openflow is the only standardized SDN protocol that allows for this granular level of manipulation.

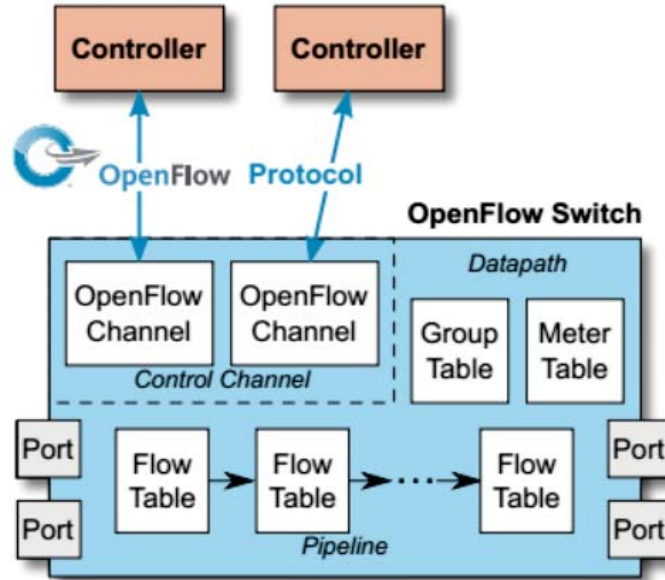


Figure 4. Components of an OpenFlow Switch [4]

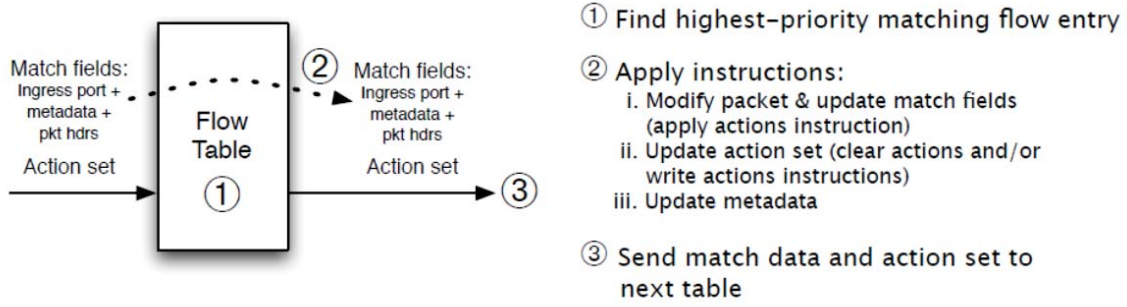


Figure 5. Packet Flow through Processing Pipeline [4]

Network Programmability and SDN Controllers.

While the idea of Software-Defined Networking might seem novel, many of the concepts evolved from a more basic notion: network programmability. The idea began in the 1990's during the initial boom of the Internet when some networking researchers, frustrated with the slow standardization process, pursued an alternative approach of opening up network control known as active networking, where a network infrastructure could be programmed for customized services [16]. The idea that a network could be programmed through an API has been built upon and modified for many years with projects like Devolved Control of ATM Networks (DCAN) [17], the Internet Engineering Task Force (IETF) Network Configuration Working Group's proposal of NETCONF [18], and Ethane [13], which, along with SANE [14], were the immediate predecessors to OpenFlow [1].

With the introduction of OpenFlow, network programmability became a much more accessible reality, and in 2007 NOX/POX, the first OpenFlow controllers, were released to the public. Since that time, many additional controllers have been created to suit different researching needs, such as Floodlight, Ryu, Beacon, Mul, and Maestro [19]. Despite differences in the controllers, such as the language it uses like Python, Java, etc., all controllers perform the same core function: manage network resources and be an interface between the network infrastructure and high level applications. By providing an API to high level applications and leveraging the granular network control provided by OpenFlow, a controller can allow users to create an endless number of software applications that can easily be implemented on any network regardless of size. Network operators can implement any number

of traditional middle box applications, such as DNS, firewalls, or IDS, as an application on top of the controller. Because the infrastructure is abstracted as a single logical unit, changes can be made easily. Even more difficult problems, such as end-to-end verification of a network or programming various network devices, can be done more quickly and easily avoiding manually intensive processes on complicated systems to verify end-to-end network connectivity [20] or creating new programming language for networks [21].

2.2 Network Attacks

As computer systems become more pervasive and vital for delivering critical information, the threat of attack or compromise also increases. Despite the wide variety of different hardware, operating systems, and services available, there is a similar methodology used in attacking systems. This section discusses this methodology, while focusing on scanning in particular.

Attack Structure.

The basic structure of an attack can be broken down into five phases — Reconnaissance, Scanning, Gaining Access, Maintaining Access, and Covering Tracks [22]. Reconnaissance focuses on gaining publicly-available knowledge about a target, like who they are, what they do, and what kind of equipment they use. Scanning focuses on assessing a network to determine the network topology and available services. Gaining Access is about getting into the system, whether it be gaining access to specific applications, operating systems, or the network. Maintaining Access deals with an attacker ensuring they do not lose access to a compromised system by installing backdoors or rootkits. Covering Tracks describes how an enemy removes traces of being on a machine or network to avoid detection. Because of the large number of different people, both good and bad, who practice the art of network penetration, there are many variations to this attack structure, but in order to be successful some form of these steps have to be followed.

Scanning in Use in Attacks.

Scanning a defender’s infrastructure can come in many forms. Some methods include War Driving techniques to find wireless access points, War Dialing to find vulnerable modems, Network Mapping to find critical hosts, routers, and firewalls, and Port Scanning to determine open ports and perform Operating System (OS) fingerprinting. Current defenses for these types of attacks are limited and rely heavily on network operators to find and remove vulnerabilities before attackers can exploit them. Implementing adequate defenses can be challenging, even for experienced network technicians, and presents greater challenges to small business owners who rely on professionals to install and/or run their networks, with no guarantee of proper installation or that a new method will not be discovered that bypasses current security.

2.3 Moving Target Defense

Because of the ever-changing landscape that of cybersecurity, network defenders have found it difficult to protect critical network infrastructure. Even with prompt patching and the latest security suites, attackers still find new and innovative ways to penetrate network defenses. Because of the key role that static configuration plays in successful attacks, a new branch of security has emerged in the last decade that, if successful, offers a new layer of defense more resilient against new attacks.

In the last decade, Moving Target Defense (MTD) has been implemented in various ways to alter the asymmetric relationship between attackers and defenders that heavily favors the attacker. There has been limited success however, because of overhead requirements and/or lack of network infrastructure to support certain systems. The goal of MTD is to “create, evaluate, and deploy mechanisms and strategies which are diverse, continually shift and change over time to increase complexity and costs for attackers, limit the exposure of vulnerabilities and opportunities for attack, and increase system resiliency” [23]. To this end, because of the important role IP Addresses and port numbers play in identifying a specific machine, most attempts to implement an MTD system for network defense have

revolved around Network Address Shuffling (NAS) [6] [7] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37].

MTD implementation can be separated into two categories, hopping and mutation. The hopping pattern is characterized by the synchronization of communication in strict time between hosts such that both sides are fully aware of the sequence used for hopping so as to maintain communication. Hopping can refer to any number of styles, but it is typically implemented with an IP Address or port hopping as they are the primary means of communication in a network. The mutation pattern does not rely on time synchronization, i.e., hosts do not necessarily have to be aware of the mutation that is occurring in the network and can update their information about addresses of other hosts with DNS or similar methods. Because of the easier and cheaper implementation and cheaper cost to implement, hopping was more heavily used in the earlier deployments of MTD, but the overhead required for each host to maintain strict synchronization with all other hosts rendered these implementations less effective, and unlikely to be utilized on a wide scale. To avoid requiring the hosts to participate in the process, the mutation pattern emerged. The individual hosts continued operating as they normally would, while specialized servers or SDN controllers mutate the IP Addresses and/or MAC addresses across the network to prevent attackers from identifying hosts.

Various researchers have evaluated MTD [38] [39] [23], and determined it as effective in a number of applications, most notably Address Space Layout Randomization (ASLR). Using MTD in the form of NAS can be effective against general enemy scanning and the subsequent attacks launched from those scans; however, using probabilistic models [39] showed that shuffling only provides limited protection from attackers who are seeking a single high-value target. This is primarily because NAS typically only modifies the IP Address of a host, allowing a dedicated attacker to identify a host based on other attributes, such as its operating system, open ports, services, and MAC address.

2.4 Random Host Mutation

Network scanning offers an attacker insight into the configuration and services of a network. If that network is configured statically, an attacker has an even greater advantage as they can then pursue directed attacks against servers or Denial of Service (DoS) attacks against the network. The basis of these attacks is the ability of the attacker to accurately identify network services. If the location of the desired target is constantly changing, executing attacks against them are more difficult. Researchers from UNC (Ehab Al-Shaer, Qi Duan, and Jarfar Haadi Jafarian) proposed several techniques to do exactly this [33] [6] [7] [34]. Their techniques, summarized by the term Random Host Mutation (RHM), utilizes either Software-Defined Networking or dedicated network devices to change the IP Address of network nodes at unpredictable intervals to unpredictable addresses. This section discusses their methodology, and specifically how they assign IP Addresses using SDN across several proof of concept proposals.

Implementations and Goals.

The researchers from UNC published four different proposals. They first published “OF-RHM: Transparent Moving Target Defense using Software-Defined Networking Networking” or OpenFlow Random Host Mutation (OF-RHM) for short [33], which was their initial proof of concept regarding RHM using Mininet [8], a SDN virtual test environment. Because of the limited scope of SDN systems, researchers then implemented RHM in a traditional network environment which they titled “Random Host Mutation for Moving Target Defense” [6]. Realizing the confines of their first two implementations, the researchers proposed a proof of concept for a new technique that would involve not only temporal mutation, but also spatial. This means the IP Address would be mutated over time, as well as mutated depending on network location, such that an attacker would see different views of the network depending on which host they were scanning. This work is called “Spatio-temporal Address Mutation for Proactive Cyber Agility against Sophisticated Attackers” [7]. Building on all of their previous work, researchers expanded RHM to not only mutate over time, but

also in response to adversary scanning. Their new system was called “Adversary-aware IP Address Randomization for Proactive Agility Against Sophisticated Attackers” [34].

Despite differences between the systems the researchers proposed, they all had very similar goals: all implementations must be highly unpredictable, fast, operationally safe, and transparent [33]. These goals will be more fully expanded in the following sections.

Virtual IP Address and Address Translation.

To meet the researchers’ goal of transparency, each implementation of RHM revolves around creating a virtual IP Address (vIP). The researchers also use the term ephemeral IP Address, but for clarity this paper will just use the term vIP. Previous network address shuffling methods ([24], [25], [27], [28], [29], and [30]) relied on involving the end hosts in the shuffling process, creating more overhead and work for network operators. RHM allows all hosts to retain their real IP Address (rIP) and relies exclusively on the network layer to perform the vIP to rIP translations. The specifics on how the vIP is generated are further discussed below, but once a vIP is generated there are two methods for providing the translation between a rIP and vIP.

The first method utilizes an SDN system. Using a controller, such as NOX [40], when a host attempts to connect to a vIP, the OpenFlow references its flow table to determine if there is a flow to that address. If so, the packet is forwarded; if not, a request is sent to the NOX Controller which responds by creating a new flow from the initiating host to the end host. Because there is a dedicated flow for every connection, if the vIP Address changes, as long as the connection is active, the host will have connectivity. Figure 6 shows a general architecture for OF-RHM, and that a single NOX Controller can be on its own VLAN and still be able to control both OpenFlow switches.

The second method, built for a traditional network, relies on dedicated equipment to perform the translations. Figure 7 shows the architecture of such a system. Rather than using a switch to perform the translation, RHM uses MTG servers positioned in each subnet to perform the translations for the subnet, similar to how a router performs network address translation. The difference between this system and NAT is that in RHM there is

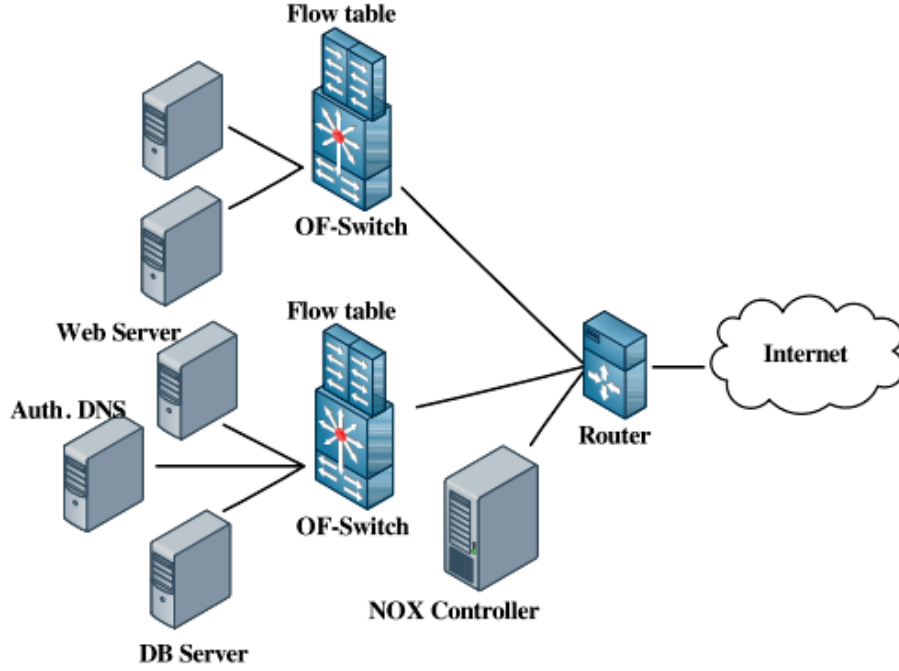


Figure 6. The Architecture of an OF-RHM Network [5]

a dedicated server, the MTC server, that performs the logic for creating and assigning new vIPs and passes down the assignments to the MTG servers. Unfortunately, the published paper did not define what MTG or MTC stands for, but their responsibilities are known.

OpenFlow Random Host Mutation.

OF-RHM has two primary objectives. First, because of the small and fragmented address space of IPv4, OF-RHM must determine an efficient way to allocate unused address ranges in such a manner that the minimum required mutation rate is met for all hosts in a particular subnet. The problem is posed as a constraint satisfiability problem and is solved with the Satisfiability Modulo Theories (SMT) [41]. Once the ranges have been allocated, the next objective is mutating the IP Address of hosts such that the minimum required mutation rate is met for all hosts and the assigned IP Addresses are unpredictable. This is accomplished in two possible ways. The first is blind mutation, where a vIP is chosen from an allocated address range with uniform probability [5]. The second is by weighted mutation, where each vIP has an associated weight based on certain criteria, such as the number

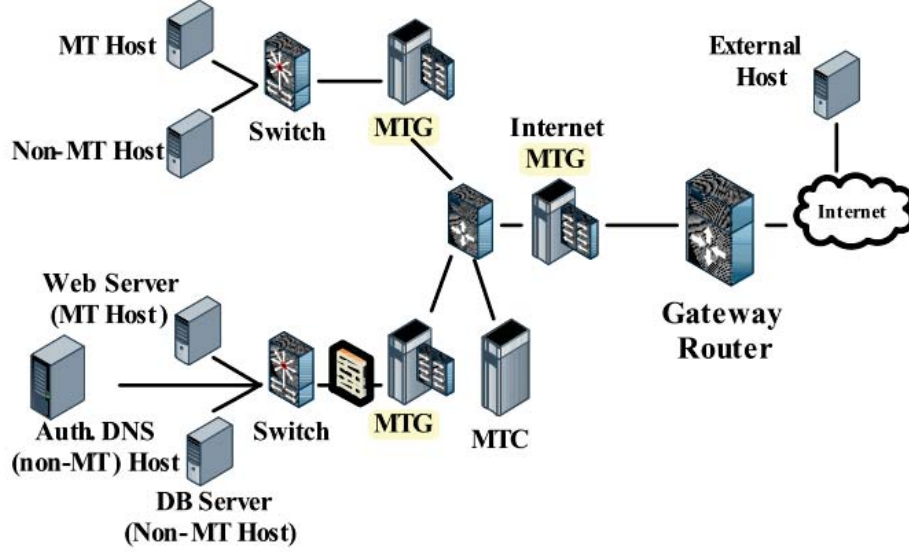


Figure 7. The Architecture of Moving Target Network [6]

of times that IP Address has been scanned. Figure 8 is the algorithm used to implement OF-RHM. This algorithm performs three basic steps: determine which address ranges have not been used in assigned rIPs; assign those ranges to each subnet so that there are enough addresses for the mutation process; and perform the translations, including creating new flows, ensuring only authorized hosts can access restricted machines, and mutating the vIPs for hosts in the network.

Spatio-Temporal Mutation.

The researchers' third proposal deviates from the previous two in how vIPs are assigned. Instead of simply mutating vIPs based on time, which is the temporal method, this method also takes into consideration where a host is in the network. By varying the vIP based on time and space, the reconnaissance information gathered at one host would not be valid at a different time or from a different host. As shown in Figure 9, there is an individual IP Address to access host B from host A; but if a user wanted to access host B from a different host, say host C (not seen in the figure), they would have to use a different IP Address. Figure 9 also shows that at different time intervals the IP Address to reach host B is also different.

```

determine unused ranges.
determine range-to-subnet assignments
for all packets  $p$  from OF-Switches do
  if  $p$  is a Type-A DNS response for host  $h_i$  then
    set DNS  $addr$  to current  $vIP(h_i)$ ,  $TTL \simeq 0$ 
  else if  $p$  is a TCP-SYN or UDP from  $h_i$  to  $h_j$  then
    if  $p.src$  is internal then
      install  $in$  flow in src OF-switch with
        action  $srcIP(p) := vIP(h_i)$ 
      install  $out$  flow in src OF-switch with
        action  $dstIP(p) := rIP(h_i)$ 
    end if
    if  $p.dst$  is rIP then
      if  $h_i$  access to  $h_j$  is authorized then
        install  $in$  and  $out$  flows in dest OF-switch
      end if
    else [ $p.dst$  is vIP]
      install  $in$  flow in dest OF-switch with
        action  $dstIP(p) := rIP(h_j)$ 
      install  $out$  flow in dest OF-switch with
        action  $srcIP(p) := vIP(h_j)$ 
    end if
  end if
for all mutation of each host  $h_i$  do
  set  $vIP(h_i)$  to a new vIP
end for
end for

```

Figure 8. Algorithm Used with NOX Controller [5]

Proactive Defense.

An additional layer of security is added to RHM with the researchers' fourth proposal, or Adversary-aware IP Address Randomization for Proactive Agility. In other words, by analyzing scans against the network, this new system will modify how the mutation of an IP Address is occurring to avoid the scanner, and is accomplished by performing two tests. The first is a non-uniformity test, which determines if the "one-tailed null hypothesis that a distribution of destination address of failed scans are non-uniform" [34] is true or not. If the test succeeds, then the IP Address mutation will move hosts to ranges with lower probabilities of being scanned. The second is a non-repetition test, which determines if an attacker is probing the address space with limited or no replacement (i.e., a specific address is only scanned once or a very limited number of times) by calculating the deviation from each scanned address to the average number of probes [34]. If the deviation is close to zero, then the attacker is most likely scanning with very limited or no replacement. If this is the case, then the mutation will focus vIPs on more recently scanned addresses because they are the least likely to be scanned again.

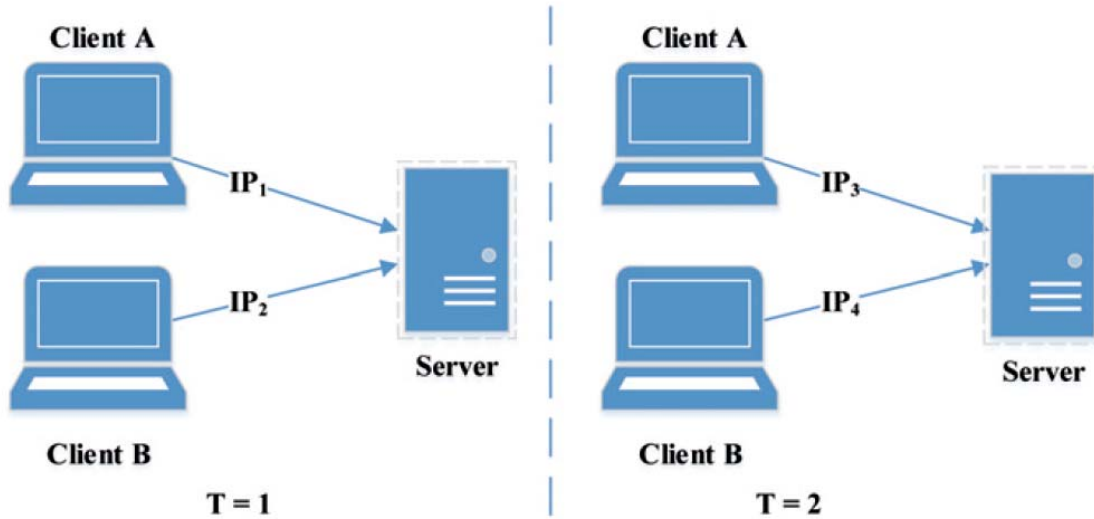


Figure 9. Spatio-Temporal IP Assignment [7]

Results and Evaluation.

While all of the implementations listed are promising and have good initial results, the evaluations for each system reflect they are simply proof of concepts and not full network-ready applications. Three of their systems, RHM for traditional networks, proactive defense, and spatio-temporal, all have explanations on the probability of an attacker's success based on a theoretical assumption of an attacker's ability, however these assumptions are only valid in certain situations, such as the Hitlist Worm, which they used frequently to show empirical results for their system, or with specific types of scans, i.e., cooperative scanners, IP sharing/mapping, local-preference scanners, etc. In an enterprise network, attackers are not going to be limited to scanning worms and predefined types of scans, but rather intelligent adversaries that are able to adapt to RHM.

The researchers' evaluated their OF-RHM system in a slightly different manner. To test the system, they ran 100 Nmap scans against their test network, finding that only 1% found real IP Addresses. They also tested the propagation of the Hitlist Worm in their system, finding that OF-RHM greatly reduced the ability of the worm to propagate; however just as with their later systems, this type of testing is limited in scope and does not fully test the

ability of their systems to slow or prevent a dedicated attacker [33]. In addition to this, while they tested their traditional network systems on real Class C networks, all work done for SDN implementations were conducted using Mininet [8], a rapid prototyping environment. While this type of testing is adequate for creating a working product and proof of concept testing, it does not satisfy the rigorous testing demanded of products before being deployed on a real network.

2.5 Chapter Summary

The idea of Software-Defined Networking is not new, and has evolved over the years from different ideas and projects. Its recent popularity stems from the pressure put on traditional network architectures from hackers and overwhelming difficulty network operators have in defending networks while simultaneously meeting user and management demands. By decoupling the control of a network from the physical data forwarding, networks can be modeled and interacted with as a single logical switch, solving many problems faced in current networks.

