A Proactive Approach to Detect IoT Based Flooding Attacks by Using Software Defined Networks and Manufacturer Usage Descriptions

by

Laurence Chang

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2018 by the
Graduate Supervisory Committee:

Stephen Yau, Chair
Adam Doupé
Dijiang Huang

ARIZONA STATE UNIVERSITY

August 2018

ABSTRACT

The advent of the Internet of Things (IoT) and its increasing appearances in Small Office/Home Office (SOHO) networks pose a unique issue to the availability and health of the Internet at large. Many of these devices are shipped insecurely, with poor default user and password credentials and oftentimes the general consumer does not have the technical knowledge of how they may secure their devices and networks. The many vulnerabilities of the IoT coupled with the immense number of existing devices provide opportunities for malicious actors to compromise such devices and use them in large scale distributed denial of service attacks, preventing legitimate users from using services and degrading the health of the Internet in general.

This thesis presents an approach that leverages the benefits of an Internet Engineering Task Force (IETF) proposed standard named Manufacturer Usage Descriptions, that is used in conjunction with the concept of Software Defined Networks (SDN) in order to detect malicious traffic generated from IoT devices suspected of being utilized in coordinated flooding attacks. The approach then works towards the ability to detect these attacks at their sources through periodic monitoring of preemptively permitted flow rules and determining which of the flows within the permitted set are misbehaving by using an acceptable traffic range using Exponentially Weighted Moving Averages (EWMA).

DEDICATION

To my parents and sister for all the encouragement and support

To my dog Remy, youse a good boy

ACKNOWLEDGMENTS

Thank you to my adviser Dr. Stephen Yau for supporting me since my initial step into the realm of cyber security. I will be forever grateful for the opportunity to be a part of the CyberCorps Scholarship for Service and all the doors that have been opened since then. I want to extend my gratitude to my committee members Dr. Adam Doupé and Dr. Dijiang Huang. The classes I have taken with you have allowed me to build up my foundations necessary for this work.

A very special thank you to Mudumbai "Ranga" Ranganathan, Michael Williams, and Doug Montgomery, the opportunity you all provided to me at National Institute of Standards and Technology (NIST) would ultimately be the very foundation of which I was able to explore and produce a topic on. I will be forever grateful for being allowed to take part in the SDN MUD project.

Thank you to Yaozhong Song, another student in Dr. Yau's lab, for all the help you have given me whether they were with questions regarding methodologies, or possible directions I could explore. Thank you to Dr. Huang's student, Ankur Chowdhary, for helping me answer the questions I have had regarding networks, network security, and for giving me general advice especially during the early stages of refining my topic.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The Internet of Things, or IoT, is an emerging paradigm that is bringing innovation in how services are provided and delivered. While there exists no firm definition for the IoT, it is generally agreed upon that the core of the concept enables the network of resource constrained devices to provide services over the traditional Internet [1]. The emergence of Internet of Things (IoT) devices in SOHO (Small office/home office) networks has led to unprecedented new methods of delivering these services to consumers. These Internet connected devices such as smart light bulbs, IP cameras, DVRs, and even baby monitors to just name a few, are becoming commonplace objects in households across the world, contributing to the exponential growth of the network infrastructure [2]. While the IoT is increasingly finding its role in the provisioning of services and operations in these types of networks, one glaring issue remains and poses a serious threat to the health of the Internet overall, namely the inherent insecurity of the devices themselves. There are a number of factors that contribute to these insecurities. The first of these concerns the sheer number of devices that make up the IoT. The number of these types resource-constrained devices are ever growing, with companies like Gartner and Cisco estimating that by 2020, the number of IoT devices in existence may be 26 billion [3] and 50 billion respectively [4]. Intel, on the other hand, estimate this number to be as high as 200 billion [5]. Regardless of the more accurate value, the common factor is that the number of Internet connected, limited-purpose, devices are increasing far beyond their human counterparts, creating concerns in scalability and open attack surface. When also taking into account the

heterogeneous nature of the variety of devices that make up the IoT, it is easy to see that proper security configuration of the network is not a trivial task. In an industry environment, administrators would not only need to ensure that these devices are receiving access to their required services but also need to ensure that only legitimate and permitted communications may reach the device itself. With different devices speaking different protocols, speaking through different ports, and in need of access to different applications and hosts, this job is both time consuming and difficult. Bring this challenge into the small office and home network environment and it becomes apparent that if the consumer lacks the technical knowledge and skill to configure their own networks, they stand very little chance against malicious actors hoping to take over these devices for their own gain.