Both traditional networks and software-defined networks are subject to attack. The majority of attacks follow a typical structure. One of the key steps in an attack is scanning a network to gain information about the network topology and hosts on the network. To prevent scanning and other forms of network or host mapping, a new branch of defense, Moving Target Defense, has emerged. MTD can take many forms, but the majority of network defense techniques focused on Network Address Shuffling, which is usually implemented as hopping and mutation. This paper focuses on random host mutation implementation of NAS, as presented by researchers from UNC. The researchers presented several iterations of RHM including proof of concepts for both SDN and traditional networks. All their methods focus on creating a virtual IP Address to allow the hosts to remain unchanged and is the foundation for the remainder of this thesis.

III. Framework Design

3.1 Overview

This chapter provides a detailed description of how Proactive Host Mutation (PHM) is implemented in a software-defined network. The design of this framework is intended to test the core concept of Host Mutation, originally presented by researchers from UNC, showing how host mutation deflects an attacker’s attempts to scan a network by reducing the infection rate of the Hitlist Worm [5]. This framework is created to verify the empirical results by taking a measured, statistical approach to testing host mutation. To achieve this, the framework is design with the goals of stability, effectiveness, and ease of implementation. Each component of the framework is designed using open source software and tools utilized by industry and network professionals. By considering the goals of the framework, and the available tools that meet the specifications listed previously, PHM can be broken into four components: the hosts, an attack machine, an SDN controller, and an SDN switch capable of implementing IP address modification. The following sections describe the design parameters and the network design.

3.2 Design Goals

The following design goals are utilized while developing the PHM framework:

Stability.

1. All hosts are reachable from inside the network
2. All hosts maintain regular network access
3. The “attack” machine can exploit any given machine

Effectiveness.

1. The PHM controller reduces attacker’s ability to discover hosts

2. The PHM controller reduces attacker’s ability to maintain persistence on any given host
3. Hosts cannot be reached through real IP addresses, only through virtual addresses

Ease of Implementation.

1. Controller can be run on a virtual machine with minimal resources (i.e., 2GB of RAM, 25 GB hard drive, and an i5 equivalent level processor)
2. Framework is compatible with any software-defined network

3.3 Hosts

To reduce variability in the experiments, all fifty hosts used in the framework are run as virtual machines on a single server, on a single network interface, the design of which is depicted in Figure 10. Each host is identical with Windows XP Service Pack 2 as the operating system. To analyze how the number of hosts affect the experiments, some hosts are running and others not. This is easily controlled by simply suspending any host that is not being utilized in the experiment. Varying the number of hosts running not only affects the results of the experiment, but it also allows for a more controlled experiment environment to run in, as suspended machines only use a minimal amount of the server resources.

3.4 Attack Machine

The attack machine is a dedicated virtual machine, running on the same server as the normal hosts, but on a second network interface, depicted in Figure 10. It is on a different interface because the hypervisor used to host the VMs (ESXi) utilizes a virtual switch on each network interface which keeps traffic local on that switch from reaching the physical switch, bypassing the controller and therefore the PHM system. The attack machine is running Kali Linux (version 2016.2), with all of the most current network scanning and attack tools.

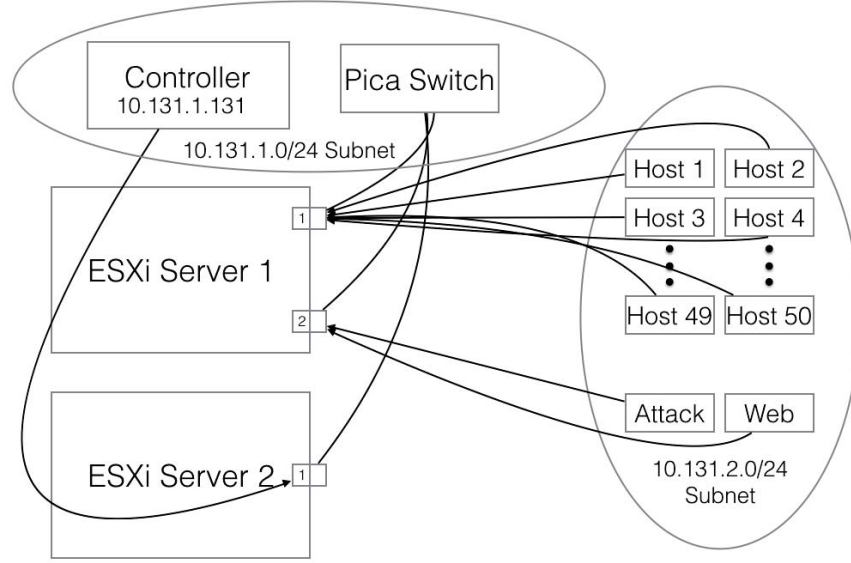


Figure 10. Network Diagram for PHM Framework

3.5 SDN Controller

The SDN controller utilized for these experiments is a Ryu controller (version 4.10) [42]. Ryu is a "component-based Software-Defined Networking framework" python-based controller, similar to POX [43] or NOX [40], but with better documentation and easier to use interface.

3.6 SDN Switch

The SDN switch used in these experiments is a Pica8 model p-3290 switch [44]. This switch is capable of running OpenFlow (version 1.3), as it is designed from the ground up to be an SDN switch, rather than OpenFlow being added as an afterthought. This allows easier setup and control throughout the experiments.

3.7 Network Design

Both traditional and software-defined networks used in this framework are designed to simulate a single network without subnets. To this end, all hosts are run as virtual machines on a single server, connected to the switch through a single network interface on the server.

The attack machine is run as a virtual machine on the same server, but it is connected through a second network interface. The controller is run as a virtual machine on a second server. It is on the same subnet as the switch, for control, and on a separate subnet from the hosts and attack machine to avoid any additional traffic on that subnet. Figure 10 shows how the PHM framework meets the design goals. It is stable by allowing all hosts to ping each other, through the vSwitch, and for the attack machine to ping the hosts through the Pica switch. It further satisfies the stability goal by allowing all hosts access to the web server through the Pica switch. The effectiveness of the PHM framework cannot be fully known from the network design, but it is tested with experimentation. The ease of implementation is fully demonstrated by building the PHM framework on actual hardware, as this will test the compatibility of it with any software-defined network as well as if it will run on any controller machine.

3.8 Framework Design

Taking the goals outline above into consideration, the following PHM algorithm was designed:

1. Assign all real addresses a vIP
2. Intercept all packets on switch without existing flows
3. Create flow between vIP Addresses
4. Mutate address mappings at given interval
5. Deleting old mappings
6. Repeat steps 2-5

This, in combination with the network design, allows for all hosts to reach each other, but only through the vIP. All hosts are still able to conduct normal network activities outside of the network as network address translation is still conducted. Because all packets not matched to an existing flow are sent to the controller, any packet attempting to reach a

rIP are ignored. Figure 11 shows this process in a graphical form. When the system first comes online, the controller assigns all rIP Addresses a vIP. Then as the switch encounters packets not matching an existing flow, it forwards the packet to the controller, which assigns a new flow between the vIP of the sender and the vIP of the receiver. This flow has an expiration of the same length as the mutation. While this allows for the flow to exist after the mutation occurs, because both hosts have a new vIP the old flow cannot be used. This process continues until the next mutation, when the controller creates new rIP to vIP mappings and removes the old mappings so that old addresses cannot be used.

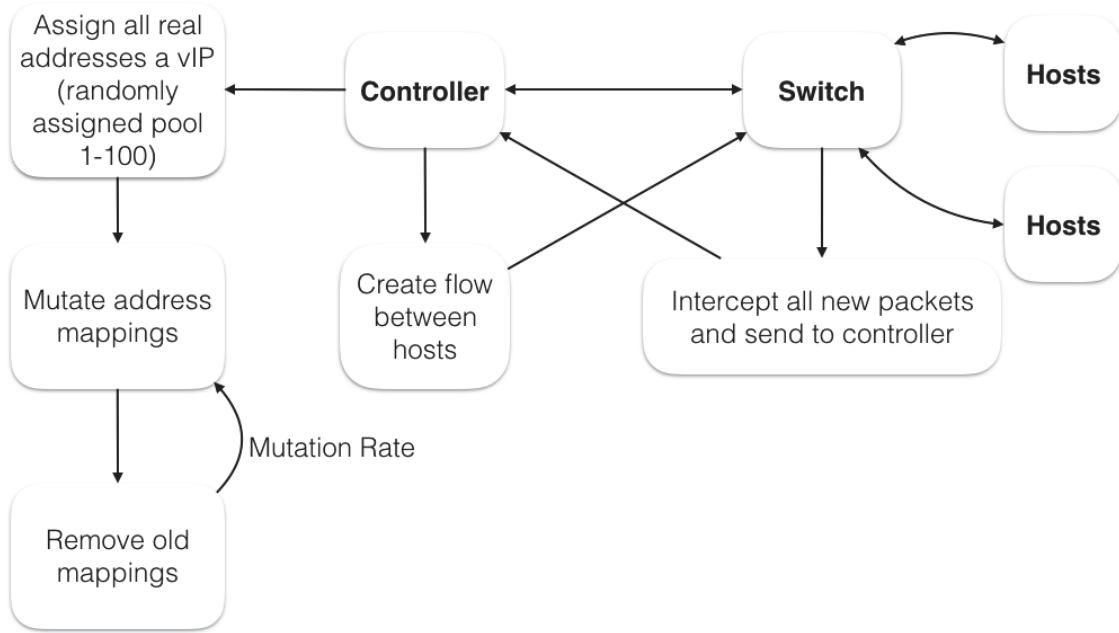


Figure 11. PHM Framework Flowchart

Example.

Figure 12 shows an example of the process used by PHM to translate virtual addresses into real addresses so that network traffic can flow. For the purposes of this example, Host A is assumed to know the vIP of Host B, but in a real network DNS would be relied upon to keep the update list of vIPs. The steps for the translation to occur are as follows:

1. Initial packet is sent from Host A to Host B

2. The switch does not have matching flow and sends the packet to the controller.
3. The controller (also acting as a learning switch) forwards the broadcast ARP if needed to find the MAC Address of target host. It stores the MAC Address of sending host. Once it has both MAC Addresses, it creates/installs a flow between Host A and B (i.e., if traffic is going from .7 to .89, translate 7 to 57 and 89 to 3).
4. This new flow allows Host B to receive the packet.
5. Host B sends a return message
6. Switch does not having matching flow and sends the packet to the controller.
7. The controller creates/installs a flow from Host B to A (i.e., if traffic is going from .3 to .57 translate 3 to 89 and 57 to 7).
8. Host A receives the packet.
9. Repeat as necessary

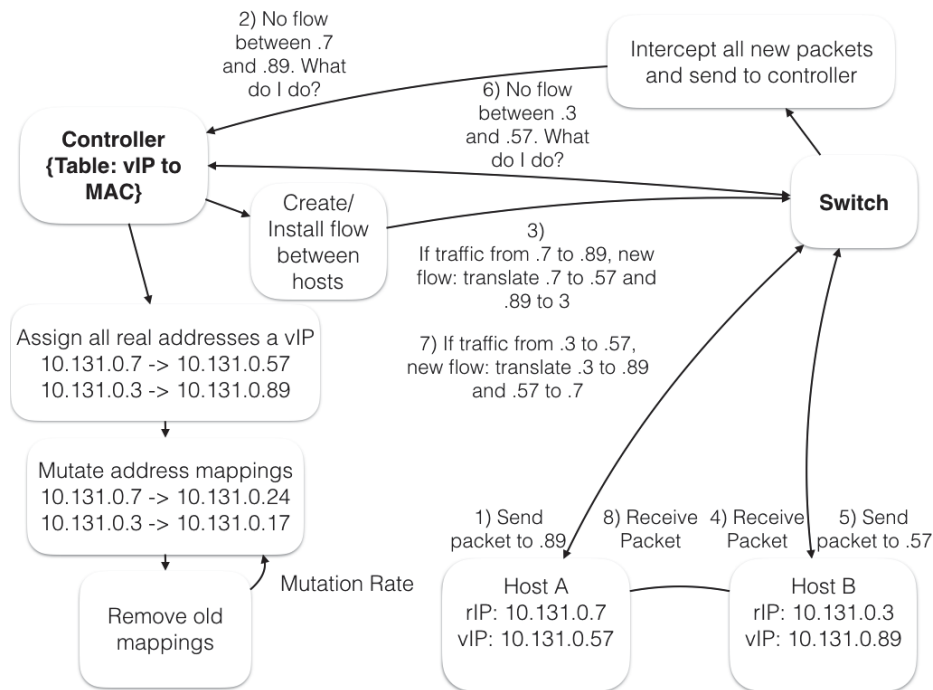


Figure 12. Example of vIP Translation

3.9 Design Summary

This chapter describes how the Proactive Host Mutation framework is designed to be stable, effective, and easy to implement. The PHM framework is designed to test the research from UNC [5], that is host mutation. The idea of host mutation is a novel approach to Moving Target Defense in that it is attempting to slow or stop attackers from being able to scan a network. The design outlined above is a simplified version of the original research [5], and is run as a single network, rather than with multiple subnets. It is a more controlled and realistic test of the host mutation concept as it is being run on physical hardware with live hosts, as opposed to the original system which was run in Mininet, an SDN emulator. Initial pilot tests (5 hosts with a 30 second mutation performed 30 times) showed that network scans were able to find all five live hosts in the traditional network, but only an average of 2.6 in the software-defined network. A t-test reveals the difference between the two networks is not due to chance with a p-value of 2.028e-12 and a 95% confidence interval of 1.977084 to 2.822916.

IV. Methodology

4.1 Problem Statement

Given the promising results, but lack of realistic infrastructure, thorough data collection, and analysis from the original research in [5], the purpose of these experiments is to measure the impact the MTD system has on the average time for an adversary to scan and penetrate a network as well as how long an attacker is able to maintain access to the exploited machine. Several different control factors are evaluated in the following sections.

4.2 Experimental Design

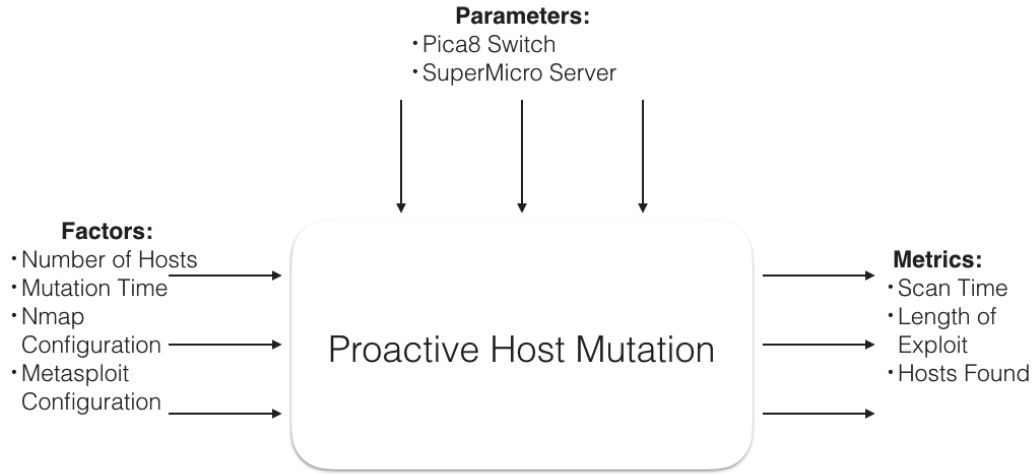


Figure 13. PHM System Diagram

Metrics.

Table 1 summarizes the metrics for each experiment below. Due to the nature of the experiments and possible discrepancies, the precision of time is limited to that of the operating system, which is deemed adequate. The list below outlines the metrics used in this experiment:

- **Scan Time** - This measures, in seconds, the amount of time it takes for an attacker to scan the network and find all hosts.
- **Length of Exploit** - This measures, in seconds, the amount of time an adversary is able to maintain access to an exploited machine
- **Number of Hosts Found** - This measures the number of hosts that Nmap is able to successfully find during a scan

Table 1. Metrics Summary

Metric	Normal operating level and range	Measurement precision, accuracy	Relationship of Metric to objective
<i>Scan Time</i>	Depends on number of live hosts	1 sec steps, measurement based on attacker's clock	Shows any increase address mutation adds to network scan time
<i>Length of Exploit</i>	Indefinite (i.e., as long as the machine is on)	1 sec steps, measurement based on attackers clock	Amount of time an attacker is able to maintain access to exploited machine
<i>Number of Hosts Found</i>	Number of Live Hosts	Counted, not measured	Shows accuracy of network scan

Factors.

Running these experiments in a non-production environment allows for many factors to be altered in order to measure the differences they cause; however this exponentially increases the number of configurations that must be tested, unless the experiments are performed as a fractional factorial experiment [45]. Therefore, to simplify the experiments,

only a limited number of variables are utilized as factors. Table 2 summarizes the following factors:

- **Hosts** - This is the number of hosts in the simulated network that an attacker scans and attempts to exploit. This value is held constant during each round of testing and is therefore considered a categorical factor with values of 5, 10, 20, 30, 40, and 50 as configuration parameters. Monitoring this value is not necessary as it will be set by only powering on that number of virtual machines.
- **Mutation Time** - This is the amount of time each host maintains their virtual IP Address. At the end of this period, all hosts receive a new virtual IP Address. This value is held constant during each round of testing and is therefore considered a categorical factor with values of 0.5, 1, 5, and 15 as configuration parameters. Monitoring this value is not necessary since it is held constant by software and is confirmed by trial runs before measurements are taken to ensure the controller is mutating at the correct intervals.
- **Nmap Configuration** - This is the configuration of Nmap used to scan the network. This value is held constant during each round of testing and is therefore considered a categorical factor with following command used for the Intense scan:

– `db_nmap -min-hostgroup 96 -T4 -A -v -n 10.131.2.0/24`

- * `db_nmap` = The command to run nmap
- * `-min-hostgroup 96` = Sets the parallel host scan group size to 96
- * `-T4` = Sets timing to 4
- * `-A` = Enable OS detection, version detection, script scanning, and traceroute
- * `-v` = Tell nmap to be verbose
- * `-n` = Never do DNS resolution/Always resolve [default: sometimes]
- * `10.131.2.0/24` = The IP Address range to conduct the scan in

and the following command used for the Quick scan:

- `db_nmap -min-hostgroup 96 -T4 -n -F 10.131.2.0/24`
 - * `db_nmap` = The command to run nmap
 - * `-min-hostgroup 96` = Sets the parallel host scan group size to 96
 - * `-T4` = Sets timing to 4
 - * `-n` = Never do DNS resolution/Always resolve [default: sometimes]
 - * `-F` = Fast mode (Scans fewer ports than the default scan)
 - * `10.131.2.0/24` = The IP Address range to conduct the scan in

Monitoring this value is not necessary as it is held constant by software, which is confirmed by trial runs before measurements are taken to ensure Nmap is working correctly.

- **Metasploit Configuration** - This is the configuration of Metasploit used to exploit and maintain access to a chosen host. This value is held constant during each round of testing and is therefore considered a categorical factor with a single factor of SMB exploit as the configuration parameter. Monitoring this value is not necessary as it is held constant by software, which is confirmed by trial runs before measurements are taken to ensure Metasploit is working correctly.

Parameters.

Throughout the course of these experiments several variables remain constant. This is not to say that these variables do not have an effect on the metrics, they most likely do.

- **Switch** - One Pica8 (model p-3290) switch, running OpenFlow (versions 1.3), and being controlled by the controller (Ryu version 4.10) remains constant throughout the experiment.
- **Servers** - Two servers, both SuperMicro SuperServers 8027R-TRF+ with a Xeon E5-4600 v2, eight 1000BASE-T network interface cards, 12 cores, and 1000GB of RAM, running ESXi, hosting the controller and all hosts. The servers do not change throughout the experiment.

Table 2. Factors Summary

Factor	Normal operating level and range	Proposed Settings	Predicted Effects
<i>Hosts</i>	0-50	5, 10, 20, 30, 40, 50	Increasing the number of hosts will increase the scan time, but also increases the likelihood an attacker exploits a box. Will not affect time to exploit box.
<i>Mutation Time</i>	0-15 Min	0.5, 1, 5, and 15	Will decrease the amount of time an attacker is able to access exploited machine.
<i>Nmap Configuration</i>	NA	Intense and Quick.	Difference in time for scan to be completed. Possible decrease host discovery.
<i>Metasploit Configuration</i>	NA	SMB	Difference in time to exploit machine and to maintain access.

Experimental Process.

The following list enumerates the steps to complete the experiment. All of the scripts listed in the list can be found in Appendix D and the code for controller can be found in Appendix E.

1. Turn on needed number of hosts
2. Turn on VM with Ryu controller

3. Access Pica8 interface (either through SSH or via putty) and delete all old flows from bridge
4. Turn on Kali VM
5. Open terminal on Kali VM and begin msfconsole (msfconsole)
6. Modify controller script with specified seconds for mutation (i.e., 30 or 300)
7. Begin controller (ryu-manager Mutator.py -verbose for experiment or ryu-manager ryu/ryu/app/simple.switch.py for control)
8. Execute quick.rc or intense.rc (i.e., resource quick.rc or resource intense.rc)
9. If an experimental run, set RHOST (i.e., set RHOST 10.131.2.242)
10. If an experimental run, execute the exploit script (i.e., resource exploit.rc)
11. If an experimental run, end the script when msfconsole displays the connection has been lost (i.e., exit)
12. Collect the length of the scan, the hosts found, and the length of the penetration, if applicable, using the test matrix in Tables 3 and the control matrix in 4.

Table 3. Sample Test Matrix

	1	2	3	4	5
Intense Scan Time					
Quick Scan Time					
Intense Pen Time					
Quick Pen Time					
Intense Hosts Found					
Quick Hosts Found					

Determining Sample Size.

Using the data collected from the pilot study, discussed in Chapter 3, and the following equation $n \geq (\frac{z * \sigma}{MOE})^2$, the required sample size is calculated. To garner a 95% confidence

Table 4. Sample Test Matrix

	1	2	3	4	5
Intense Scan Time					
Quick Scan Time					
Intense Hosts Found					
Quick Hosts Found					

level a z of 1.96 is used. A Margin of Error (MOE) of five seconds is used. The needed sample size is calculated from several different measured metrics collected during the pilot study, including the time it takes to scan the traditional network with an intense and quick scan, the time it takes to scan the software-defined network with an intense and quick scan, the number of hosts found with all four scans performed, and the length of the exploit during the quick scans (no exploits were able to complete during the intense scan). The sample standard deviation, σ , is calculated from each of the eight metrics collected and used to find the required sample size. The resulting eight different sample sizes ranged from 0.035 to 576.389. After reviewing the data from the metric that produced 576 required samples, it was determined that this data was not a representation of the population since the scan was not able to complete before the mutation causing a significantly larger standard deviation, skewing the results. By omitting this metric, the range of needed samples ranges from 0.035 to 4.432. Rounding to be more accurate, five samples are needed for each experiment to maintain a confidence level of 95% and a MOE of five seconds. Table 5 displays the results garnered from the pilot study. The results of the pilot study are contained in Appendix A

Testing for Accessibility.

As part of the pilot study, all fifty hosts are pinged one at a time from the attacking machine. This test is conducted while running the PHM framework to ensure the framework itself is not blocking access to any of the hosts. Both the vIP's and the rIP's are pinged. The test is successful and all fifty hosts are reachable from the attacking machine through their vIP, but not through their rIP. As a secondary measure of accessibility, the simple web server, shown in Figure 13, is accessed once by all of the hosts during the pilot study to ensure the PHM framework is allowing web access. All fifty hosts were able to access the

Table 5. Pilot Study Results

Metric	Mean	Standard Deviation	Calculated Sample Size
Intense Scan Time (Control)	17.908	3.362	4.432
Intense Hosts Found (Control)	4.867	0.730	0.209
Quick Scan Time (Control)	2.571	1.594	0.996
Quick Hosts Found (Control)	5	0	0
Intense PHM Scan Time	109.19	38.346	576.389
Intense PHM Hosts Found	3.045	0.302	0.036
Intense PHM Exploit Length	NA	NA	NA
Quick PHM Scan Time	9.990	2.781	2.781
Quick PHM Hosts Found	3.045	0.302	0.503
Quick PHM Exploit Length	2.6	1.133	0.0358

web server.

Data Processing.

The time to complete a scan and the number of hosts found is collected from the output of the Nmap scan. The duration the attack machine is able to maintain access to an infected host is measured by means of an automated attack script outputting the current time, measured from the internal system clock, when it begins running as the start time, and the current time again when the script ends as the end time. The difference is the length of the exploit. These results are recorded in a spreadsheet and later copied into RStudio [46] (an Integrated Developing Environment for R) for statistical processing.

4.3 Methodology Summary

This chapter describes the methodology used to measure the stability and effectiveness of the PHM framework. The experiments use varying levels of hosts and mutation times to determine how effective PHM is in preventing an adversary from gaining full knowledge of a given network. This is accomplished by contrasting the results of network scans performed on a traditional network versus the same network running the PHM framework. All other controllable factors, such as the switch and servers being utilized, are kept constant to ensure

accuracy of the resulting data. The stability of the network is determined by whether or not the attacking machine can reach all the vulnerable hosts on the same network.

V. Results and Analysis

5.1 Summary of Results

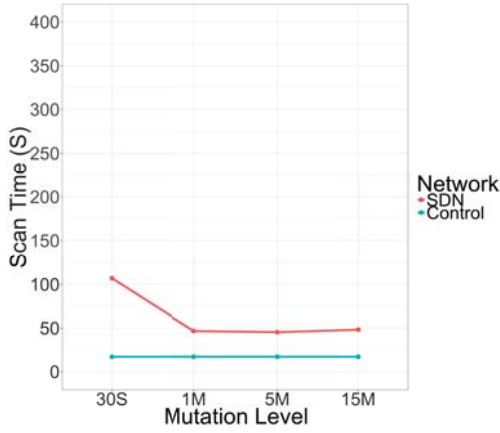
The goal of this experiment is to show a statistically significant difference between a traditional network with no additional security and a software-defined network utilizing PHM. A standard t-test with a 90% confidence level is utilized to show the differences between the two networks at each level. While the pilot study was performed with a 95% confidence level, the data collected resulted in some confidence intervals with a negative lower bound. To shift those into the positive range, the confidence level was lowered. This decreases the number of samples needed, so five samples was still acceptable. For the intense scan there are several interesting attributes throughout the experiments, most notably the thirty second mutation in multiple runs is drastically different than the rest of the mutation rates. This is due to Nmap failing to complete an entire intense scan before the PHM changed the addresses of the hosts on the network. Not only does this produce a negative t-value, but confidence intervals with a negative values. Technically a confidence interval that includes zero is invalid, but in this case indicates the effectiveness of PHM. If the results of the thirty second mutations are omitted, then the results of both scans are much more in-line with the original hypothesis: that the PHM network is significantly worse at finding and maintaining connections to hosts. The p-values for the remaining experiments are approximately equal to or less than 0.1, which implies the null hypothesis is not equal to 0 and is not due to chance, but rather an outside force (in this case the PHM framework). The results of the experiments are in Appendix B and the R code used to perform the t-tests is in Appendix C.

5.2 Intense Scan

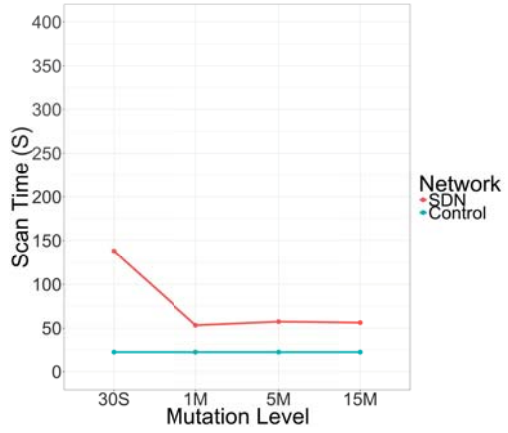
Scan Time.

The scan time for the intense scans is much more erratic than the quick scans for the same experiments, due mainly to Nmap crashing during the thirty second mutation rates

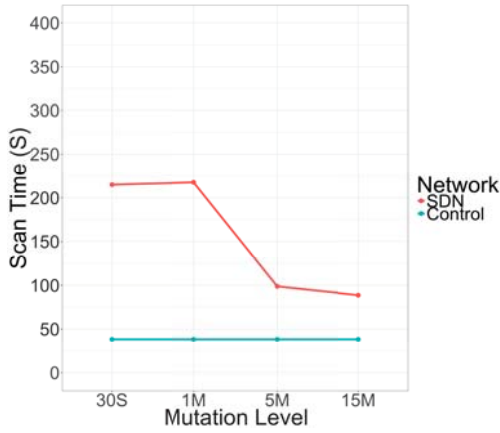
and timing out during the one minute mutation rates for the experiments with thirty, forty, and fifty hosts. Figure 14 shows a side by side comparison of the scan times for the traditional (control) network and the PHM (SDN) network. Figure 15 shows the confidence intervals from the t-test conducted for each run. In every experiment, where Nmap did not crash, the SDN scan took a significantly longer time to complete, as displayed in Tables 6, 7, 8, 9, 10, and 11 at the end of this section. The general trend for each experiment is a spike in the scan time for either the thirty second or one minute mutation rates, due to Nmap crashing or timing out, and then leveling out for the five and fifteen minute mutations, when Nmap is able to complete the scan.



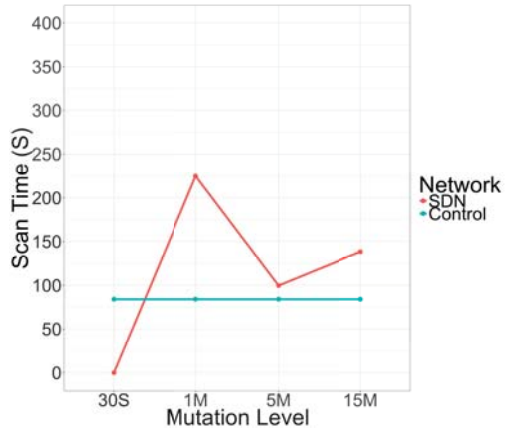
(a) 5 Hosts



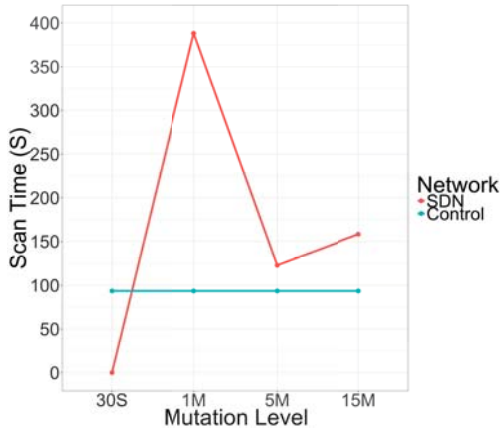
(b) 10 Hosts



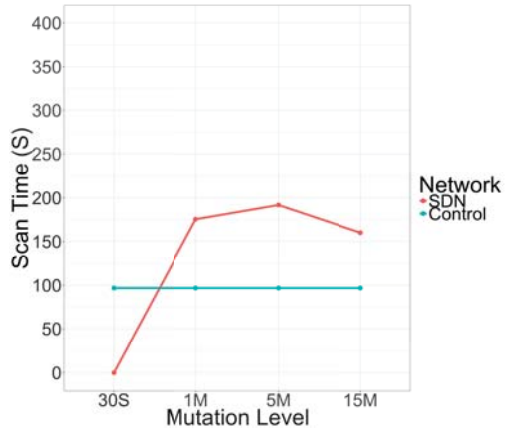
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 14. Scan Time Observed During Intense Scans

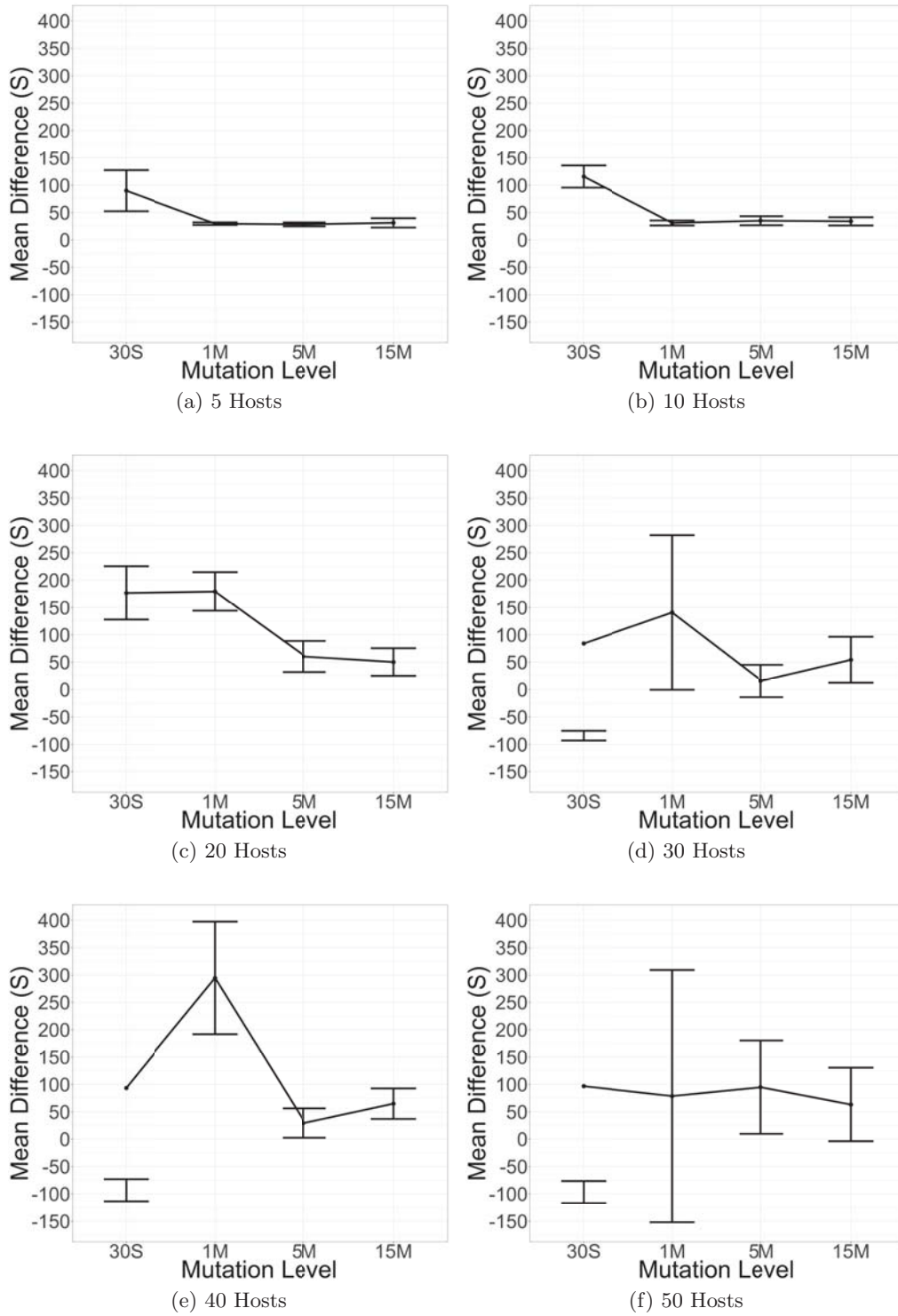
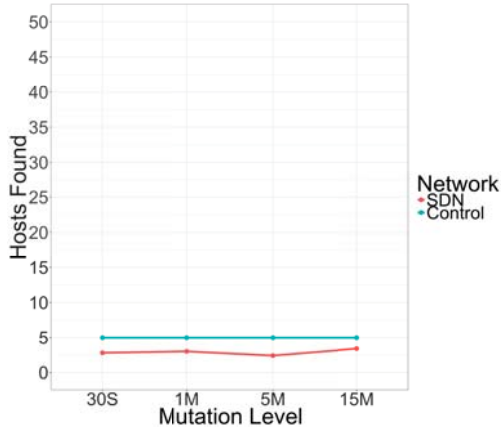


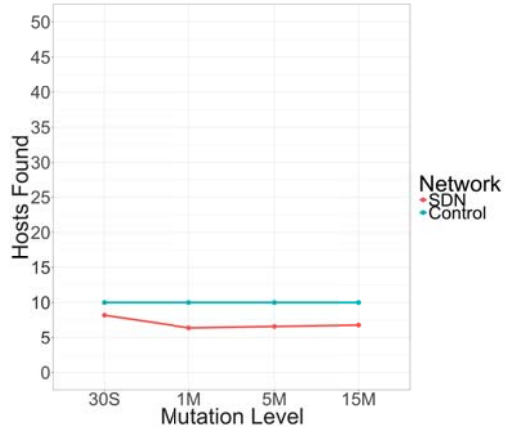
Figure 15. Confidence Intervals of Mean Intense Scan Time Difference between Traditional and SDN Networks

Hosts Found.

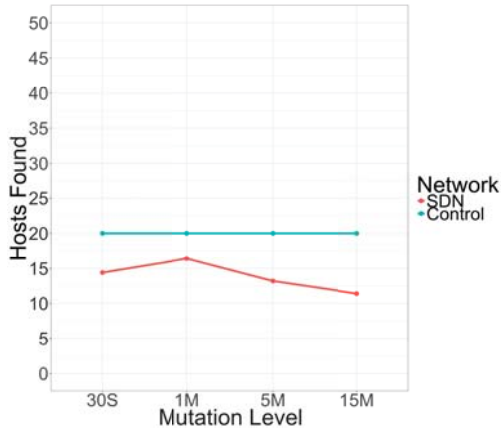
As expected, in the traditional network, in almost every experiment Nmap is able to find all the hosts and when it did not, it found all but one. The PHM (SDN) network scans are less consistent for both the intense and quick scans. The traditional network can easily be modeled with a simple linear growth, where the number of hosts found is directly proportional to the number of live hosts; but the PHM network does not exhibit the same growth. In addition to not finding any hosts for three experiments, the intense scan is not able to find more than thirty hosts, as seen in Figure 16, which shows a comparison between the traditional network and the PHM network. Figure 17 displays the confidence intervals from the t-tests performed on the six intense scan experiments.



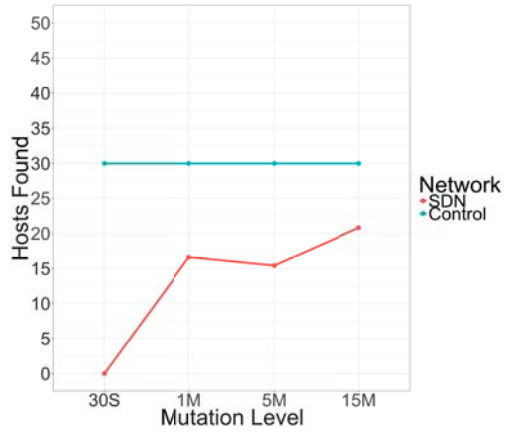
(a) 5 Hosts



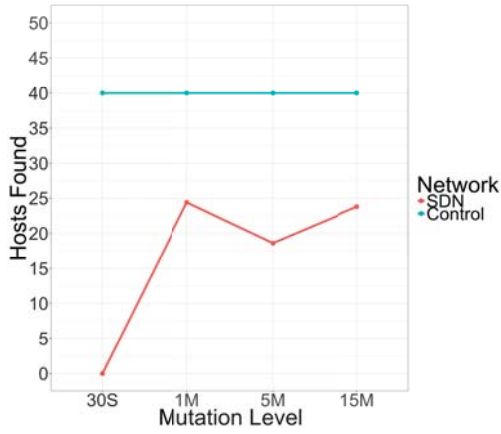
(b) 10 Hosts



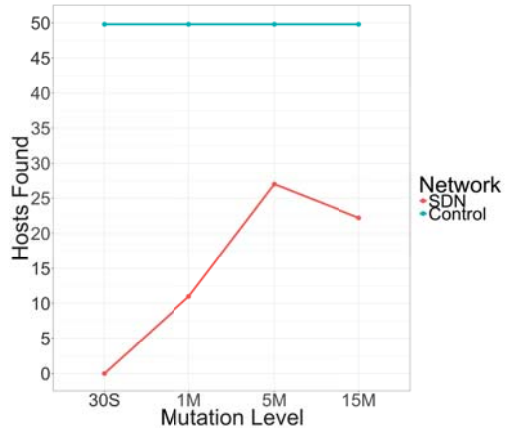
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 16. Number of Hosts Found During Intense Scans

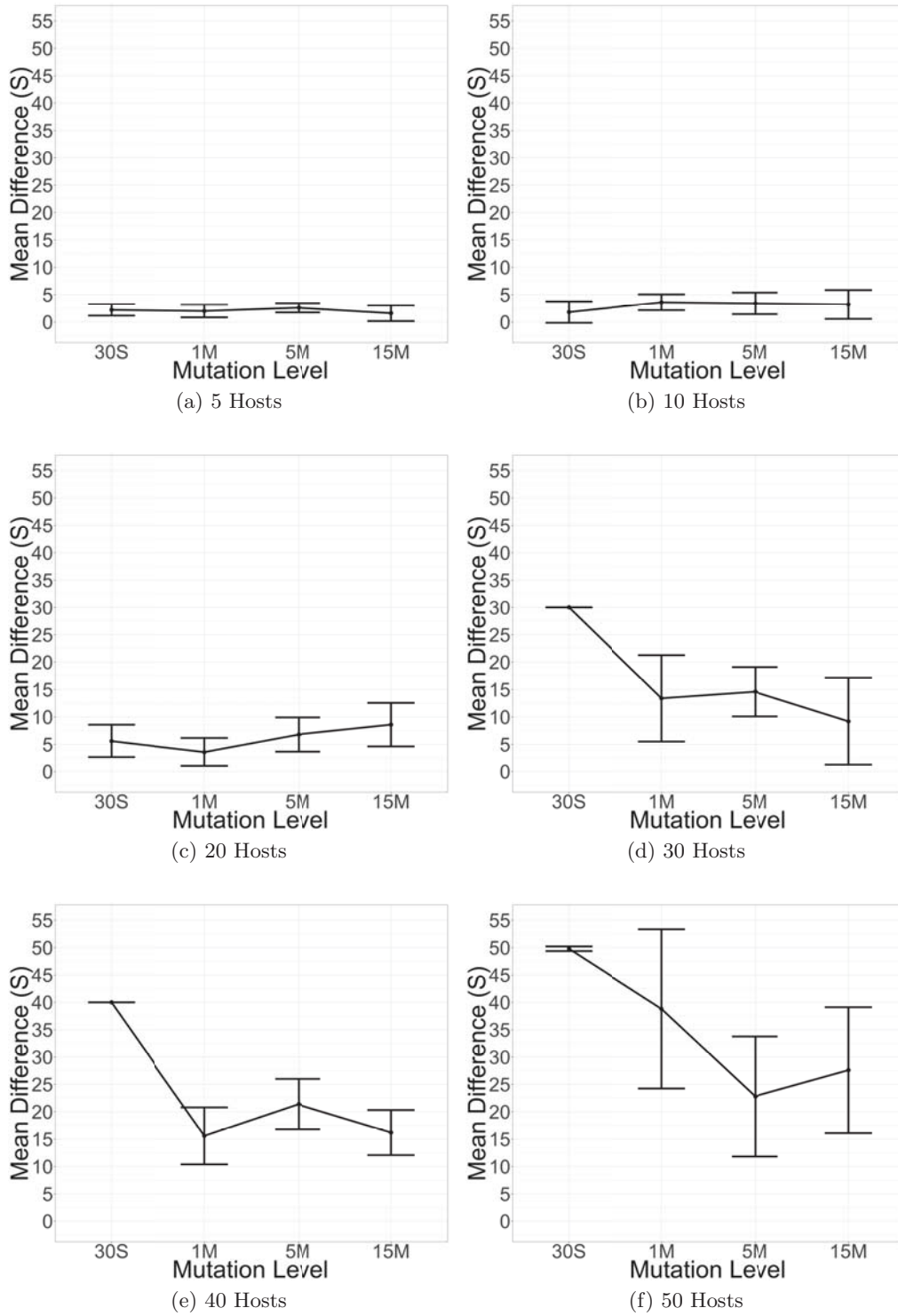
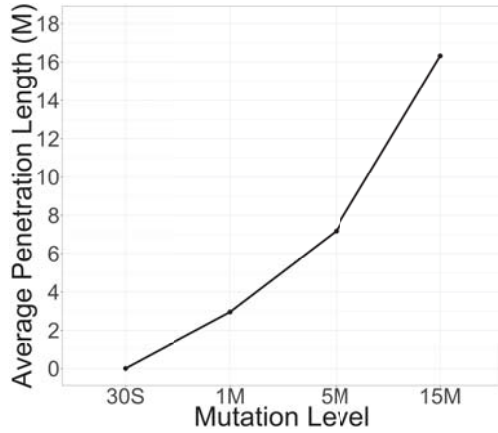


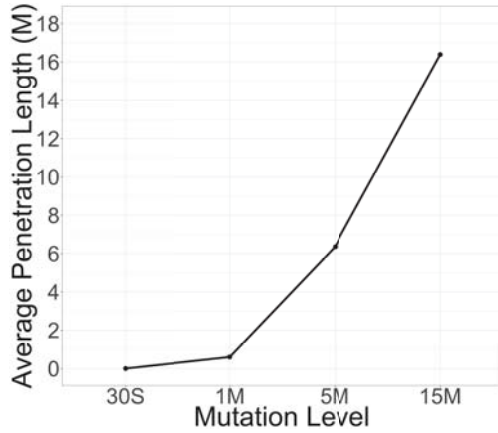
Figure 17. Confidence Intervals of Mean Intense Scan Host Difference between Traditional and SDN Networks

Penetration Length.