In the event that unauthorized network access is achieved by an adversary, limited-purpose IoT devices unfortunately do not make the task of hijacking them any more difficult. Due to their resource-constrained nature, these devices lack the capability and computing power to truly establish practical protection mechanisms. Furthermore, IoT devices suffer from the same security issues that most computing systems suffer from, namely those caused by human factors. Developers are bound at some point to make mistakes and this directly affects the security of devices that run these vulnerable applications. Vulnerabilities may go undiscovered for quite awhile and when they are discovered, it can take quite some time for patches to roll out, by which it may be too late. Despite these known threats, reports by HP and Gartner claim that by 2020, 60% of their predicted 26 billion devices predicted will still be insecure and highly susceptible to exploition [6].

The consequences of IoT insecurity are already present. Evidence for the utilization of so called IoT botnets have already been found in multiple incidents of

distributed denial-of-service attacks in the past couple years. The Mirai malware, arguably the most well-known IoT malware, has been largely responsible for these attacks. The Mirai botnet is known to have been comprised of IoT devices that include routers, printers, IP cameras, and webcameras, and has already been uitilized in several occasions [7]. In one such case on September 22, 2016, the security website *KrebsOnSecurity* owned by journalist Brian Krebs, was brought down for about four days in an attack that peaked around to 620 Gbps [8]. A few days before the Krebs attack, on September 19, 2016 the Mirai botnet was used against the French webhost and cloud service provider OVH, an attack peaking at around 1.1 Tbps [9]. Perhaps the most infamous use of Mirai occurred just a month later in October when the botnet was used against the Domain Name System (DNS) provider Dyn in an attack that reportedly peaked up to 1.2 Tbps, effectively bringing down the availability of websites such as Twitter, Netflix, Reddit, and Github for several hours [9][10].

These incidents have given insight to how severe IoT based DDoS attacks can be in a world encompassed by an ever increasing number of vulnerable IoT devices with no means of protecting themselves. Since the release/leak of Mirai's source code in October 2016, a variety of Mirai variants have emerged, each with different capabilities and demonstrating that the IoT is going to be utilized for malicious intentions as long as they stay insecure. The capability of generating attacks reaching sizes of terabits per seconds is no longer a difficult task due to the potential masses of IoT devices that are susceptible to hijacking, and finding a solution to detect and mitigate these attacks is crucial for the health and availability of the Internet.

This thesis seeks to provide an approach for the detection of these types of attacks by utilizing a proposed Internet Engineering Task Force (IETF) specification, Manufacturer Usage Descriptions [4][11]. Basing our work off those proposed at the

National Institute of Standards and Technology [12], we introduce a proof of concept that implements the policing properties of the Manufacturer Usage Description architecture in an Software Defined Network context in order to define the predetermined traffic flows that are associated with the limited-purpose IoT devices that would be supported by the MUD specification, which we define as a MUD enabled device, or MUD device. From this set of predetermined flows, we demonstrate that we can narrow down the possible victims that potentially compromised MUD devices can attack such that we only need to monitor the flows within the permitted set to detect those flows behaving abnormally and indicative of flooding attacks. Furthermore, we envision that this type of system will be deployed at the network edge and managed by a third party with a view of all the managed IoT networks (potentially an ISP-like entity) such that detection may be done as close to the source as possible, allowing for the stoppage of malicious traffic directly at the gateways that are used by the misbehaving MUD devices. Through use of the OpenFlow protocol, we are given the capability of collecting statistics from each flow within the set of permitted flows. We attempt to model the behaviors of each MUD flow in terms of a four-tuple of IP flow features computed during a predetermined time interval, using a weighted exponential moving average time-series to determine if the metrics computed for a flow during a given interval is within the bounds of the recorded normal behavior or if the flow is indicative of flooding attacks.

The main contributions of this work are:

- A simplified SDN implementation of the MUD controller, the policy decision point component of the MUD architecture.
- An approach to selecting flows of interest originating from MUD IoT devices by utilizing the assumptions provided by the Manufacturer Usage Descriptions

specification. We define these flows as those displaying the characteristics of volumetric and asymmetric features common in DDoS flooding attacks.