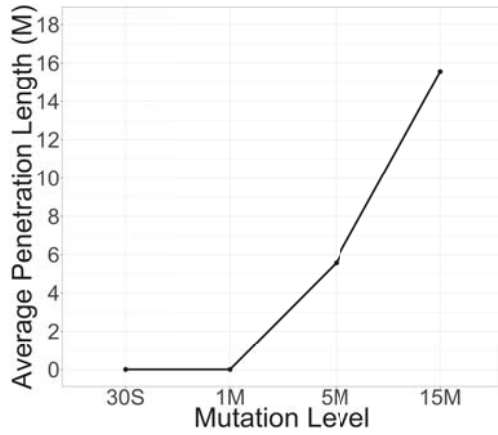
The time the attacking machine is able to maintain a connection to a host machine is directly related to the time of the mutation, as seen in Figure 18. The connection did not immediately terminate when the mutation occurred; this is because of the relationship between the attacking machine and the host machine. In a perfect world once the connection between the two is no longer needed, the machines would negotiate a disconnection. Because PHM is attempting to disrupt attackers during their scans or attacks, it is specifically designed to prevent this negotiation by immediately severing the connection between the machines with the vIP mutation. Because of this, the attacking machine tries to reestablish the connection for several minutes after the mutation until it gives up. To be consistent, the time measurement is taken when the attacking machine accepted the connection is dead. This explains the variability in Figure 18. Figure 19 displays the confidence intervals from the t-test comparing the two networks. Because the connection to the exploited host machine never ended in the traditional network, a value of six hours (which is well outside the longest mutation range of thirty minutes) was used so that a t-test could be conducted. The confidence intervals have a width of at most 10 seconds, but appear to have a downward trend. This can be explained by the fact that the penetration length increases as the mutation level increases, which lowers the mean difference from six hours.. This brings the average difference between the two networks down. This, combined with extremely low p-values, less than 1×10^{-12} , clearly concludes there is a difference.



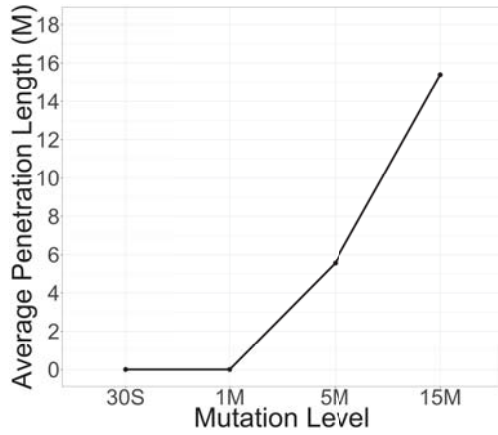
(a) 5 Hosts



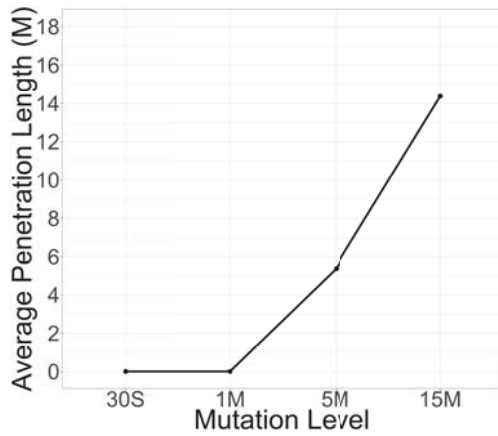
(b) 10 Hosts



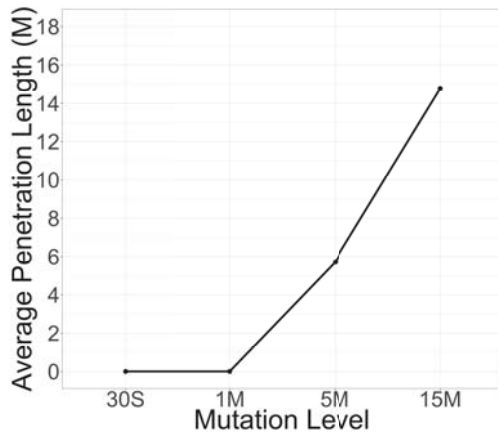
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 18. Length of Penetration Attack during Intense Scans

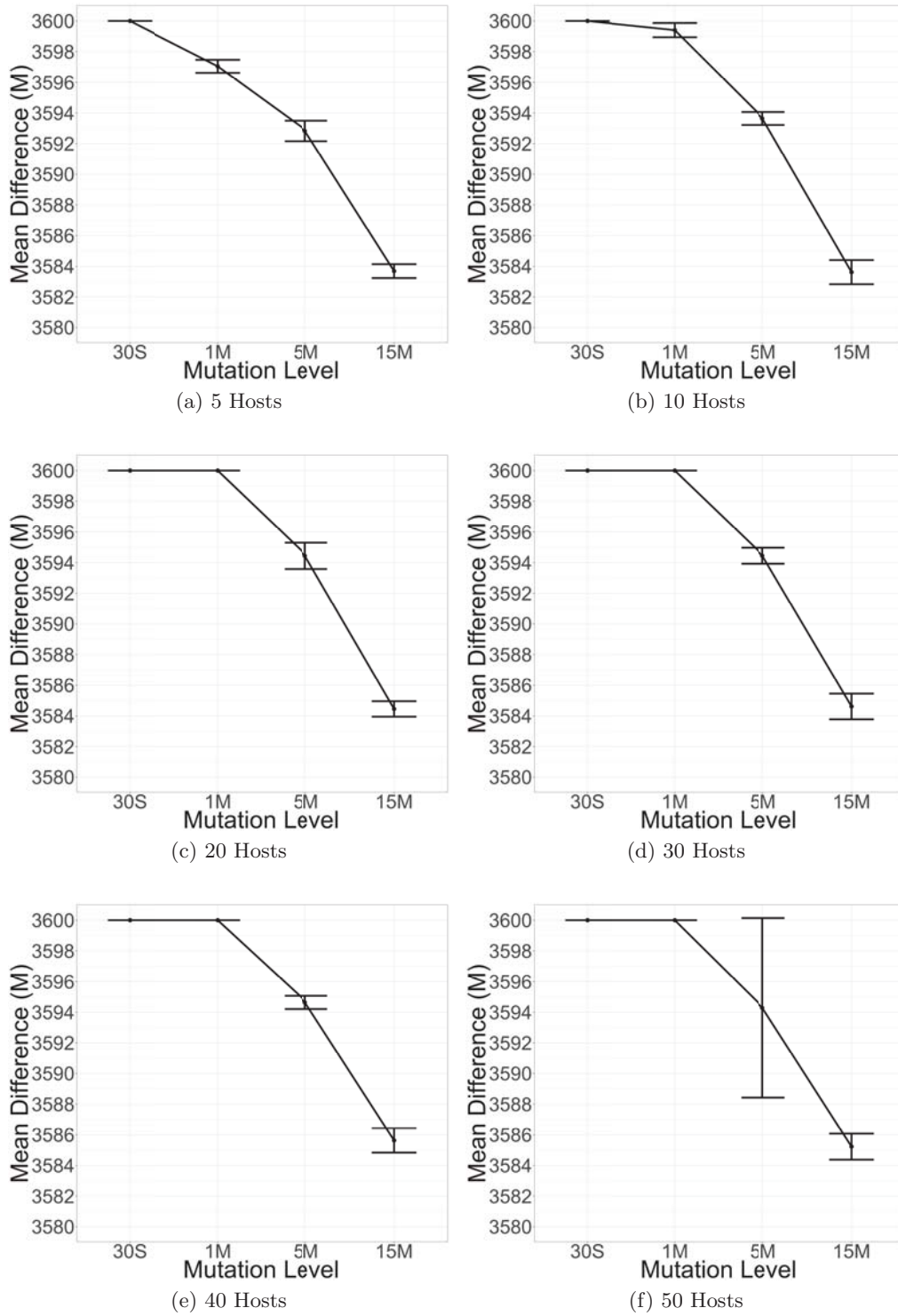


Figure 19. Confidence Intervals of Mean Intense Scan Penetration Length Difference between Traditional and SDN Networks

Results - 5 Hosts.

As with the other experiments, Nmap is not able to successfully perform the intense scan in thirty seconds and timed out after an extended amount of time, leading to a wide range in the confidence interval range and large X Mean as shown in Table 6 for the thirty second mutation. The remainder of the experiments for the five host configuration produced expected results; low enough p-values to rule out randomness and evidence to support that the PHM is significantly different than the traditional network.

Table 6. Intense Scan - 5 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
5H30S – Scan	5.113	4.018	$6.841 \cdot 10^{-3}$	52.515	127.469	106.874	16.882
5H30S – Host	4.491	4.000	$1.09 \cdot 10^{-2}$	1.156	3.244	2.800	5.000
5H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
5H1M – Scan	24.223	7.910	$1.052 \cdot 10^{-8}$	27.529	32.115	46.704	16.882
5H1M – Host	3.651	4.000	$2.174 \cdot 10^{-2}$	0.832	3.168	3.000	5.000
5H1M – Pen	18,120.276	4.000	$5.565 \cdot 10^{-17}$	3,596.623	3,597.469	2.954	3,600.000
5H5M – Scan	15.968	6.003	$3.816 \cdot 10^{-6}$	25.085	32.035	45.442	16.882
5H5M – Host	6.500	4.000	$2.89 \cdot 10^{-3}$	1.747	3.453	2.400	5.000
5H5M – Pen	11,459.348	4.000	$3.479 \cdot 10^{-16}$	3,592.162	3,593.498	7.170	3,600.000
5H15M – Scan	7.684	4.338	$1.104 \cdot 10^{-3}$	22.868	39.900	48.266	16.882
5H15M – Host	2.359	4.000	$7.774 \cdot 10^{-2}$	0.154	3.046	3.400	5.000
5H15M – Pen	16,917.340	4.000	$7.325 \cdot 10^{-17}$	3,583.230	3,584.134	16.318	3,600.000

Results - 10 Hosts.

The ten host experiments, Table 7, yielded similar results to the five host experiments where the thirty second scan revealed a large mean and a wide confidence interval because of Nmap. This configuration differed with the t-test conducted for the number of hosts the thirty second scan found. The test shows a much higher p-value than the others, above 0.1, which indicates this result could be due, at least in part, to randomness rather than the PHM framework. It is not clear what caused the higher p-value, as there is no clear explanation seen by looking at the data collected, seen in Appendix B.

Results - 20 Hosts.

As can be seen in Figure 14, as the number of hosts increases so does the time it takes for the intense scan to complete. Because of this increase in time, the configuration with

Table 7. Intense Scan - 10 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
10H30S – Scan	12.115	4.176	$2.074 \cdot 10^{-4}$	95.650	135.910	137.964	22.184
10H30S – Host	1.964	4.000	0.121	−0.154	3.754	8.200	10.000
10H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
10H1M – Scan	13.143	7.337	$2.305 \cdot 10^{-6}$	26.703	35.625	53.348	22.184
10H1M – Host	5.308	4.000	$6.055 \cdot 10^{-3}$	2.154	5.046	6.400	10.000
10H1M – Pen	16,504.074	4.000	$8.087 \cdot 10^{-17}$	3,598.937	3,599.867	0.598	3,600.000
10H5M – Scan	8.689	5.066	$3.128 \cdot 10^{-4}$	27.097	43.399	57.432	22.184
10H5M – Host	3.666	4.000	$2.146 \cdot 10^{-2}$	1.423	5.377	6.600	10.000
10H5M – Pen	18,100.842	4.000	$5.589 \cdot 10^{-17}$	3,593.221	3,594.067	6.356	3,600.000
10H15M – Scan	9.106	5.267	$2.03 \cdot 10^{-4}$	26.675	41.621	56.332	22.184
10H15M – Host	2.579	4.000	$6.142 \cdot 10^{-2}$	0.554	5.846	6.800	10.000
10H15M – Pen	9,735.853	4.000	$6.678 \cdot 10^{-16}$	3,582.829	3,584.399	16.386	3,600.000

twenty hosts is where Nmap begins to struggle to complete a full scan before the mutation during a one minute configuration. This leads to a wide confidence interval for both the thirty second and one minute configurations as seen in Table 8. The results for the five and fifteen minute configurations are more in line with the expected results.

Table 8. Intense Scan - 20 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
20H30S – Scan	7.714	4.005	$1.513 \cdot 10^{-3}$	127.835	225.437	214.958	38.322
20H30S – Host	4.000	4.000	$1.613 \cdot 10^{-2}$	2.615	8.585	14.400	20.000
20H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
20H1M – Scan	10.825	4.009	$4.081 \cdot 10^{-4}$	143.990	214.550	217.592	38.322
20H1M – Host	2.979	4.000	$4.076 \cdot 10^{-2}$	1.024	6.176	16.400	20.000
20H1M – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
20H5M – Scan	4.537	4.014	$1.044 \cdot 10^{-2}$	32.069	88.811	98.762	38.322
20H5M – Host	4.648	4.000	$9.673 \cdot 10^{-3}$	3.681	9.919	13.200	20.000
20H5M – Pen	8,917.427	4.000	$9.488 \cdot 10^{-16}$	3,593.583	3,595.301	5.558	3,600.000
20H15M – Scan	4.240	4.018	$1.313 \cdot 10^{-2}$	25.053	75.583	88.640	38.322
20H15M – Host	4.623	4.000	$9.856 \cdot 10^{-3}$	4.635	12.565	11.400	20.000
20H15M – Pen	15,145.164	4.000	$1.14 \cdot 10^{-16}$	3,583.943	3,584.953	15.552	3,600.000

Results - 30 Hosts.

The thirty host configuration is where Nmap completely crashes during the thirty second scan, leading to a zero second scan and zero hosts found. Table 9 shows the result of this with a negative confidence interval for the thirty second scan and 30.000 for the higher and lower bounds of the confidence interval for the number of hosts found because the traditional network found exactly thirty for all five runs. Both the one minute and five

minute configurations have a negative lower bound, but just as with the results of the ten host experiments, in both instances there is a single run that is significantly different than the others leading to the conclusion that the results for those experiments might be due, in some part, to chance rather than exclusively because of the PHM.

Table 9. Intense Scan - 30 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
30H30S – Scan	–20.289	4.000	$3.484 \cdot 10^{-5}$	–92.928	–75.256	$2 \cdot 10^{-7}$	84.092
30H30S – Host	$1.5 \cdot 10^8$	4.000	$1.185 \cdot 10^{-32}$	30.000	30.000	$2 \cdot 10^{-7}$	30.000
30H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
30H1M – Scan	2.119	4.031	0.101	–0.513	281.981	224.826	84.092
30H1M – Host	3.626	4.000	$2.225 \cdot 10^{-2}$	5.521	21.279	16.600	30.000
30H1M – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
30H5M – Scan	1.076	4.706	0.334	–14.018	45.234	99.700	84.092
30H5M – Host	6.913	4.000	$2.297 \cdot 10^{-3}$	10.098	19.102	15.400	30.000
30H5M – Pen	14,671.836	4.000	$1.295 \cdot 10^{-16}$	3,593.928	3,594.972	5.550	3,600.000
30H15M – Scan	2.690	4.351	$4.999 \cdot 10^{-2}$	12.255	96.357	138.398	84.092
30H15M – Host	2.464	4.000	$6.939 \cdot 10^{-2}$	1.240	17.160	20.800	30.000
30H15M – Pen	9,137.582	4.000	$8.606 \cdot 10^{-16}$	3,583.776	3,585.448	15.388	3,600.000

Results - 40 Hosts.

The results for the forty host experiments, shown in Table 10, displays the same characteristics as the thirty host experiment, namely that the results for the thirty second and one minute configurations are unreliable because Nmap did not run to completion. The results for the other experiments demonstrate the expected results of low p-values and tighter confidence intervals.

Table 10. Intense Scan - 40 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
40H30S – Scan	–9.854	4.000	$5.95 \cdot 10^{-4}$	–113.719	–73.265	$2 \cdot 10^{-7}$	93.492
40H30S – Host	$2 \cdot 10^8$	4.000	$3.75 \cdot 10^{-33}$	40.000	40.000	$2 \cdot 10^{-7}$	40.000
40H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
40H1M – Scan	5.985	4.308	$3.081 \cdot 10^{-3}$	191.772	397.368	388.062	93.492
40H1M – Host	6.390	4.000	$3.079 \cdot 10^{-3}$	10.395	20.805	24.400	40.000
40H1M – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
40H5M – Scan	2.025	7.840	$7.822 \cdot 10^{-2}$	2.322	56.370	122.838	93.492
40H5M – Host	9.913	4.000	$5.813 \cdot 10^{-4}$	16.798	26.002	18.600	40.000
40H5M – Pen	17,784.345	4.000	$5.998 \cdot 10^{-17}$	3,594.207	3,595.069	5.362	3,600.000
40H15M – Scan	4.374	7.740	$2.562 \cdot 10^{-3}$	37.213	92.675	158.436	93.492
40H15M – Host	8.377	4.000	$1.111 \cdot 10^{-3}$	12.077	20.323	23.800	40.000
40H15M – Pen	9,623.498	4.000	$6.996 \cdot 10^{-16}$	3,584.834	3,586.422	14.372	3,600.000

Results - 50 Hosts.

The fifty host experiments display similar characteristics to the thirty and forty runs-wide and/or negative confidence intervals for the thirty second and one minute configurations because of Nmap. In addition, the fifteen minute scan has a run that completed in half the time of the other runs, causing uncertainty in the results seen in Table 11.

Table 11. Intense Scan - 50 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
50H30S – Scan	−10.220	4.000	$5.165 \cdot 10^{-4}$	−116.972	−76.596	$2 \cdot 10^{-7}$	96.784
50H30S – Host	249.000	4.000	$1.561 \cdot 10^{-9}$	49.374	50.226	$2 \cdot 10^{-7}$	49.800
50H30S – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
50H1M – Scan	0.725	4.061	0.508	−151.884	309.348	175.516	96.784
50H1M – Host	5.681	4.007	$4.713 \cdot 10^{-3}$	24.248	53.352	11.000	49.800
50H1M – Pen	$1.8 \cdot 10^{10}$	4.000	$5.716 \cdot 10^{-41}$	3,600.000	3,600.000	$2 \cdot 10^{-7}$	3,600.000
50H5M – Scan	2.305	4.445	$7.588 \cdot 10^{-2}$	9.601	180.315	191.742	96.784
50H5M – Host	4.426	4.012	$1.138 \cdot 10^{-2}$	11.827	33.773	27.000	49.800
50H5M – Pen	1,308.225	4.000	$2.048 \cdot 10^{-12}$	3,588.425	3,600.139	5.718	3,600.000
50H15M – Scan	1.923	4.716	0.116	−3.938	130.538	160.084	96.784
50H15M – Host	5.109	4.011	$6.886 \cdot 10^{-3}$	16.093	39.107	22.200	49.800
50H15M – Pen	8,981.435	4.000	$9.221 \cdot 10^{-16}$	3,584.373	3,586.075	14.776	3,600.000

Intense Scan Summary.

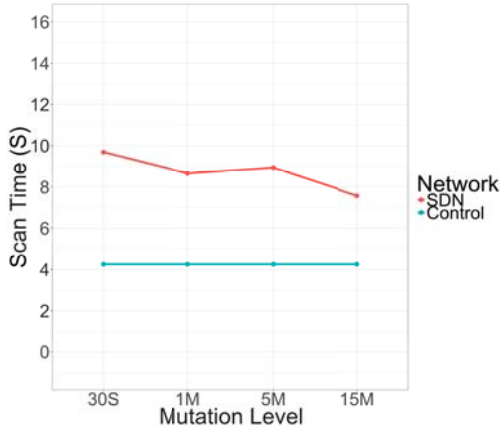
The intense scan reveals the power of PHM, specifically how it can either prevent an attacker from completing their network scan or significantly increasing the time it takes for them to conduct a scan (while limiting the number of hosts found). This is precisely the goal of PHM, because with this system in place, an attacker will either give up or be forced to increase the rate at which they are scanning to garner the same amount of information. The scans conducted in these experiments ran at full speed, meaning an adversary running slower scans can be expected to experience significant obstruction to their scans.

5.3 Quick Scan

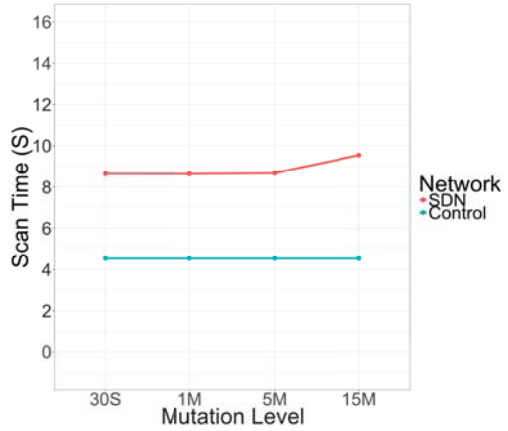
Scan Time.

The quick scan yielded more consistent results than the intense scan, as can be seen in Figure 20. At first glance, it appears somewhat erratic, but this is simply because the

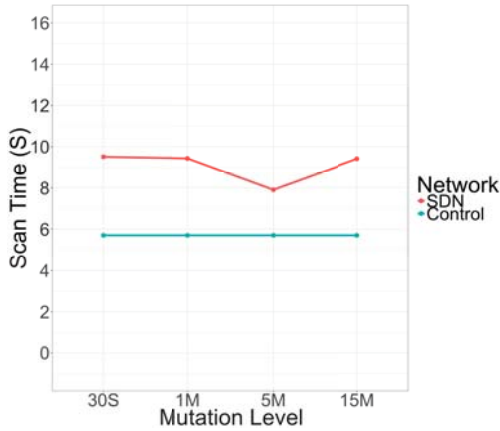
quick scans finished so much faster than the intense scans, thereby changing the scale on the graph. Even at the most extreme, the largest difference between the traditional and software-defined network scans is only fifteen seconds. As with the intense scan though, the scan time for the SDN is clearly higher than the traditional network, as confirmed by the t-tests, seen in Tables 13, 14, 15, 16, and 17. Figure 20 presents a clear picture of the time growth seen in the quick scans as the number of hosts increased, while Figure 21 shows the confidence intervals resulting from the t-tests for the quick scan results.