We begin this thesis by covering the background of the problem we seek to address (DDoS flooding attacks, particularly those generated by IoT sources), along with the technologies that make up our proposed proof of concept, in Chapter 2. We continue on in Chapter 3 by examining the related works for DDoS and IoT network security using SDN. In Chapter 4, we present the components of the system and the design of the testbed, which are described in detail. Finally, in Chapter 5, we layout our experimentation procedure and evaluate the results.

Chapter 2

BACKGROUND

2.1 Distributed Denial of Service Attacks and IoT Botnets

Denial of Service (DoS) attacks are network attacks that seek to compromise the availability of network resources such that the attempts by legitimate users to access them are denied. A Distributed Denial of Service is a type of DoS in which the attack traffic stem from many different sources, thus it is distributed in nature. A DDoS attack is carried out through several phases, identified by [13] as the *recruitment, exploitation,* and *infection* phases. The recruitment phase of a DDoS attack involves the search for vulnerable machines that may be added to the attacker's arsenal and used as an attack entity. This is commonly performed through the scanning of remote machines in order to find certain vulnerabilities the attacker can exploit. After finding such vulnerabilities, the attacker may enact the exploitation and infection phases by first breaking into the device by exploitation the known vulnerability, and then running attack code such that control of the device is gained by the attacker. The compromised devices are finally turned into agents, known as *bots*, that the attacker can control. The continuous recruitment of potential victims results in a larger *botnet* that the attacker can then further utilize to launch an attack on a victim or repeat the process of scanning for new victims.

Zargar *et al* [14] describe the two main current methods that are used to launch DDoS attacks on the Internet. The first method involves the attacker delivering malformed packets to the victim with the intent of either confusing a network protocol

or breaking a vulnerable application the victim may be running. The second and more common method involves the attacker attempting to exhaust either network or server resources by launching network/transport layer flooding attacks to achieve the former, and application layer flooding attacks to accomplish the latter.

2.1.1 Flooding Attacks

Flooding attacks can be categorized into those that target the network/transport layer or the application layer. Network and transport layer DDoS flooding attacks often launch TCP (Transmission Control Protocol), UDP (User Datagram Protocol), ICMP (Internet Control Message Protocol), and DNS (Domain Name System) protocol packets with a goal of exhausting a victim's network resources by way of four possible methods [14]. The first is by way of delivering large volumes of spoofed or non-spoofed packets that take up a network's bandwidth, disrupting legitimate connections from occurring. The second is through protocol exploitation flooding attacks in whch a feature of a protocol is exploited, thus consuming resources of the victim like in the case of a TCP SYN flood attack. The third method is through reflection based methods, where the attackers spoof the victim and send forged requests to a network of *reflectors*. In turn, a mass volume of responses that think the request was legitimate overwhelm the victim, exhausting the bandwidth available for legitimate users. Finally, the last way in which an attack can be carried out against the network/transport layer is by way of amplification-based attacks. In these types of attacks, the attacker exploits services that generate large responses in comparison to a smaller request and direct these to the victim, resulting in the victim receiving large volume of traffic.

Application layer attacks differ from network and transport layer attacks in that they seek to target the services of the victim, aiming for resources such as sockets, CPU, memory, disk/database bandwidth, and I/O bandwidth [14]. The detection of application layer attacks is not a trivial task and is often very difficult to do so because of the difficulty in distinguishing attack traffic from normal application layer traffic. For example, in HTTP (Hypertext Transfer Protocol) flooding attacks, standard HTTP protocol methods are used against the victim, such as GET and POST, without any need of sending malformed payloads or spoofing IP addresses. Most HTTP flooding attacks are carried out with bots that all simultaneously request a specified resource such that legitimate requests are not responded to due to the compromised server-side processing. The consequences of these types of attacks are that less bandwidth is consumed by attackers and are more stealthy in behavior as compared to their network and transport attack counterparts.