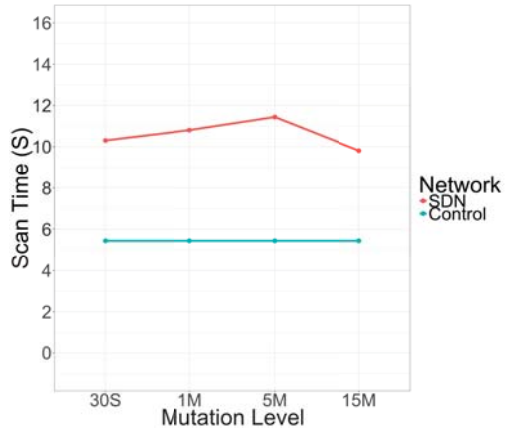
(a) 5 Hosts



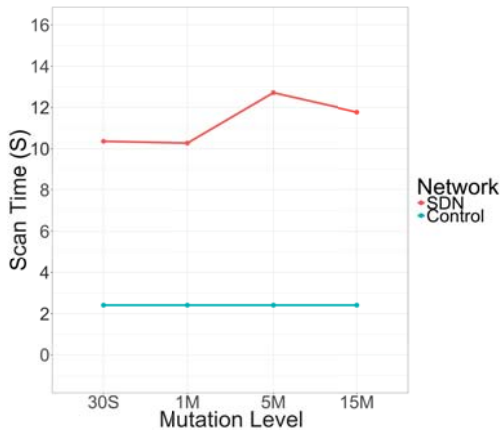
(b) 10 Hosts



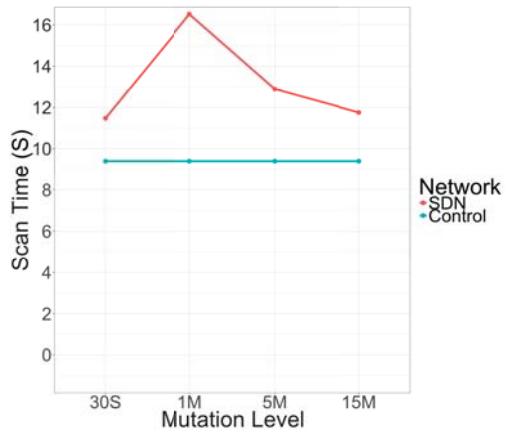
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 20. Scan Time Observed During Quick Scans

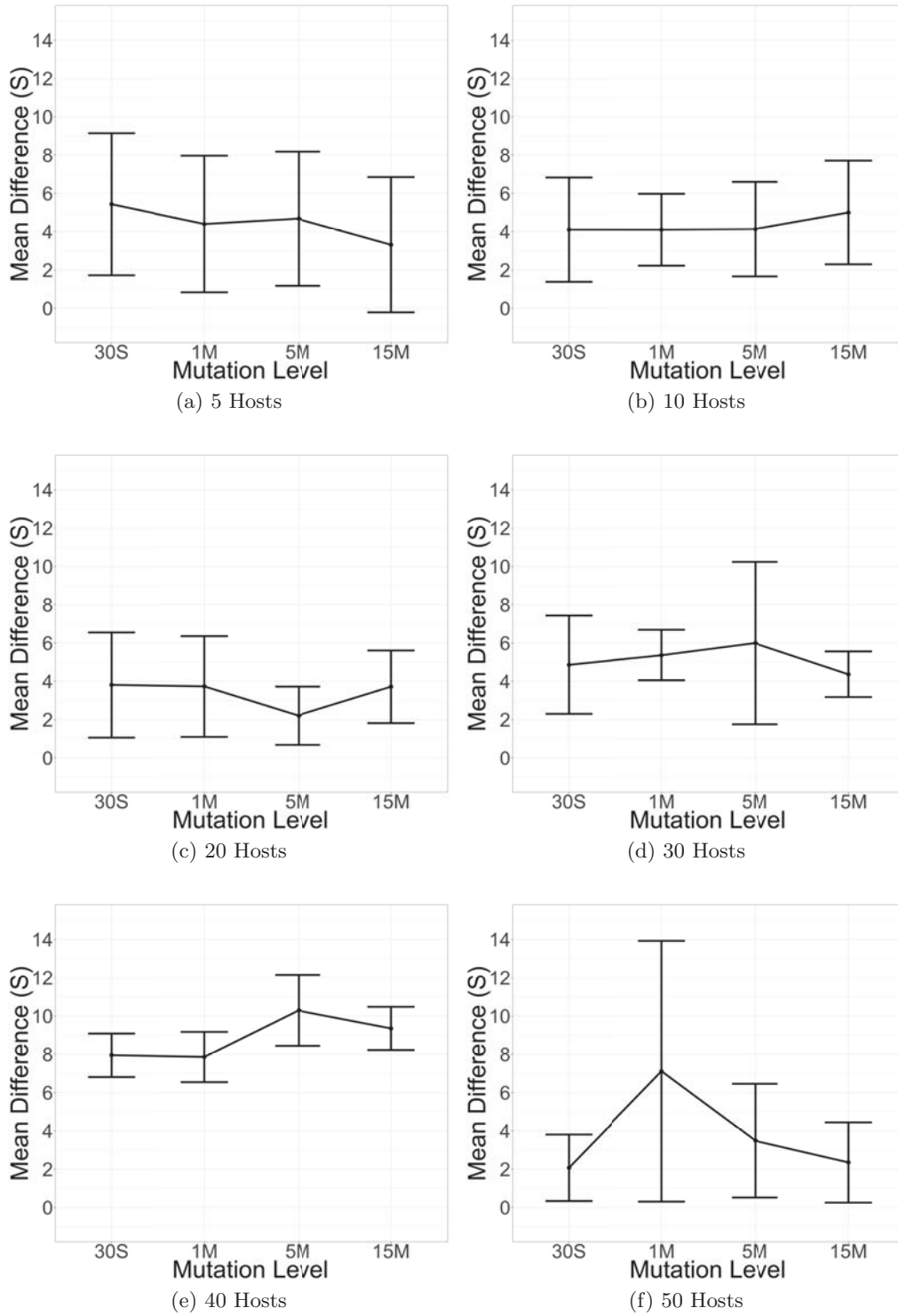
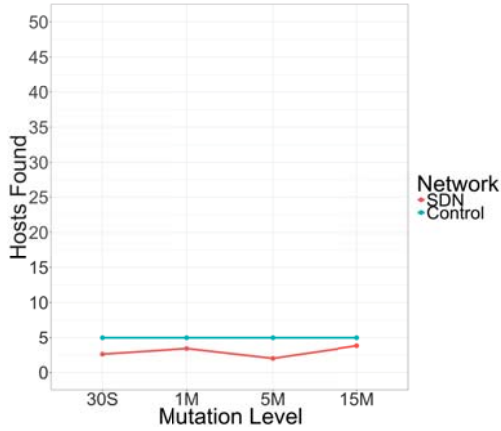


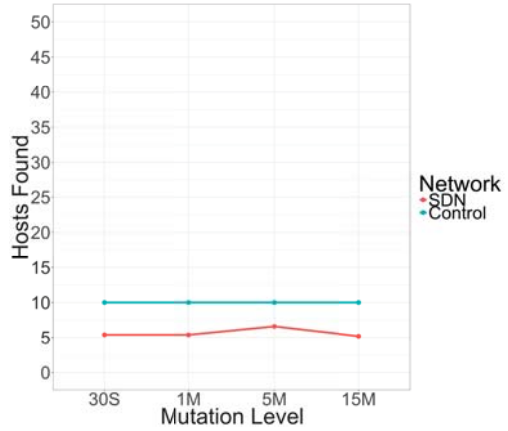
Figure 21. Confidence Intervals of Mean Quick Scan Time Difference between Traditional and SDN Networks

Hosts Found.

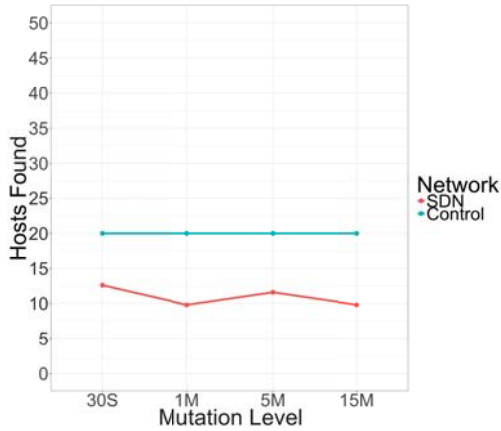
The number of hosts the quick scans were able to identify follow a similar pattern to that of the intense hosts, in that every host is found in almost all scans of the traditional network, while the number of hosts found in the software-defined network follow a more shallower growth pattern. These results are displayed in Figure 22. The confidence intervals for the number of hosts found, seen in Figure 23, clearly shows same trend seen in the intense scans, that as the number of hosts increases the difference in the number of hosts found also increases.



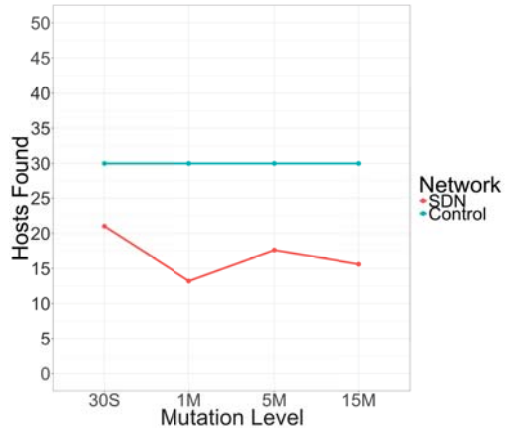
(a) 5 Hosts



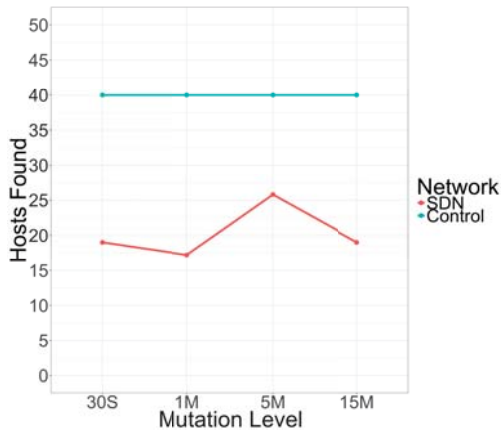
(b) 10 Hosts



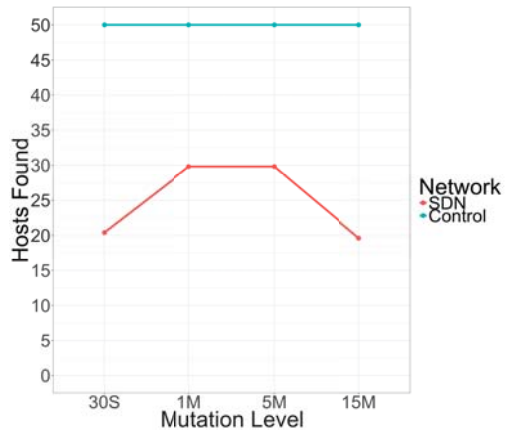
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 22. Number of Hosts Found During Quick Scans

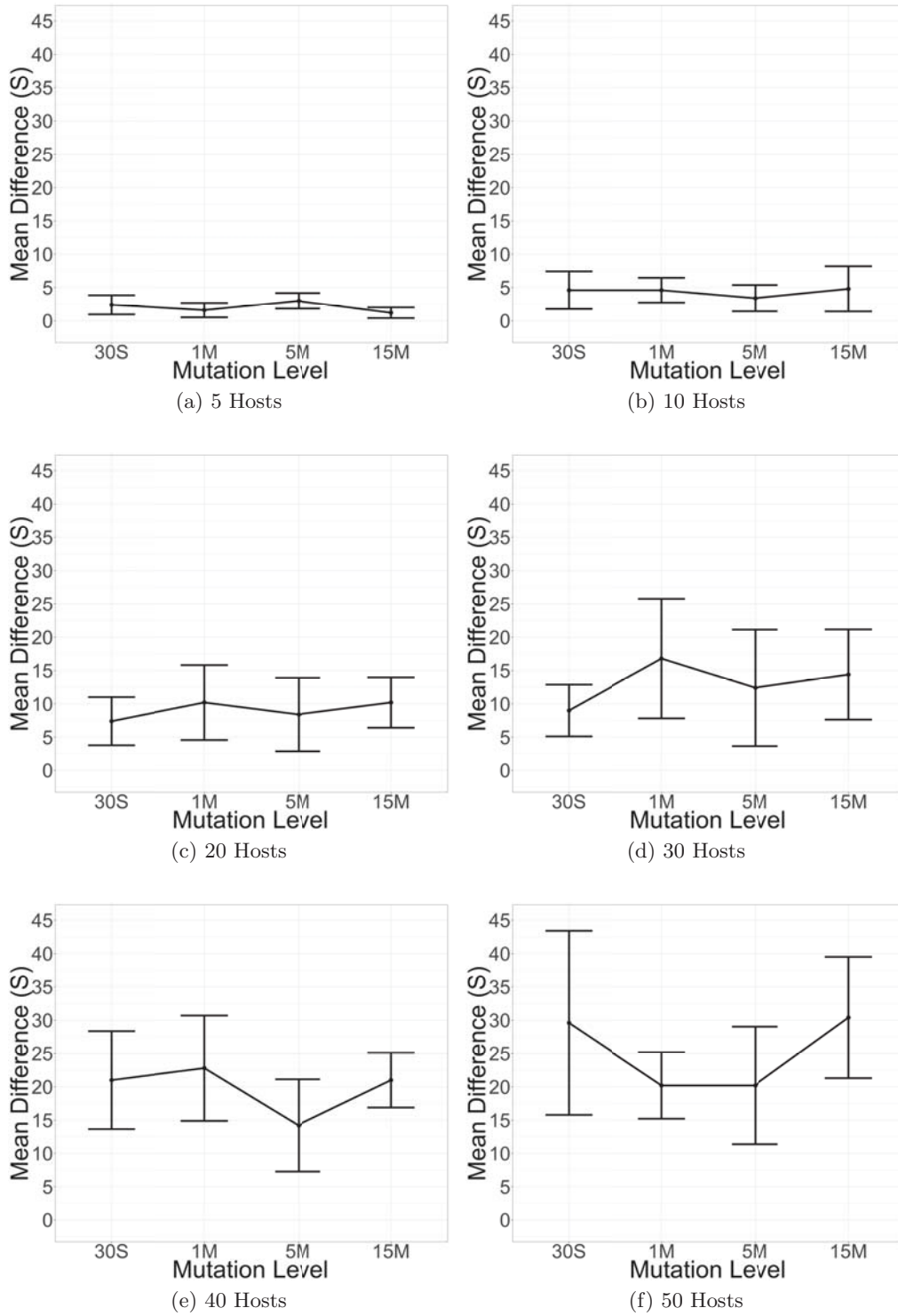
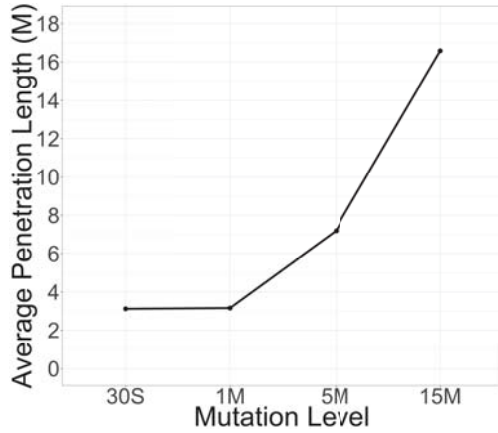


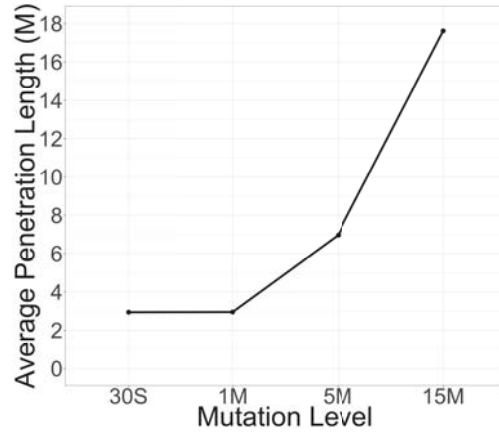
Figure 23. Confidence Intervals of Mean Quick Scan Host Difference between Traditional and SDN Networks

Penetration Length.

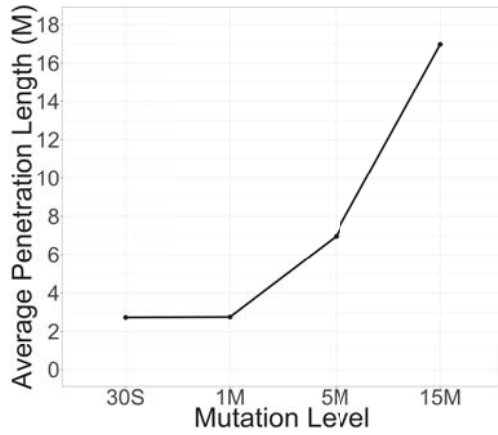
As shown in Figure 24, the penetration length for the quick scan experiments follows the expected pattern- the connection lasts a length of time proportional to the mutation rate (i.e., the one minute mutation produces approximately a three minute penetration length). As explained in section 1.2.3, this is due to the fact that even though the connection between the attacking machine and the target host has been lost, the attacking machine attempts to reconnect for approximately two minutes, and then reports the connection has been killed. Figure 24 displays the confidence intervals from the t-tests comparing the penetration length of the two networks. Because the connection to the exploited host machine never ended in the traditional network, a value of six hours (which is well outside the longest mutation range of thirty minutes) was used so that a t-test could be conducted. The confidence intervals have a width of at most 10 seconds, but appear to have a downward trend. This can be explained by the fact that the penetration length increases as the mutation level increases, which lowers the mean difference from six hours. This brings the average difference between the two networks down. This, combined with extremely low p-values, less than 1×10^{-12} , clearly concludes there is a difference.



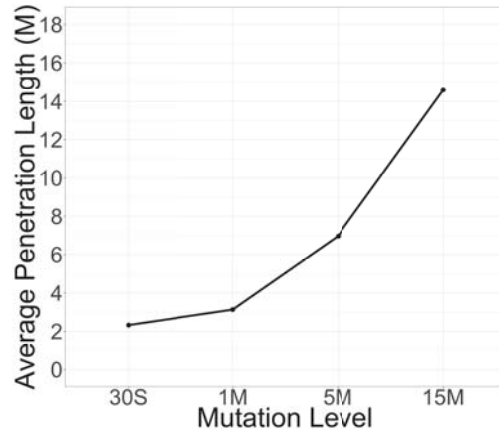
(a) 5 Hosts



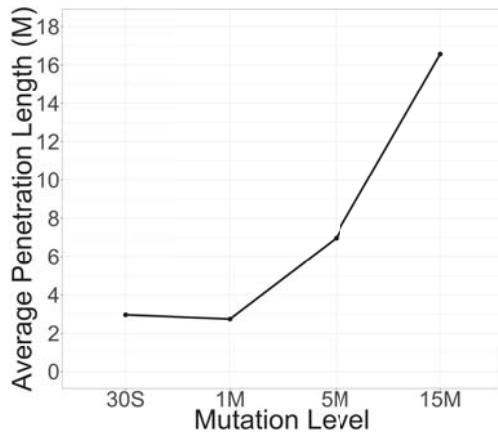
(b) 10 Hosts



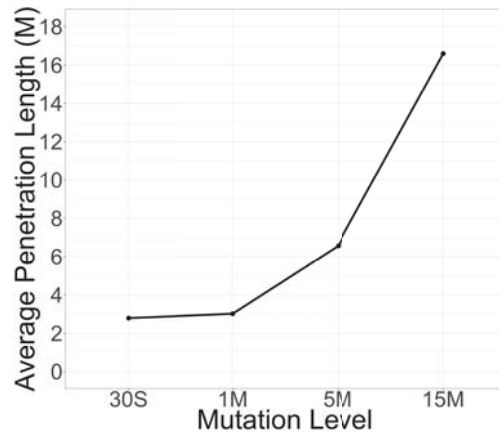
(c) 20 Hosts



(d) 30 Hosts

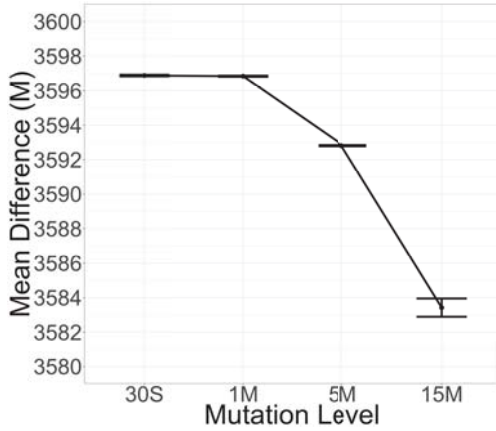


(e) 40 Hosts

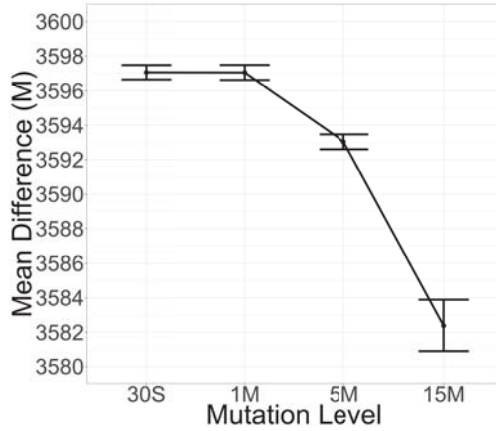


(f) 50 Hosts

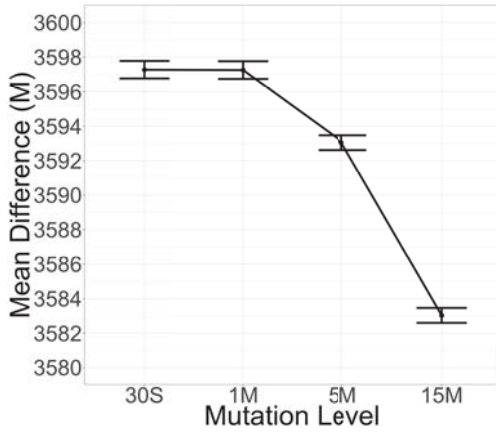
Figure 24. Length of Penetration Attack during Quick Scans



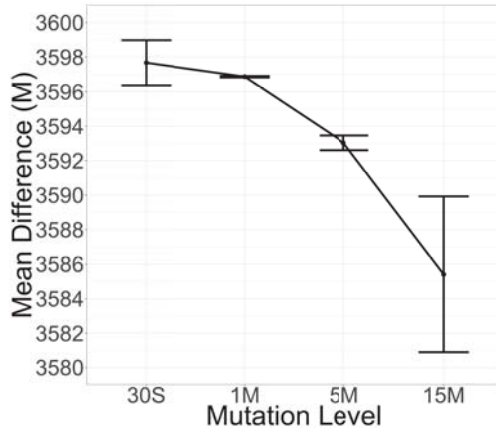
(a) 5 Hosts



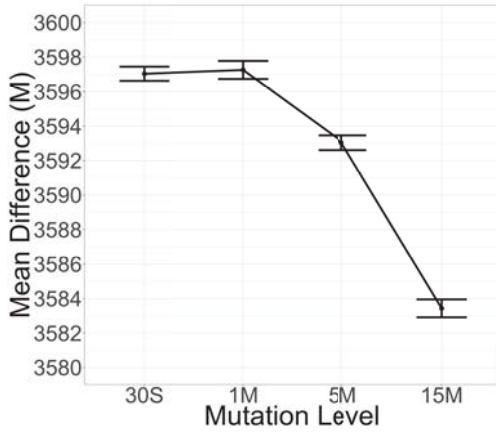
(b) 10 Hosts



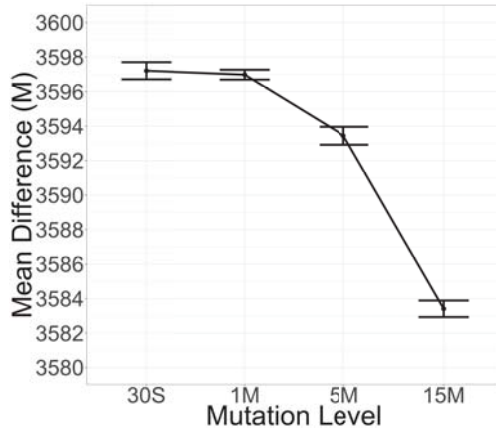
(c) 20 Hosts



(d) 30 Hosts



(e) 40 Hosts



(f) 50 Hosts

Figure 25. Confidence Intervals of Mean Quick Scan Penetration Length Difference between Traditional and SDN Networks

Results - 5 Hosts.