2.1.2 IoT Botnet

The IoT has increasingly become a concern for the security of modern computer networks, especially those IoT devices that are considered to be resource-constrained and limited-purpose. The IoT's odd mixture of features including its ubiquity, limited computational capabilities, sheer size, and the continuous open Internet connection of its devices, has led to the exposure of security holes that have been exploited by IoT malware. To get an idea of how IoT devices are recruited, exploited, and infected for use in a DDoS attack, we examine the most well-known IoT malware, Mirai, and how it operates as detailed by [2][9][15]. The Mirai botnet is composed of four major components that consist of the bots themselves, the command and control server,

the loader responsible for selecting the right platform executables to use for infection, and the report server which maintains a database of connected devices in the botnet. Mirai first initializes the recruitment step by scanning for random public IP addresses on TCP ports 23 or 2323, the ports used by the Telnet protocol. Upon finding a host, Mirai begins a brute-force search for common preconfigured credentials (such as admin:password) in order to break into the device itself. Upon gaining access to a command line, device information is delivered to the report server whom the command and control server may reach to check the statuses of potential victims and devices already integrated into the botnet. Upon determining the hosts susceptible to infection, a command is issued to the loader to login to the victim with the found credentials and downloads the executable that corresponds to the architecture of the device. After installation of the malware, the command and control server can now communicate directly with the bots to initiate an attack from a list of 11 available commands as shown in Table 2.1.

Table 2.1. Mirai attack types

Attack Name	**Type**
`ATK_VEC_UDP`	UDP Flood
`ATK_VEC_VSE`	Valve Source Engine query Flood (Specific to the Source game engine)
`ATK_VEC_DNS`	DNS Water Torture [16]
`ATK_VEC_SYN`	SYN Flood
`ATK_VEC_ACK`	ACK Flood
`ATK_VEC_STOMP`	ACK Flood to bypass mitigation devices
`ATK_VEC_GREIP`	GRE IP Flood
`ATK_VEC_GREETH`	GRE Ethernet Flood
`ATK_VEC_PROXY`	Proxy knockback connection
`ATK_VEC_UDP_PLAIN`	Plain UDP flood optimized for speed
`ATK_VEC_HTTP`	HTTP layer 7 flood

Source: Özçelik *et al* [6]

The release of Mirai's source code has surprisingly not led to more implementations of detection and defense mechanisms to address IoT threats. In fact, a 2016 Q4

Username/Password	Manufacturer	Link to supporting evidence
admin/123456	ACTi IP Camera	https://ipvm.com/reports/ip-cameras-default-passwords-directory
root/anko	ANKO Products DVR	http://www.cctvforum.com/viewtopic.php?f=3&t=44250
root/pass	Axis IP Camera, et. al	http://www.cleancss.com/router-default/Axis/0543-001
root/vizxv	Dahua Camera	http://www.cam-it.org/index.php?topic=5192.0
root/888888	Dahua DVR	http://www.cam-it.org/index.php?topic=5035.0
root/666666	Dahua DVR	http://www.cam-it.org/index.php?topic=5035.0
root/7ujMko0vizxv	Dahua IP Camera	http://www.cam-it.org/index.php?topic=9396.0
root/7ujMko0admin	Dahua IP Camera	http://www.cam-it.org/index.php?topic=9396.0
666666/666666	Dahua IP Camera	http://www.cleancss.com/router-default/Dahua/DH-IPC-HDW4300C
root/dreambox	Dreambox TV receiver	https://www.satellites.co.uk/forums/threads/reset-root-password-plugin.101146/
root/zlxx	EV ZLX Two-way Speaker?	?
root/juantech	Guangzhou Juan Optical	https://news.ycombinator.com/item?id=11114012
root/xc3511	H.264 - Chinese DVR	http://www.cctvforum.com/viewtopic.php?f=56&t=34930&start=15
root/hi3518	HiSilicon IP Camera	https://acassis.wordpress.com/2014/08/10/i-got-a-new-hi3518-ip-camera-modules/
root/klv123	HiSilicon IP Camera	https://gist.github.com/gabonator/74cdd6ab4f733ff047356198c781f27d
root/klv1234	HiSilicon IP Camera	https://gist.github.com/gabonator/74cdd6ab4f733ff047356198c781f27d
root/jvbzd	HiSilicon IP Camera	https://gist.github.com/gabonator/74cdd6ab4f733ff047356198c781f27d
root/admin	IPX-DDK Network Camera	http://www.ipxinc.com/products/cameras-and-video-servers/network-cameras/
root/system	IQinVision Cameras, et. al	https://ipvm.com/reports/ip-cameras-default-passwords-directory
admin/meinsm	Mobotix Network Camera	http://www.forum.use-ip.co.uk/threads/mobotix-default-password.76/
root/54321	Packet8 VOIP Phone, et. al	http://webcache.googleusercontent.com/search?q=cache:W1phozQZURUJ:community.freepbx.org/t/packet8-atas-phones/411
root/00000000	Panasonic Printer	https://www.experts-exchange.com/questions/26194395/Default-User-Password-for-Panasonic-DP-C405-Web-Interface.html
root/realtek	RealTek Routers	
admin/1111111	Samsung IP Camera	https://ipvm.com/reports/ip-cameras-default-passwords-directory
root/xmhdipc	Shenzhen Anran Security Camera	https://www.amazon.com/MegaPixel-Wireless-Network-Surveillance-Camera/product-reviews/B00EB6FNDI
admin/smcadmin	SMC Routers	http://www.cleancss.com/router-default/SMC/ROUTER
root/ikwb	Toshiba Network Camera	http://faq.surveillixdvrsupport.com/index.php?action=artikel&cat=4&id=8&artlang=en
ubnt/ubnt	Ubiquiti AirOS Router	http://setuprouter.com/router/ubiquiti/airos-airgrid-m5hp/login.htm
supervisor/supervisor	VideoIQ	https://ipvm.com/reports/ip-cameras-default-passwords-directory
root/<none>	Vivotek IP Camera	https://ipvm.com/reports/ip-cameras-default-passwords-directory
admin/1111	Xerox printers, et. al	https://atyourservice.blogs.xerox.com/2012/08/28/logging-in-as-system-administrator-on-your-xerox-printer/
root/Zte521	ZTE Router	http://www.ironbugs.com/2016/02/hack-and-patch-your-zte-f660-routers.html