Because the quick scan runs faster than the intense scan, the results seen in the t-tests have far fewer extraneous data points. For the five host configuration, Table 12, all scans except one confirm the hypothesis of the two networks being different, with very little chance of the results being due to randomness (i.e., very low p-values and tight confidence intervals). The fifteen minute scan has a slightly higher chance of randomness, seen by a p-value of 0.118, which contributed to the negative lower bound for the corresponding confidence interval. Once again, the negative lower bound means it is not certain that zero is not the average, but the data is still useful in the context of the other experiments.

Table 12. Quick Scan - 5 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
5H30S – Scan	2.766	7.188	$2.712 \cdot 10^{-2}$	1.728	9.148	9.692	4.254
5H30S – Host	3.539	4.000	$2.404 \cdot 10^{-2}$	0.954	3.846	2.600	5.000
5H30S – Pen	$1.659 \cdot 10^5$	4.000	$7.919 \cdot 10^{-21}$	3,596.834	3,596.926	3.120	3,600.000
5H1M – Scan	2.379	6.358	$5.25 \cdot 10^{-2}$	0.843	7.961	8.656	4.254
5H1M – Host	3.138	4.000	$3.492 \cdot 10^{-2}$	0.513	2.687	3.400	5.000
5H1M – Pen	$2.372 \cdot 10^5$	4.000	$1.896 \cdot 10^{-21}$	3,596.808	3,596.872	3.160	3,600.000
5H5M – Scan	2.606	5.901	$4.098 \cdot 10^{-2}$	1.181	8.191	8.940	4.254
5H5M – Host	5.477	4.000	$5.408 \cdot 10^{-3}$	1.832	4.168	2.000	5.000
5H5M – Pen	$2.339 \cdot 10^5$	4.000	$2.006 \cdot 10^{-21}$	3,592.783	3,592.849	7.184	3,600.000
5H15M – Scan	1.816	6.149	0.118	-0.217	6.849	7.570	4.254
5H15M – Host	3.207	4.000	$3.268 \cdot 10^{-2}$	0.402	1.998	3.800	5.000
5H15M – Pen	14,500.526	4.000	$1.357 \cdot 10^{-16}$	3,582.893	3,583.947	16.580	3,600.000

Results - 10 Hosts.

The ten host configuration exemplifies the differences between the two networks with all scans having low p-values and close confidence intervals. The results for these experiments can be seen in Table 13.

Results - 20 Hosts.

The twenty host configuration is similar to the ten, in that the data is exactly what is expected. The results are listed in Table 14.

Table 13. Quick Scan - 10 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
10H30S – Scan	3.151	4.324	$3.096 \cdot 10^{-2}$	1.388	6.828	8.648	4.540
10H30S – Host	3.467	4.000	$2.564 \cdot 10^{-2}$	1.772	7.428	5.400	10.000
10H30S – Pen	18,347.551	4.000	$5.295 \cdot 10^{-17}$	3,596.638	3,597.474	2.944	3,600.000
10H1M – Scan	4.479	4.681	$7.628 \cdot 10^{-3}$	2.230	5.978	8.644	4.540
10H1M – Host	5.277	4.000	$6.185 \cdot 10^{-3}$	2.741	6.459	5.400	10.000
10H1M – Pen	17,668.703	4.000	$6.156 \cdot 10^{-17}$	3,596.612	3,597.480	2.954	3,600.000
10H5M – Scan	3.485	4.394	$2.167 \cdot 10^{-2}$	1.669	6.599	8.674	4.540
10H5M – Host	3.666	4.000	$2.146 \cdot 10^{-2}$	1.423	5.377	6.600	10.000
10H5M – Pen	17,615.038	4.000	$6.232 \cdot 10^{-17}$	3,592.601	3,593.471	6.964	3,600.000
10H15M – Scan	3.863	4.329	$1.559 \cdot 10^{-2}$	2.301	7.703	9.542	4.540
10H15M – Host	3.012	4.000	$3.948 \cdot 10^{-2}$	1.402	8.198	5.200	10.000
10H15M – Pen	5,082.285	4.000	$8.993 \cdot 10^{-15}$	3,580.877	3,583.883	17.620	3,600.000

Table 14. Quick Scan - 20 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
20H30S – Scan	2.891	4.377	$3.996 \cdot 10^{-2}$	1.069	6.551	9.514	5.704
20H30S – Host	4.376	4.000	$1.191 \cdot 10^{-2}$	3.795	11.005	12.600	20.000
20H30S – Pen	15,143.766	4.000	$1.141 \cdot 10^{-16}$	3,596.768	3,597.780	2.726	3,600.000
20H1M – Scan	2.950	4.411	$3.717 \cdot 10^{-2}$	1.106	6.358	9.436	5.704
20H1M – Host	3.872	4.000	$1.796 \cdot 10^{-2}$	4.584	15.816	9.800	20.000
20H1M – Pen	14,984.128	4.000	$1.19 \cdot 10^{-16}$	3,596.740	3,597.764	2.748	3,600.000
20H5M – Scan	2.909	5.231	$3.172 \cdot 10^{-2}$	0.692	3.720	7.910	5.704
20H5M – Host	3.255	4.000	$3.123 \cdot 10^{-2}$	2.898	13.902	11.600	20.000
20H5M – Pen	17,870.323	4.000	$5.883 \cdot 10^{-17}$	3,592.617	3,593.475	6.954	3,600.000
20H15M – Scan	3.991	4.787	$1.137 \cdot 10^{-2}$	1.821	5.611	9.420	5.704
20H15M – Host	5.756	4.000	$4.518 \cdot 10^{-3}$	6.422	13.978	9.800	20.000
20H15M – Pen	17,680.237	4.000	$6.14 \cdot 10^{-17}$	3,582.598	3,583.462	16.970	3,600.000

Results - 30 Hosts.

Table 15 displays the results of the thirty host experiments and shows they are similar to the twenty and thirty host configurations confirming the hypothesis.

Results - 40 Hosts.

The forty host configuration also meets all expectations and confirms the hypothesis. Results can be seen in Table 16.

Results - 50 Hosts.

The fifty scan configuration follows the pattern of the twenty, thirty, and forty and is displayed in Table 17.

Table 15. Quick Scan - 30 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
30H30S – Scan	3.953	4.354	$1.421 \cdot 10^{-2}$	2.302	7.430	10.306	5.440
30H30S – Host	4.954	4.000	$7.737 \cdot 10^{-3}$	5.127	12.873	21.000	30.000
30H30S – Pen	5,863.994	4.000	$5.074 \cdot 10^{-15}$	3,596.366	3,598.982	2.326	3,600.000
30H1M – Scan	8.069	5.323	$3.513 \cdot 10^{-4}$	4.044	6.688	10.806	5.440
30H1M – Host	3.989	4.000	$1.628 \cdot 10^{-2}$	7.821	25.779	13.200	30.000
30H1M – Pen	$1.88 \cdot 10^5$	4.000	$4.802 \cdot 10^{-21}$	3,596.825	3,596.907	3.134	3,600.000
30H5M – Scan	2.990	4.130	$3.872 \cdot 10^{-2}$	1.761	10.239	11.440	5.440
30H5M – Host	3.029	4.000	$3.882 \cdot 10^{-2}$	3.672	21.128	17.600	30.000
30H5M – Pen	17,917.325	4.000	$5.822 \cdot 10^{-17}$	3,592.610	3,593.466	6.962	3,600.000
30H15M – Scan	7.199	5.619	$4.874 \cdot 10^{-4}$	3.176	5.564	9.810	5.440
30H15M – Host	4.540	4.000	$1.05 \cdot 10^{-2}$	7.638	21.162	15.600	30.000
30H15M – Pen	1,690.243	4.000	$7.351 \cdot 10^{-13}$	3,580.878	3,589.922	14.600	3,600.000

Table 16. Quick Scan - 40 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
40H30S – Scan	13.828	5.434	$1.923 \cdot 10^{-5}$	6.810	9.086	10.360	2.412
40H30S – Host	6.088	4.000	$3.681 \cdot 10^{-3}$	13.646	28.354	19.000	40.000
40H30S – Pen	18,672.385	4.000	$4.936 \cdot 10^{-17}$	3,596.629	3,597.451	2.960	3,600.000
40H1M – Scan	12.017	5.079	$6.353 \cdot 10^{-5}$	6.547	9.173	10.272	2.412
40H1M – Host	6.151	4.000	$3.544 \cdot 10^{-3}$	14.898	30.702	17.200	40.000
40H1M – Pen	14,685.752	4.000	$1.29 \cdot 10^{-16}$	3,596.738	3,597.782	2.740	3,600.000
40H5M – Scan	11.456	4.544	$1.594 \cdot 10^{-4}$	8.446	12.150	12.710	2.412
40H5M – Host	4.374	4.000	$1.193 \cdot 10^{-2}$	7.279	21.121	25.800	40.000
40H5M – Pen	17,910.646	4.000	$5.831 \cdot 10^{-17}$	3,592.610	3,593.466	6.962	3,600.000
40H15M – Scan	16.406	5.460	$7.437 \cdot 10^{-6}$	8.226	10.482	11.766	2.412
40H15M – Host	10.917	4.000	$3.997 \cdot 10^{-4}$	16.899	25.101	19.000	40.000
40H15M – Pen	14,748.999	4.000	$1.268 \cdot 10^{-16}$	3,582.912	3,583.948	16.570	3,600.000

Table 17. Quick Scan - 50 Hosts

Experiment	t – value	Deg.Freedom	p – value	90CILow	90CIHigh	PMean	TMean
50H30S – Scan	2.269	7.052	$5.731 \cdot 10^{-2}$	0.344	3.808	11.474	9.398
50H30S – Host	4.570	4.000	$1.026 \cdot 10^{-2}$	15.791	43.409	20.400	50.000
50H30S – Pen	15,447.929	4.000	$1.054 \cdot 10^{-16}$	3,596.716	3,597.708	2.788	3,600.000
50H1M – Scan	2.167	4.447	$8.928 \cdot 10^{-2}$	0.312	13.924	16.516	9.398
50H1M – Host	8.661	4.000	$9.779 \cdot 10^{-4}$	15.228	25.172	29.800	50.000
50H1M – Pen	26,813.315	4.000	$1.161 \cdot 10^{-17}$	3,596.698	3,597.270	3.016	3,600.000
50H5M – Scan	2.269	6.311	$6.158 \cdot 10^{-2}$	0.528	6.460	12.892	9.398
50H5M – Host	4.879	4.000	$8.165 \cdot 10^{-3}$	11.374	29.026	29.800	50.000
50H5M – Pen	14,667.713	4.000	$1.296 \cdot 10^{-16}$	3,592.918	3,593.962	6.560	3,600.000
50H15M – Scan	2.095	7.928	$6.984 \cdot 10^{-2}$	0.262	4.450	11.754	9.398
50H15M – Host	7.114	4.000	$2.063 \cdot 10^{-3}$	21.290	39.510	19.600	50.000
50H15M – Pen	15,927.988	4.000	$9.322 \cdot 10^{-17}$	3,582.928	3,583.888	16.592	3,600.000

Quick Scan Summary.

While the intense scan experiments exemplified all the positive qualities of PHM, the quick scan experiments were more limited. All the quick scans were able to successfully run

to completion, but just as with the intense scans, very few were able to identify all of the hosts. This robustness is beneficial, and further limits what attackers can accomplish in the network.

VI. Conclusions and Recommendations

6.1 Introduction

This chapter summarizes the research conducted. Section 6.2 provides the conclusions from conducting the research. Section 6.3 provides an overview of how this research integrates with and furthers this field of study. Section 6.4 presents new strands of research that can be followed based on the results of Proactive Host Mutation.

6.2 Research Conclusions

This research is successful in meeting the three goals identified in the design phase. Specifically, the research would be stable, effective, and easy to implement. As hypothesized, the PHM framework created a statistically significant difference in the scan time and hosts found. Because it is not feasible to measure the penetration length of the traditional network due to the fact that it is not known if the connection will ever end, it cannot be said with certainty if the differences in penetration length is statistically significant, but given there is a measurable penetration length for the PHM network it can be assumed the difference is significant. Using the design in Chapter 3 and the goals from Section 3.2, the framework is tested and found to have met all three goals.

Stability.

Initial framework testing showed all fifty hosts were accessible from inside the network and able to maintain regular network access. This is accomplished by pinging each host from the attacking machine, which is on a separate vSwitch guaranteeing the controller is involved, as well as connecting each host to a simple web server hosted on a separate machine. During testing, there were no failures in the attack machine being able to successfully launch an attacker due to the network. There are, of course, failures from the PHM mutating before a connection could be established, but this is not a stability issue as given enough time, the host could be exploited.

Effectiveness.

The effectiveness of the PHM framework is shown to be quite robust, as presented in Chapter 5. The results show how the PHM framework significantly increases the scan time and decreases the number of hosts found and the length an attacker can maintain access. The t-tests between the traditional network and the software-defined network for each configuration statistically prove there is a statistically-significant difference between the two networks, with very little likelihood the results are due to chance, per the low p-values for each test. The initial framework testing satisfies the third goal of effectiveness; hosts cannot be reached through their real IP addresses, only through the virtual address. This is accomplished by attempting to ping the real addresses of all hosts and failing to communicate with them.

Ease of Implementation.

The ease of implementation for this framework is demonstrated in two ways. First, the framework is run on an Ubuntu virtual machine on a typical enterprise server. Second, because the controller is a python script running on the Ryu framework [42], it can easily be moved from one machine to another with the Ryu framework and be launched with a single command.

6.3 Significance of Research

Contributions.

The main focus of this research is to extend the research conducted by UNC from a simulated network to a real network and validate the results. Random Host Mutation for a software-defined network was originally proposed by UNC researchers in their conference paper [42] and later migrated to a traditional network in [6], [7], and [34]. The original research was conducted entirely in Mininet [8]. While this platform is great for creating rapid prototypes or testing ideas, it simulates the hosts and switches instead of using actual hardware/software. This type of research provides a working proof of concept, but does not

allow for true statistical analysis to be conducted to verify empirical results. The research described herein took the initial idea of RHM on SDN and implemented it on a real network, with real hardware and real operating systems to statistically show the host mutation does indeed impede an adversary's attempts to scan and attack a network. To that end, this research metrics highly effective, showing significant increases in scan time and decreases in hosts found and penetration length.

Applications.

The application of PHM, or any flavor of host mutation, are numerous. The obvious application is the original goal of host mutation: prevent an attacker from successfully scanning a network. Throughout this research other benefits have emerged, such as increasing the time required to scan a network. This frustrates an attacker, since most scans are conducted at a very slow speed to avoid detection by a network IDS, and now the time required to scan a network has increased even more. Because the network is constantly changing, this only leaves the attacker two choices: either increase the scan rate and risk detection by the IDS, or stop scanning all together, which leaves him incomplete or inaccurate data to build attacks. Because PHM is a simple framework, it can be modified with a few characters to change the mutation rate as each user sees fit.

6.4 Future Work

The possibilities for future work expanding on this research are numerous. In addition to the continued research conducted at UNC [6], [7], [34], a few immediate strands of research come to mind.

First and foremost is extending the PHM framework to expand address mutation to include servers. Given that servers host critical network functionality and data, they are highly valuable targets for attackers. The limitation seen in this research is that some network functions require lengthy connections, such as FTP. If these connections were to be interrupted by the mutation, it would interfere with user connectivity, which is unacceptable in most enterprise networks. An addition to PHM to allow for extended connections, without

allowing for a new attack vector would significantly improve the functionality of PHM.

The second direction for future work would be to study PHM in a true network environment to measure the relationship between the mutation rate, scan rate, and the ability of an IDS to catch attackers. The experiments conducted for this research focused on isolating the hosts from network traffic to measure the impact of the framework on scan time and accuracy. Further research would be armed with this knowledge and could shift the focus to more real-world applications- practical settings to actually stop and/or catch attackers.

This research brought host mutation to life on real hardware. A third of research venue could make host mutation more realistic by introducing a human factor. Using the same framework, or an improved version, experiments could be conducted with a professional penetration team to determine system robustness. Not only would this provide invaluable insight into how the framework can succeed, it would also generate data for system improvements.

6.5 Chapter Summary

Moving Target Defense, and specifically host mutation, is a cutting-edge defense technique that focuses on defeating attackers before they ever have the chance to launch an attack. By stopping or catching network scans and further providing the rest with inaccurate data about the network, defenders can utilize Proactive Host Mutation as an additional layer in their defenses. This chapter provided the conclusions drawn from extensive testing of the PHM framework on actual hardware versus a simulation, it presented applications for PHM in the real world, and it described future work that could build upon this research.

Appendix A. Pilot Study Results

This appendix is an excerpt from the console of R-Studio after running the code for the pilot study. The variables followed by `c()`, is where there date from the study is entered into R. `sd` stands for standard deviation.

```
> ConfidenceInterval = 1.96
> MarginOfError = 5
>
> IntenseScan = c(16.07, 16.17, 20.17, 16.04, 15.96, 14.01, 15.09, 15.92, 15.94, 14.95,
                  27.17, 27.03, 22.49, 23.13, 19.83, 17.67, 18.92, 20.9, 19.9, 16.63, 17.5, 15.79,
                  15.97, 17.67, 15.85, 16.11, 15.89, 15.96, 17.58, 14.93)
>
> mean(IntenseScan)
[1] 17.908
> sd(IntenseScan)
[1] 3.36244
> (ConfidenceInterval*(sd(IntenseScan))^2)/MarginOfError
[1] 4.431953
>
> IntenseHosts = c(5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 1)
>
> mean(IntenseHosts)
[1] 4.866667
> sd(IHIntenseHosts)
[1] 0.7302967
> (ConfidenceInterval*(sd(IntenseHosts))^2)/MarginOfError
[1] 0.2090667
>
> QuickScan = c(10.57, 2.07, 2.07, 2.89, 3.67, 2.99, 3.67, 2.08, 2.07, 2.06, 2.07, 2.08,
```

```

2.06,2.06,2.08,2.06,2.07,2.09,2.07,2.07,2.08,2.09,2.07,2.06,2.09,2.06,
2.07,2.10,3.59,2.08)

>
> mean(QuickScan)
[1] 2.571333
> sd(QuickScan)
[1] 1.593763
> ConfidenceInterval*(sd(QuickScan)^2)/MarginOfError
[1] 0.9957117
+
+ QuickHosts = c(5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5)
>
> mean(QuickHosts)
[1] 5
> sd(QuickHosts)
[1] 0
> ConfidenceInterval*(sd(QuickHosts)^2)/MarginOfError
[1] 0
+
+ IntensePHMScan = c(119.18,110.12,117.55,146.5,41.02,156.13,190.91,113.83,
48.43,39.79,140.12,73.64,145.11,64.64,87.73,69.59,112.19,113.16,113.19,
155.79,111.14,113.15,195.49,111.43,96,96.53,113.71,76.69,89.61,113.33)
>
> mean(IntensePHMScan)
[1] 109.19
> sd(IMScan)
[1] 38.34552
> ConfidenceInterval*(sd(IntensePHMScan))^2/MarginOfError

```

```

[1] 576.3887
>
> IntensePHMHosts = c(4,3,3,3,1,4,2,3,2,1,3,4,3,3,4,4,1,2,5,3,1,3,3,1,4,4,
3,3,4,2)
>
> mean(IntensePHMHosts)
[1] 3.045
> sd(IntensePHMHosts)
[1] 0.3024127
> ConfidenceInterval*sd(IntensePHMHosts)^2/MarginOfError
[1] 0.03584975
>
> QuickPHMScan = c(12.01,7.32,12.66,9.13,7.34,11.86,15.58,10.44,8.86,13.97,
10.03,10.8,10.16,7.6,11.88,12.45,11.66,5.69,12.7,9.15,10.48,6,12.26,
6.43,8.64,11.54,5.01,6.83,8.56,12.67)
>
> mean(QuickPHMScan)
[1] 9.990333
> sd(QuickPHMScan)
[1] 2.663554
> (ConfidenceInterval*sd(QuickPHMScan)^2)/MarginOfError
[1] 2.781052
>
> QuickLengthExploit = c(3.15,3.14,3.16,3.11,3.04,3.15,3.13,3.16,3.15,3.13,
3.17,3.13,3.16,2.17,3.13,3.14,3.18,2.16,3.19,3.14,3.13,3.11,2.14,3.14,
3.18,3.14,3.15,3.17,3.15,3.15)
>
> mean(QuickLengthExploit)

```

```

[1] 3.045
> sd(QuickLengthExploit)
[1] 0.3024127
> ConfidenceInterval*sd(QuickLengthExploit)^2/MarginOfError
[1] 0.03584975
>
> QuickPHMHosts = c(5, 2, 1, 3, 2, 2, 1, 2, 3, 1, 2, 3, 4, 4, 1, 2, 3, 5, 3,
                    3, 3, 4, 1, 3, 3, 2, 3, 3, 3, 1)
>
> mean(QuickPHMHosts)
[1] 2.6
> sd(QuickPHMHosts)
[1] 1.132589
> ConfidenceInterval*sd(QuickPHMHosts)^2/MarginOfError
[1] 0.5028414

```

Appendix B. Experiment Results

The following tables display the results from all of the experiments run for this thesis. Each table is the data from one experiment. Each row of the table represents a single run for that experiment, capturing the time elapsed for both scans, the number of hosts found in each scan, and the how long the penetration lasted, if it began at all. Any tables with scan times of 0 with one entry at 0.000001, is used to represent instances where Nmap could not complete its scan and therefore did not produce any results. Any tables with Pen Times of 0 means Kali could not launch the attack fast enough before the network mutated.

2.1 5 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	16.070	10.570	5.000	5.000	3,600.000	3,600.000
2	16.170	2.070	5.000	5.000	3,600.000	3,600.000
3	20.170	2.070	5.000	5.000	3,600.000	3,600.000
4	16.040	2.890	5.000	5.000	3,600.000	3,600.000
5	15.960	3.670	5.000	5.000	3,600.000	3,600.000

Table 18. 5 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	119.180	12.010	4.000	5.000	0.000	3.150
2	110.120	7.320	3.000	2.000	0.000	3.140
3	117.550	12.660	3.000	1.000	0.000	3.160
4	146.500	9.130	3.000	3.000	0.000	3.110
5	41.020	7.340	1.000	2.000	$1 \cdot 10^{-6}$	3.040

Table 19. 5 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	49.330	11.240	4.000	2.000	3.150	3.150
2	43.930	9.140	3.000	4.000	3.150	3.150
3	45.660	7.270	1.000	3.000	3.150	3.220
4	46.860	9.630	3.000	3.000	3.160	3.140
5	47.740	6.000	4.000	5.000	2.160	3.140

Table 20. 5 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	44.420	6.770	3.000	2.000	7.130	7.210
2	45.220	9.720	3.000	2.000	8.200	7.230
3	40.230	7.260	1.000	4.000	7.160	7.160
4	49.490	10.160	3.000	1.000	6.220	7.170
5	47.850	10.790	2.000	1.000	7.140	7.150

Table 21. 5 Hosts - 5 Minutes

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	44.340	7.040	2.000	4.000	17.160	17.170
2	62.870	6.010	5.000	3.000	16.050	16.170
3	50.540	10.650	5.000	3.000	16.170	16.200
4	40.710	5.990	2.000	5.000	16.150	17.200
5	42.870	8.160	3.000	4.000	16.060	16.160

Table 22. 5 Hosts - 15 Minutes

2.2 10 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	26.110	4.620	10.000	10.000	3,600.000	3,600.000
2	24.960	4.820	10.000	10.000	3,600.000	3,600.000
3	19.100	4.200	10.000	10.000	3,600.000	3,600.000
4	20.640	5.280	10.000	10.000	3,600.000	3,600.000
5	20.110	3.780	10.000	10.000	3,600.000	3,600.000

Table 23. 10 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	105.050	12.170	6.000	1.000	0.000	3.150
2	149.260	9.400	9.000	5.000	0.000	3.150
3	133.180	10.170	6.000	5.000	0.000	3.130
4	161.290	6.290	10.000	7.000	0.000	2.160
5	141.040	5.210	10.000	9.000	$1 \cdot 10^{-6}$	3.130

Table 24. 10 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	60.100	8.730	9.000	4.000	0.000	2.140
2	53.110	8.290	6.000	6.000	0.440	3.170
3	53.570	10.970	6.000	3.000	0.350	3.150
4	51.520	5.640	5.000	6.000	1.140	3.170
5	48.440	9.590	6.000	8.000	1.060	3.140

Table 25. 10 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	57.100	7.700	7.000	10.000	6.150	7.150
2	67.650	11.400	8.000	5.000	6.160	7.220
3	60.970	8.010	8.000	6.000	6.170	7.140
4	57.150	5.180	7.000	7.000	6.150	7.160
5	44.290	11.080	3.000	5.000	7.150	6.150

Table 26. 10 Hosts - 5 Minutes

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	49.260	11.500	5.000	3.000	17.160	17.200
2	64.830	5.460	8.000	8.000	15.190	16.190
3	63.570	12.210	10.000	2.000	16.200	17.220
4	48.140	7.780	3.000	10.000	17.170	20.330
5	55.860	10.760	8.000	3.000	16.210	17.160

Table 27. 10 Hosts - 15 Minutes

2.3 20 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	40.230	6.220	20.000	20.000	3,600.000	3,600.000
2	36.920	5.130	20.000	20.000	3,600.000	3,600.000
3	38.720	5.420	20.000	20.000	3,600.000	3,600.000
4	38.080	5.230	20.000	20.000	3,600.000	3,600.000
5	37.660	6.520	20.000	20.000	3,600.000	3,600.000

Table 28. 20 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	164.880	10.250	11.000	9.000	0.000	2.160
2	217.440	9.190	16.000	12.000	0.000	3.070
3	242.850	14.000	17.000	17.000	0.000	2.130
4	283.950	6.620	17.000	9.000	0.000	3.130
5	165.670	7.510	11.000	16.000	$1 \cdot 10^{-6}$	3.140

Table 29. 20 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	226.020	12.530	19.000	5.000	0.000	3.150
2	230.280	7.840	16.000	17.000	0.000	2.160
3	156.500	5.750	12.000	15.000	0.000	2.160
4	218.670	11.610	17.000	4.000	0.000	3.130
5	256.490	9.450	18.000	8.000	$1 \cdot 10^{-6}$	3.140

Table 30. 20 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	62.070	6.550	10.000	16.000	6.140	7.140
2	91.030	8.420	13.000	11.000	6.160	6.150
3	86.430	10.440	10.000	2.000	6.190	7.150
4	113.460	6.980	17.000	16.000	4.150	7.160
5	140.820	7.160	16.000	13.000	5.150	7.170

Table 31. 20 Hosts - 5 Minutes

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	57.190	12.200	7.000	5.000	15.200	17.160
2	129.990	8.530	8.000	9.000	15.150	17.190
3	80.780	10.720	14.000	10.000	16.080	16.160
4	92.480	7.340	17.000	16.000	15.150	17.190
5	82.760	8.310	11.000	9.000	16.180	17.150

Table 32. 20 Hosts - 15 Minutes

2.4 30 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	93.840	4.680	30.000	30.000	3,600.000	3,600.000
2	85.600	5.830	30.000	30.000	3,600.000	3,600.000
3	89.880	5.860	30.000	30.000	3,600.000	3,600.000
4	81.380	4.980	30.000	30.000	3,600.000	3,600.000
5	69.760	5.850	30.000	30.000	3,600.000	3,600.000

Table 33. 30 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	0.000	9.740	0.000	20.000	0.000	3.160
2	0.000	15.040	0.000	23.000	0.000	0.000
3	0.000	9.460	0.000	21.000	0.000	3.150
4	0.000	8.800	0.000	15.000	0.000	2.150
5	$1 \cdot 10^{-6}$	8.490	$1 \cdot 10^{-6}$	26.000	$1 \cdot 10^{-6}$	3.170

Table 34. 30 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	449.310	8.490	30.000	28.000	0.000	3.060
2	68.760	10.610	10.000	17.000	0.000	3.150
3	160.780	11.830	13.000	6.000	0.000	3.170
4	155.620	11.560	11.000	6.000	0.000	3.150
5	289.660	11.540	19.000	9.000	$1 \cdot 10^{-6}$	3.140

Table 35. 30 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	154.050	8.670	23.000	27.000	5.150	6.160
2	89.600	11.250	15.000	13.000	6.160	7.170
3	89.150	8.190	15.000	28.000	6.140	7.160
4	74.900	19.110	10.000	11.000	5.160	7.150
5	90.800	9.980	14.000	9.000	5.140	7.170

Table 36. 30 Hosts - 5 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	111.770	11.600	15.000	9.000	16.290	6.160
2	155.550	9.700	28.000	21.000	15.180	16.290
3	78.130	8.210	9.000	24.000	16.170	16.160
4	192.470	9.380	26.000	16.000	14.140	17.170
5	154.070	10.160	26.000	8.000	15.160	17.220

Table 37. 30 Hosts - 15 Minute

2.5 40 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	126.680	1.710	40.000	40.000	3,600.000	3,600.000
2	101.910	2.730	40.000	40.000	3,600.000	3,600.000
3	84.550	3.050	40.000	40.000	3,600.000	3,600.000
4	79.870	2.270	40.000	40.000	3,600.000	3,600.000
5	74.450	2.300	40.000	40.000	3,600.000	3,600.000

Table 38. 40 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	0.000	8.920	0.000	21.000	0.000	3.150
2	0.000	10.430	0.000	18.000	0.000	3.140
3	0.000	11.390	0.000	14.000	0.000	3.180
4	0.000	11.620	0.000	11.000	0.000	2.190
5	$1 \cdot 10^{-6}$	9.440	$1 \cdot 10^{-6}$	31.000	$1 \cdot 10^{-6}$	3.140

Table 39. 40 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	546.800	10.590	33.000	22.000	0.000	2.140
2	394.400	11.140	23.000	13.000	0.000	2.140
3	318.290	8.140	23.000	29.000	0.000	3.140
4	262.970	11.660	18.000	8.000	0.000	3.140
5	417.850	9.830	25.000	14.000	$1 \cdot 10^{-6}$	3.140

Table 40. 40 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	128.170	13.840	20.000	37.000	5.150	7.160
2	160.490	11.690	26.000	29.000	5.180	6.160
3	118.930	15.350	17.000	19.000	5.140	7.170
4	93.910	10.320	13.000	22.000	6.170	7.140
5	112.690	12.350	17.000	22.000	5.170	7.180

Table 41. 40 Hosts - 5 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	179.710	12.230	25.000	23.000	13.170	17.160
2	129.080	11.030	18.000	14.000	15.160	16.160
3	177.790	13.620	26.000	24.000	14.190	16.170
4	173.490	11.080	29.000	17.000	14.170	17.170
5	132.110	10.870	21.000	17.000	15.170	16.190

Table 42. 40 Hosts - 15 Minute

2.6 50 Hosts

Experiment	Intense Scan	Quick Scan	Intense Hosts	Quick Hosts	Pen Time - I	Pen Time - Q
1	127.470	11.600	50.000	50.000	3,600.000	3,600.000
2	108.740	9.870	50.000	50.000	3,600.000	3,600.000
3	90.550	10.100	49.000	50.000	3,600.000	3,600.000
4	76.660	8.010	50.000	50.000	3,600.000	3,600.000
5	80.500	7.410	50.000	50.000	3,600.000	3,600.000

Table 43. 50 Hosts - Control

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	0.000	10.510	0.000	13.000	0.000	3.200
2	0.000	12.030	0.000	15.000	0.000	2.280
3	0.000	13.220	0.000	46.000	0.000	3.140
4	0.000	11.050	0.000	11.000	0.000	3.160
5	$1 \cdot 10^{-6}$	10.560	$1 \cdot 10^{-6}$	17.000	$1 \cdot 10^{-6}$	2.160

Table 44. 50 Hosts - 30 Second

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	398.540	20.160	24.000	24.000	0.000	2.480
2	0.000	27.380	0.000	31.000	0.000	3.130
3	0.000	12.020	0.000	28.000	0.000	3.170
4	479.040	10.340	31.000	38.000	0.000	3.150
5	0.000	12.680	0.000	28.000	$1 \cdot 10^{-6}$	3.150

Table 45. 50 Hosts - 1 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	150.830	18.010	24.000	35.000	5.160	7.170
2	152.860	12.480	19.000	40.000	4.140	7.150
3	302.860	10.690	39.000	19.000	0.000	6.160
4	265.870	12.540	39.000	34.000	3.140	6.170
5	86.290	10.740	14.000	21.000	16.150	6.150

Table 46. 50 Hosts - 5 Minute

Experiment	Scan - I	Scan - Q	Num Hosts - I	Num Hosts - Q	Pen Time - I	Pen Time - Q
1	156.580	11.400	19.000	9.000	15.210	17.130
2	269.060	10.850	42.000	34.000	13.180	16.370
3	158.260	12.010	20.000	18.000	15.160	17.140
4	144.480	9.770	21.000	23.000	15.140	16.160
5	72.040	14.740	9.000	14.000	15.190	16.160

Table 47. 50 Hosts - 15 Minute

Appendix C. R Code

```
library(ggplot2)
library(reshape2)
library(scales)

ggplot <- function(...) { ggplot2::ggplot(...) + theme_bw() }

# Import Data
hosts = data.frame(read.csv("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Results/Sheet 1-5 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-5 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-5 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-5 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-5 Hosts - 15 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-10 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-10 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-10 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-10 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-10 Hosts - 15 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
```

```

Sheet 1-20 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-20 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-20 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-20 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-20 Hosts - 15 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-30 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-30 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-30 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-30 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-30 Hosts - 15 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-40 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-40 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-40 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-40 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/

```

```

Sheet 1-40 Hosts - 15 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-50 Hosts - Control.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-50 Hosts - 30 Sec.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-50 Hosts - 1 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-50 Hosts - 5 Min.csv"),
read.csv("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Results/
Sheet 1-50 Hosts - 15 Min.csv"))

# Intense
intenseTest = data.frame(statistic = double(), parameter = double(),
p.value = double(), conf.int.low = double(),
conf.int.high = double(),
estimateX = double(), estimateY = double())

intenseSummary = data.frame(avgScanTime = double(),
avgCScanTime = double(),
sdScanTime = double(), sdCScanTime = double(),
avgHost = double(), avgCHost = double(),
sdHost = double(), sdCHost = double(),
avgPenTime = double(), avgCPenTime = double(),
sdPenTime = double(), sdCPenTime = double())

# Quick
quickTest = data.frame(statistic = double(), parameter = double(),

```

```

p.value = double(), conf.int.low = double(),
conf.int.high = double(),
estimateX = double(), estimateY = double())

quickSummary = data.frame(avgScanTime = double(),
avgCScanTime = double(),
sdScanTime = double(), sdCScanTime = double(),
avgHost = double(), avgCHost = double(),
sdHost = double(), sdCHost = double(),
avgPenTime = double(), avgCPenTime = double(),
sdPenTime = double(), sdCPenTime = double())

#### Data Analysis ####

# Changes based on different number of hosts
for(numHosts in 1:6) {
# Changes based on the number of experiments done at each host level
for(expNum in 1:28){
# Determines the index for the control
contNum = 35 * (numHosts - 1) + expNum %% 7

if(contNum == 0){
contNum = 7
}
if(contNum == 35){
contNum = 42
}
}

```

```

if (contNum == 70){
contNum = 77
}
if (contNum == 105){
contNum = 112
}
if (contNum == 140){
contNum = 147
}
if (contNum == 175){
contNum = 182
}
cont = hosts[contNum]

# Determines the index for the tested value
testNum = 35 * (numHosts - 1) + expNum + 7
test = hosts[testNum]

# Performs the T tests for all experiments
# if (expNum %% 7 > 1 && expNum %% 7 <= 5){

# Adds averages and sd for the summary values to lists
if (expNum %% 7 == 2){
tTest = t.test(test, cont, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[1],
tTest$estimate[2])

```

```

names(newTestRow) = names(intenseTest)
intenseTest = rbind(intenseTest, as.list(newTestRow))

intenseScanRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),
sd(cont[,1]))

} else if(expNum %% 7 == 3){
tTest = t.test(test, cont, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[1],
tTest$estimate[2])

names(newTestRow) = names(quickTest)
quickTest = rbind(quickTest, as.list(newTestRow))

quickScanRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),
sd(cont[,1]))

} else if(expNum %% 7 == 4){
tTest = t.test(cont, test, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[2],
tTest$estimate[1])

names(newTestRow) = names(intenseTest)
intenseTest = rbind(intenseTest, as.list(newTestRow))

intenseHostsRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),

```

```

sd(cont[,1]))
newIntenseRow = c(intenseScanRow, intenseHostsRow)

} else if(expNum %% 7 == 5){
tTest = t.test(cont, test, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[2],
tTest$estimate[1])

names(newTestRow) = names(quickTest)
quickTest = rbind(quickTest, as.list(newTestRow))

quickHostsRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),
sd(cont[,1]))
newQuickRow = c(quickScanRow, quickHostsRow)

} else if(expNum %% 7 == 6){
tTest = t.test(cont, test, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[2],
tTest$estimate[1])

names(newTestRow) = names(intenseTest)
intenseTest = rbind(intenseTest, as.list(newTestRow))

intensePenRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),
sd(cont[,1]))
newIntenseRow = c(newIntenseRow, intensePenRow)

```

```

names(newIntenseRow) = names(intenseSummary)
intenseSummary = rbind(intenseSummary, as.list(newIntenseRow))

} else if(expNum %% 7 == 0){
tTest = t.test(cont, test, conf.level = 0.90)
newTestRow = c(tTest$statistic, tTest$parameter, tTest$p.value,
tTest$conf.int[1], tTest$conf.int[2], tTest$estimate[2],
tTest$estimate[1])

names(newTestRow) = names(quickTest)
quickTest = rbind(quickTest, as.list(newTestRow))

quickPenRow = c(mean(test[,1]), mean(cont[,1]), sd(test[,1]),
sd(cont[,1]))
newQuickRow = c(newQuickRow, quickPenRow)
names(newQuickRow) = names(quickSummary)
quickSummary = rbind(quickSummary, as.list(newQuickRow))
}
} # else{
# if(expNum %% 7 == 6){
# intensePenRow = c(mean(test[,1]), sd(test[,1]))
# newRow = c(newIntenseRow, intensePenRow)
#
# names(newRow) = names(intenseSummary)
# intenseSummary = rbind(intenseSummary, as.list(newRow))
#
# } else if(expNum %% 7 == 0){
# quickPenRow = c(mean(test[,1]), sd(test[,1]))

```

```

#      newRow = c(newQuickRow, quickPenRow)
#
#      names(newRow) = names(quickSummary)
#      quickSummary = rbind(quickSummary, as.list(newRow))
#  }
# }
}

# Plot the data

mutationLevel = c("30S", "1M", "5M", "15M", "30S", "1M",
"5M", "15M", "30S", "1M", "5M", "15M",
"30S", "1M", "5M", "15M", "30S", "1M",
"5M", "15M", "30S", "1M", "5M", "15M")

Hosts = c("5 Hosts", "5 Hosts", "5 Hosts", "5 Hosts", "10 Hosts",
"10 Hosts", "10 Hosts", "10 Hosts", "20 Hosts", "20 Hosts",
"20 Hosts", "20 Hosts", "30 Hosts", "30 Hosts", "30 Hosts",
"30 Hosts", "40 Hosts", "40 Hosts", "40 Hosts", "40 Hosts",
"50 Hosts", "50 Hosts", "50 Hosts", "50 Hosts")

textSize = 45

# Intense Scan
# Hosts found
for(plotNum in 1:6){
lower = 4 * (plotNum - 1) + 1
upper = plotNum * 4

```

```

hostFound = data.frame(intenseSummary$avgHost[lower : upper],
intenseSummary$avgCHost[lower : upper],
mutationLevel[lower : upper])
names(hostFound) = c("SDN", "Control", "Mutation Level")

plotHost = melt(hostFound, variable.name = "Network")
plot = ggplot(plotHost, aes('Mutation Level', value, fill = Network)) +
geom_line(aes(colour=Network, group = Network), size = 2) +
geom_point(aes(colour=Network, group = Network), size = 4) +
scale_fill_brewer(palette = "Set1") +
coord_cartesian(ylim=c(0, 50)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Hosts Found") +
scale_x_discrete(limits = hostFound$'Mutation Level') +
theme(text = element_text(size = textSize),
axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntenseScanHost", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)

# Conf Int Plot
base = 12 * (plotNum - 1) + 2
rows = c(base, base + 3, base + 6, base + 9)
df = intenseTest[rows, ]
hostTimeCI = data.frame(abs(df$estimateX - df$estimateY),
df$conf.int.low,

```

```

df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(hostTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(hostTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = hostTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(-1, 55)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (S)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntenseHostConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Scan time
for(plotNum in 1:6){

# Scan Time Plot

```

```

lower = 4 * (plotNum - 1) + 1
upper = plotNum * 4

scanTime = data.frame(intenseSummary$avgScanTime[lower : upper],
intenseSummary$avgCScanTime[lower : upper],
mutationLevel[lower : upper])
names(scanTime) = c("SDN", "Control", "Mutation Level")

plotHost = melt(scanTime, variable.name = "Network")

plot = ggplot(plotHost, aes('Mutation Level', value, fill = Network)) +
geom_line(aes(colour=Network, group = Network), size = 2) +
geom_point(aes(colour=Network, group = Network), size = 4) +
scale_fill_brewer(palette = "Set1") +
coord_cartesian(ylim=c(-1, 400)) +
scale_y_continuous(breaks = pretty_breaks(n=10), name = "Scan Time (S)") +
scale_x_discrete(limits = scanTime$'Mutation Level') +
theme(text = element_text(size = textSize), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntenseScanTime", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)

# Conf Int Plot
base = 12 * (plotNum - 1) + 1
rows = c(base, base + 3, base + 6, base + 9)
df = intenseTest[rows, ]

```

```

scanTimeCI = data.frame(abs(df$estimateX - df$estimateY), df$conf.int.low,
df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(scanTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(scanTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = scanTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(-160, 400)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (S)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntenseScanConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Pen Time
for(plotNum in 1:6){
# Scan Time Plot

```

```

lower = 4 * (plotNum - 1) + 1
upper = plotNum * 4

penTime = data.frame(intenseSummary$avgPenTime[lower : upper],
  mutationLevel[lower : upper],
  Hosts[lower : upper])
names(penTime) = c("Pen Time", "Mutation Level", "Hosts")

plot = ggplot(penTime, aes(x = 'Mutation Level', y = 'Pen Time',
  group = 'Hosts')) +
  scale_x_discrete(limits = penTime$'Mutation Level') +
  geom_line(size = 2) +
  geom_point(size = 4) +
  coord_cartesian(ylim=c(0, 18)) +
  scale_y_continuous(breaks = pretty_breaks(n=10),
  name = "Average Penetration Length (M)") +
  theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntensePenTime", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)

# Conf Int Plot
base = 12 * (plotNum - 1) + 3
rows = c(base, base + 3, base + 6, base + 9)
df = intenseTest[rows, ]
penTimeCI = data.frame(abs(df$estimateX - df$estimateY), df$conf.int.low,

```

```

df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(penTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(penTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = penTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(3580, 3600)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (M)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/IntensePenConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Table of t-test results
write.csv(intenseTest, "/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Results/intenseTest.csv")

```

```

# Quick Scan
# Hosts found
for(plotNum in 1:6){
  lower = 4 * (plotNum - 1) + 1
  upper = plotNum * 4

  hostFound = data.frame(quickSummary$avgHost[lower : upper],
    quickSummary$avgCHost[lower : upper],
    mutationLevel[lower : upper])

  names(hostFound) = c("SDN", "Control", "Mutation Level")

  plotHost = melt(hostFound, variable.name = "Network")
  plot = ggplot(plotHost, aes('Mutation Level', value, fill = Network)) +
    geom_line(aes(colour=Network, group = Network), size = 2) +
    geom_point(aes(colour=Network, group = Network), size = 4) +
    scale_fill_brewer(palette = "Set1") +
    coord_cartesian(ylim=c(0, 50)) +
    scale_y_continuous(breaks = pretty_breaks(n=10), name = "Hosts Found") +
    scale_x_discrete(limits = hostFound$'Mutation Level') +
    theme(text = element_text(size = textSize), axis.title.y=element_text(vjust=1))

  ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/Aust-Thesis/Figures/
    QuickScanHost", toString(plotNum), ".png", sep=""), height = 12, width = 14)
  #print(plot)

# Conf Int Plot
base = 12 * (plotNum - 1) + 2

```

```

rows = c(base, base + 3, base + 6, base + 9)
df = quickTest[rows, ]
hostTimeCI = data.frame(abs(df$estimateX - df$estimateY), df$conf.int.low,
df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(hostTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(hostTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = hostTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(-1, 45)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (S)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/QuickHostConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Scan time

```

```

for(plotNum in 1:6){
lower = 4 * (plotNum - 1) + 1
upper = plotNum * 4

scanTime = data.frame(quickSummary$avgScanTime[lower : upper],
quickSummary$avgCScanTime[lower : upper],
mutationLevel[lower : upper])
names(scanTime) = c("SDN", "Control", "Mutation Level")

plotHost = melt(scanTime, variable.name = "Network")

plot = ggplot(plotHost, aes('Mutation Level', value, fill = Network)) +
geom_line(aes(colour=Network, group = Network), size = 2) +
geom_point(aes(colour=Network, group = Network), size = 4) +
scale_fill_brewer(palette = "Set1") +
coord_cartesian(ylim=c(-1, 16)) +
scale_y_continuous(breaks = pretty_breaks(n=10), name = "Scan Time (S)") +
scale_x_discrete(limits = scanTime$'Mutation Level') +
theme(text = element_text(size = textSize), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/QuickScanTime", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)

# Conf Int Plot
base = 12 * (plotNum - 1) + 1
rows = c(base, base + 3, base + 6, base + 9)

```

```

df = quickTest[rows, ]
scanTimeCI = data.frame(abs(df$estimateX - df$estimateY), df$conf.int.low,
df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(scanTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(scanTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = scanTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(-1, 15)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (S)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/QuickScanConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Pen Time
for(plotNum in 1:6){

```

```

# Scan Time Plot

lower = 4 * (plotNum - 1) + 1
upper = plotNum * 4

penTime = data.frame(quickSummary$avgPenTime[lower : upper],
  mutationLevel[lower : upper],
  Hosts[lower : upper])
names(penTime) = c("Pen Time", "Mutation Level", "Hosts")

plot = ggplot(penTime, aes(x = 'Mutation Level', y = 'Pen Time',
  group = 'Hosts')) +
  scale_x_discrete(limits = penTime$'Mutation Level') +
  geom_line(size = 2) +
  geom_point(size = 4) +
  coord_cartesian(ylim=c(0, 18)) +
  scale_y_continuous(breaks = pretty_breaks(n=10),
  name = "Average Penetration Length (M)") +
  theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/QuickPenTime", toString(plotNum), ".png", sep=""),
  height = 12, width = 14)
#print(plot)

# Conf Int Plot

base = 12 * (plotNum - 1) + 3
rows = c(base, base + 3, base + 6, base + 9)
df = quickTest[rows, ]

```

```

penTimeCI = data.frame(abs(df$estimateX - df$estimateY), df$conf.int.low,
df$conf.int.high, mutationLevel[lower : upper],
Hosts[lower : upper])
names(penTimeCI) = c("Difference", "Confidence Interval - Low",
"Confidence Interval - High", "Mutation Level", "Hosts")

plot = ggplot(penTimeCI, aes(x = 'Mutation Level', y = 'Difference',
group = 'Hosts')) +
geom_errorbar(aes(ymin = 'Confidence Interval - Low',
ymax = 'Confidence Interval - High'),
width = .5, size = 2) +
scale_x_discrete(limits = penTimeCI$'Mutation Level') +
geom_line(size = 2) +
geom_point(size = 4) +
coord_cartesian(ylim=c(3580, 3600)) +
scale_y_continuous(breaks = pretty_breaks(n=10),
name = "Mean Difference (M)") +
theme(text = element_text(size = 55), axis.title.y=element_text(vjust=1))

ggsave(paste("/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Figures/QuickPenConfInt", toString(plotNum), ".png", sep=""),
height = 12, width = 14)
#print(plot)
}

# Table of t-test results
write.csv(quickTest, "/Users/kooki3monst4r/Documents/AFIT/
Aust-Thesis/Results/quickTest.csv")

```

Appendix D. Scripts

```
## intense.rc

<ruby>
# Start Timer
startTime = Time.now

#Nmap Scan
run_single("db_nmap --min-hostgroup 96 -T4 -A -v -n 10.131.2.0/24")

endTime = Time.now

puts "Start Time: " + startTime.inspect + "End Time: " +
EndTime.inspect

# quick.rc

<ruby>
# Start Timer
startTime = Time.now

#Nmap Scan
run_single("db_nmap --min-hostgroup 96 -T4 -n -F 10.131.2.0/24")

endTime = Time.now

puts "Start Time: " + startTime.inspect + "End Time: " +
EndTime.inspect
```

```

# exploit.rc

<ruby>

# Start Timer

startTime = Time.now


run_single("use exploit/windows/smb/ms08_067_netapi")
run_single("set PAYLOAD windows/meterpreter/bind_tcp")
run_single("set LHOST 10.131.2.51")
run_single("set LPORT 10693")
run_single("set RPORT 445")
run_single("set RHOST #{framework.datastore['RHOST']}")
run_single("set SMBPIPE BROWSER")
run_single("exploit")
run_single("exit")

endTime = Time.now


puts "Start Time: " + startTime.inspect + "End Time: " +
endTime.inspect

```

Appendix E. Proactive Host Mutation - Controller

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import arp
from ryu.lib.packet import icmp
from ryu.lib.packet import ether_types
from threading import Timer
from random import randint

class Mutator(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Mutator, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.RIP_VIP = {}
        self.VIP_RIP = {}

    # Setup default for IP translation
    # self.RIP_VIP.setdefault(dpid, {})
    # self.VIP_RIP.setdefault(dpid, {})

    self.mutate()
```

```

t = Timer(900, self.mutate)
t.start()

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # install table-miss flow entry
    #
    # We specify NO BUFFER to max_len of the output action due to
    # OVS bug. At this moment, if we specify a lesser number, e.g.,
    # 128, OVS will send Packet-In with invalid buffer_id and
    # truncated packet data. In that case, we cannot output packets
    # correctly. The bug has been fixed in OVS v2.1.0.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
ofproto.OFPCMLNO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)

def add_flow(self, datapath, priority, match, actions,
    buffer_id=None):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    timeout = 900

```

```

inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
actions)]

if buffer_id:
mod = parser.OFPFlowMod(datapath=datapath, idle_timeout=timeout,
hard_timeout=timeout, buffer_id=buffer_id,
priority=priority, match=match,
instructions=inst)
else:
mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
match=match, instructions=inst)
datapath.send_msg(mod)

def mutate(self):
self.VIP_RIP = {}
self.RIP_VIP = {}
for address in range(1,12):
VIP = '10.131.2.'+str(randint(12,100))
while(self.VIP_used(VIP)):
VIP = '10.131.2.'+str(randint(11,100))
self.RIP_VIP['10.131.2.'+str(address)] = VIP
self.VIP_RIP[VIP] = '10.131.2.'+str(address)

self.logger.info(self.RIP_VIP)
self.logger.info(self.VIP_RIP)
def VIP_used(self, VIP):
if VIP in self.RIP_VIP or VIP in self.VIP_RIP:
return True
else:

```

```

return False

def address_translation(self, RIP, VIP):
    if RIP not in self.RIP_VIP:
        return False
    elif VIP not in self.VIP_RIP:
        return False
    else:
        return True

def packet_out(self, msg, ofproto, parser, datapath, in_port,
actions):
    data = None

    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

    out = parser.OFPPacketOut(datapath=datapath,
buffer_id=msg.buffer_id,
in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)

def arpTranslation(self, arpPkt, dpid, parser, out_port, ofproto,
msg, datapath, in_port):
    arpPkt = arpPkt[0]
    src_rip = arpPkt.src_ip
    dst_vip = arpPkt.dst_ip

```

```

# Catch if there doesn't exist a translation
if not self.address_translation(src_rip, dst_vip):
    return

src_vip = self.RIP_VIP[src_rip]
dst_rip = self.VIP_RIP[dst_vip]

self.logger.info('arp: ')
self.logger.info('src_RIP: %s, src_VIP: %s', src_rip, src_vip)
self.logger.info('dst_RIP: %s, dst_VIP: %s', dst_rip, dst_vip)

actions = [parser.OFPActionSetField(arp_tpa=dst_rip),
           parser.OFPActionSetField(arp_spa=src_vip),
           parser.OFPActionOutput(out_port)]

# install a flow to avoid the controller having to decide
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_type=0x806,
                           arp_tpa=dst_vip, arp_spa=src_rip)
    # verify if we have a valid buffer_id, if yes avoid to send both
    # flow_mod & packet_out
    if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
    return
else:
    self.add_flow(datapath, 1, match, actions)

self.packet_out(msg, ofproto, parser, datapath, in_port, actions)

```

```

def icmpTranslation(self, ipv4Pkt, dpid, parser, out_port,
    ofproto, msg, datapath, in_port):
    ipv4Pkt = ipv4Pkt[0]
    src_rip = ipv4Pkt.src
    dst_vip = ipv4Pkt.dst

    # Catch if there exists a translation
    if not self.address_translation(src_rip, dst_vip):
        return

    src_vip = self.RIP_VIP[src_rip]
    dst_rip = self.VIP_RIP[dst_vip]
    #
    # self.logger.info('ICMP: ')
    # self.logger.info('src_RIP: %s, src_VIP: %s', src_rip, src_vip)
    # self.logger.info('dst_RIP: %s, dst_VIP: %s', dst_rip, dst_vip)

    actions = [parser.OFPActionSetField(ipv4_dst=dst_rip),
        parser.OFPActionSetField(ipv4_src=src_vip),
        parser.OFPActionOutput(out_port)]

    # install a flow to avoid the controller having to decide
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_type=0x800,
            ipv4_dst=dst_vip, ipv4_src=src_rip)
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out

```

```

if msg.buffer_id != ofproto.OFP_NO_BUFFER:
    self.add_flow(datapath, 1, match, actions, msg.buffer_id)
    return
else:
    self.add_flow(datapath, 1, match, actions)
    self.packet_out(msg, ofproto, parser, datapath, in_port,
        actions)

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
            ev.msg.msg_len, ev.msg.total_len)

    ''' General Setup '''
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id

    # Setup the packet
    pkt = packet.Packet(msg.data)

    # Get the different protocols, if the exist

```

```

eth = pkt.get_protocols(ethernet.ethernet)[0] #
We know that there will be an ethernet component,
so we can get the first element
icmpPkt = pkt.get_protocols(icmp.icmp)
ipv4Pkt = pkt.get_protocols(ipv4.ipv4)
arpPkt = pkt.get_protocols(arp.arp)

# TODO: Create random virtualization (to include random ttl)
timeout = 30

'''Basic Switch Functionality'''
if eth.ethertype == ether_types.ETH_TYPE_LLDP:
# ignore lldp packet
return
dst = eth.dst
src = eth.src

'''Functionality for mac to port mappings'''
# Default behavior for mac to port mappings
self.mac_to_port.setdefault(dpid, {})

# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port

if dst in self.mac_to_port[dpid]:
out_port = self.mac_to_port[dpid][dst]
else:
out_port = ofproto.OFPP_FLOOD

```

```

'''Translation Rules'''
if arpPkt:
self.arpTranslation(arpPkt, dpid, parser, out_port, ofproto,
msg, datapath, in_port)
elif icmpPkt:
self.icmpTranslation(ipv4Pkt, dpid, parser, out_port, ofproto,
msg, datapath, in_port)
elif ipv4Pkt:
self.icmpTranslation(ipv4Pkt, dpid, parser, out_port, ofproto,
msg, datapath, in_port)
else:
actions = [parser.OFPActionOutput(out_port)]
# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
# verify if we have a valid buffer_id, if yes avoid to send both
# flow_mod & packet_out
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
self.add_flow(datapath, 1, match, actions, msg.buffer_id)
return
else:
self.add_flow(datapath, 1, match, actions)
self.packet_out(msg, ofproto, parser, datapath, in_port, actions)

```

Bibliography

1. Bruno Astuto A Nunes, Marc Mendonca, Xuan Nam Nguyen, Katia Obraczka, and Thierry Turetletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Commun. Surv. Tutorials*, 16(3):1617–1634, 2014.
2. Open Networking Foundation. Software-Defined Networking: The New Norm for Networks [white paper]. <http://www.opennetworking.org/>, 2012.
3. N McKeown, Tom Anderson, and H Balakrishnan. OpenFlow: Enabling Innovation in Campus Networks. *Acm Sigcomm*, 38(2):69–74, 2008.
4. Michael Todd. Dynamic Network Security Control Using Software Defined Networking. <https://www.afit.edu/docs/AFIT-ENG-MS-16-M-049.pdf>.
5. J Jafarian, E Al-Shaer, and Q Duan. Openflow Random Host Mutation: Transparent Moving Target Defense Using Software Defined Networking. *Proc. 1st Work. Hot Top. Softw. Defin. Networks (HotSDN 2012)*, pages 127–132, 2012.
6. Ehab Al-Shaer, Qi Duan, and Jafar Haadi Jafarian. Random Host Mutation for Moving Target Defense. *Lect. Notes Inst. Comput. Sci. Soc. Telecommun. Eng.*, 106 LNICS:310–327, 2013.
7. Jafar Haadi H. Jafarian, Ehab Al-Shaer, and Qi Duan. Spatio-temporal Address Mutation for Proactive Cyber Agility against Sophisticated Attackers. *Proc. First ACM Work. Mov. Target Def. - MTD '14*, pages 69–78, 2014.
8. Bob Lantz, Brandon Heller, and N McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. *Work. Hot Top. Networks*, pages 1–6, 2010.
9. Albert Greenberg, Gisli Hjalmtýsson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM Comput. Commun. Rev.*, 35(3):41, 2005.
10. Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The Case for Separating Routing from Routers. *Proc. ACM SIGCOMM Work. Futur. Dir. Netw. Archit. - FDNA '04*, page 5, 2004.
11. OpenFlow. Open Networking Foundation. Online. <http://www.opennetworking.org/>, 2016.
12. Martín Casado, Teemo Koponen, Scott Shenker, and Amin Tootoonchian. Fabric: A Retrospective on Evolving SDN. *Hot Top. Softw. Defin. Netw.*, pages 85–89, 2012.
13. Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: taking control of the enterprise. *Sigcomm '07*, pages 1–12, 2007.

14. Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. *15th USENIX Secur. Symp.*, pages 137–151, 2006.
15. Global Environment for Networking Innovations. <http://www.geni.net>, 2016.
16. Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN: An Intellectual History of Programmable Networks. *ACM Sigcomm Comput. Commun.*, 44(2):87–98, 2014.
17. Devolved Control of ATM Networks. <https://www.cl.cam.ac.uk/research/srg/netos/projects/archive/dcan/>.
18. R Enns. NETCONF configuration protocol. <https://tools.ietf.org/html/rfc6241>, 2011.
19. Alexander Shalimov, Dmitry Zuikov, Daria Zimarina, Vasily Pashkov, and Ruslan Smeliansky. Advanced Study of SDN/OpenFlow controllers. *Proc. 9th Cent. East. Eur. Softw. Eng. Conf. Russ. - CEE-SECR '13*, (October):1–6, 2013.
20. Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Network Configuration in a Box: Towards End-to-End Verification of Network Reachability and Security. *Proc. - Int. Conf. Netw. Protoc. ICNP*, pages 123–132, 2009.
21. Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. *Proceeding 16th ACM SIGPLAN Int. Conf. Funct. Program. - ICFP '11*, page 279, 2011.
22. E Skoudis and T Liston. *Counter Hack Reloaded: A Step-by-Step Guide to Computer Attacks and Effective Defenses*. Pearson Education, 2005.
23. Guilin Cai, Baosheng Wang, Xiaofeng Wang, and Yulei Yuan. An Introduction to Network Address Shuffling. pages 185–190, 2016.
24. D. Kewley, R. Fink, J. Lowry, and M. Dean. Dynamic Approaches to Thwart Adversary Intelligence Gathering. *Proc. DARPA Inf. Surviv. Conf. Expo. II. DISCEX'01*, 1(February):176–185, 2001.
25. Air Force and Information Directorate. Applications That Participate in their Own Defense (APOD). <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA415561>.
26. M. Atighetchi, P. Pal, F. Webber, and C. Jones. Adaptive Use of Network-Centric Mechanisms in Cyber-Defense. *Proc. - 2nd IEEE Int. Symp. Netw. Comput. Appl. NCA 2003*, pages 179–188, 2003.
27. Manolis Sifalakis, Stefan. Schmid, and David. Hutchison. Network Address Hopping: A Mechanism to Enhance Data Protection for Packet Communications. *IEEE Int. Conf. Commun. 2005. ICC 2005. 2005*, 3(May):1518–1523, 2005.

28. Levi Shi, Chunfu Jia, and Shuwang Lu. DoS Evading Mechanism upon Service Hopping. *Proc. - 2007 IFIP Int. Conf. Netw. Parallel Comput. Work. NPC 2007*, pages 119–122, 2007.
29. Chunlei Zhao, Chunfu Jia, and Kai Lin. Technique and Application of End-hopping in Network Defense. In *Proc. - 2010 1st ACIS Int. Symp. Cryptogr. Netw. Secur. Data Min. Knowl. Discov. E-Commerce Its Appl. Embed. Syst. CDEE 2010*, pages 266–270, 2011.
30. Matthew Dunlop, Stephen Groat, William Urbanski, Randy Marchany, and Joseph Tront. MT6D: A Moving Target IPv6 Defense. *Proc. - IEEE Mil. Commun. Conf. MILCOM*, pages 1321–1326, 2011.
31. S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending Against Hitlist Worms using Network Address Space Randomization. *Comput. Networks*, 51(12):3471–3490, 2007.
32. Justin Yackoski, Peng Xie, Harry Bullen, Jason Li, and Kun Sun. A Self-Shielding Dynamic Network Architecture. *Mil. Commun. Conf. 2011-Milcom 2011*, pages 1381–1386, 2011.
33. Ehab Al-Shaer Haadi Jafarian, Qi Duan. OF-RHM: Transparent Moving Target Defense using Software Defined Networking. *ACM SIGCOMM HotSDN Work.*, (August):127–132, 2012.
34. J H Jafarian, E Al-Shaer, and Qi Duan. Adversary-Aware IP address randomization for Proactive Agility Against Sophisticated Attackers. *Comput. Commun. (INFOCOM), 2015 IEEE Conf.*, pages 738–746, 2015.
35. Q Jia, K Sun, and A Stavrou. CellMov: Cellular-aided Moving Target Defense Against Internet Denial of Service Attacks. <http://www.cs.wm.edu/~ksun/publications/Motag-icccn2013.pdf>.
36. Andrew Clark, Kun Sun, and Radha Poovendran. Effectiveness of IP Address Randomization in Decoy-Based Moving Target Defense. *Proc. IEEE Conf. Decis. Control*, pages 678–685, 2013.
37. Douglas C Macfarland and Craig A Shue. The SDN Shuffle: Creating a Moving-Target Defense using Host-based Software-Defined Networking. *MTD’15 Oct. 12 2015, Denver, CO, USA*, 2015.
38. David Evans, a Nguyen-Tuong, and John Knight. Effectiveness of Moving Target Defenses. *Mov. Target Def. An Asymmetric Approach to Cyber Secur.*, pages 81–100, 2011.
39. Thomas E. Carroll, Michael Crouse, Errin W. Fulp, and Kenneth S. Berenhaut. Analysis of Network Address Shuffling as a Moving Target Defense. *2014 IEEE Int. Conf. Commun. ICC 2014*, pages 701–706, 2014.
40. Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

41. Nikolaj Bjørner and L. De Moura. Z310: Applications, Enablers, Challenges and Directions. *Sixth Int. Work. Constraints Form. Verif.*, pages 1–16, 2009.
42. RYU. <https://osrg.github.io/ryu/>.
43. POX. About POX — NOXRepo. <http://www.noxrepo.org/pox/about-pox/>, 2015.
44. Pica8. <http://www.pica8.com>.
45. Douglas C Montgomery. Design and Analysis of Experiments Eighth Edition. page 757, 2013.
46. RStudio. <https://www.rstudio.com>.

REPORT DOCUMENTATION PAGE					<i>Form Approved</i> OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 23-03-2017		2. REPORT TYPE Master's Thesis		3. DATES COVERED (From — To) Aug 2015 — Mar 2017		
4. TITLE AND SUBTITLE Proactive Host Mutation in Software-Defined Networking				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Aust, Matthew E., 2d Lt, USAF				5d. PROJECT NUMBER 17G139		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765				8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-17-M-003		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Intentionally Left Blank				10. SPONSOR/MONITOR'S ACRONYM(S)		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.						
14. ABSTRACT With Software-Defined Networking (SDN), hosts can be assigned a virtual IP address that changes at random intervals, allowing hosts to maintain their real IP Address, while presenting a moving target to network scanners. The original concept comes from the University of North Carolina. Their testing was performed in an SDN emulator without any statistical analysis. To further this field of research, a testbed with fifty hosts and one attack machine controlled by an SDN controller is established on physical servers. Experiments are conducted with different possible configurations (i.e., 5, 10, ... 50 hosts; and 30 sec, 1 min, 5 min, and 15 min mutation rates). Results show there is a statistically measurable difference between a traditional network and a software-defined network running host mutation software, with the same configuration. Comparing the scan times and number of hosts found from scans of both networks with a t-test, resulted in low P-Values, approximately 0.05 or lower. In addition, as the number of hosts increased from five to fifty, the difference between the number of hosts found in the traditional network and the SDN increased.						
15. SUBJECT TERMS Software-Defined Networking, Moving Target Defense, Host Mutation, Network Scanning Defense,						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins , AFIT/ENG	
U	U	U	U	135	19b. TELEPHONE NUMBER (include area code) (937) 255-3636, x7979; Barry.Mullins@afit.edu	