Figure 2.1. List of suspected devices susceptible to Mirai
Source: *Krebs on Security* [7]

DDoS Threat report by the security company *Nexusguard* has found that the number of bot instances have actually doubled from 213,000 to 493,000 since the emergence of Mirai variants, two months after code release [9]. A variety of other IoT malware have also been identified since then including the first IoT botnet written in Lua aptly named *Luabot*, Hajime, and Brickerbot. Variants of Mirai continue to emerge as late as January 2018 with the identified Mirai variant *Mirai Okiru* that targets devices running on the Argonaut RISC Core (ARC) processor [17]. More than 2 billion products are shipped with the ARC CPU including cameras, mobile devices, utility meters, televisions, flash drives, and automotives, giving insight as to just how large and diverse an attack may be comprised of.

2.2 Software Defined Networks

Software Defined Networking (SDN) is a networking paradigm that seeks to address the various issues that are associated with traditional IP networks such as the difficulty of managing increasingly complex networks and the hindering of network innovation, due to the "closed box" solutions of proprietary networking elements. SDN serves as a promising solution for such issues due to its motivation of separating the control logic of the network from the network elements that forward traffic based on the decisions made at the control plane. This decoupling of the control and data plane results in abstractions of the underlying network infrastructure that provides an operator with one of the fundamental pillars of the SDN concept, the programmability of the network. There are three abstractions that define an SDN; forwarding, distribution, and specification [18]. In the forwarding abstraction, details of the underlying networking hardware are hidden away such that an operator may be able to shape forwarding behavior as desired without needing to worry about individual device states. The distribution abstraction on the other hand is responsible for the dynamic management of the forwarding devices as well as the collection of network status. These efforts result in a global and centralized view of the network that masks the distributed nature underneath. Finally, the specification abstraction details the notion of allowing a network application to enable a desired modification of the network behavior without requiring direct interaction with the data plane itself.

These three abstractions can be achieved through the implementation of a network operating system (NOS), also known as an SDN controller. The SDN controller is a logically centralized entity that acts as the "brains" of the network and has direct control of the data plane via what is known as the Southbound APIs. This interface

grants the controller the capability of dynamically configuring the switches or routers with the forwarding decisions made separately at the control plane. An interface is also provided for application developers via the Northbound APIs, enabling the communication between network applications and the SDN controller. These applications run on top of the controller and are able to program the underlying data plane through the controller using high level instruction sets or programming languages that are translated into the instruction sets used by the Southbound API.

SDN has emerged as a premier candidate for DDoS defense solutions. Yan *et al* [19] describe five major features of SDN that make it such a promising tool to defeat DDoS attacks.

1. *Separation of control and data plane:*

   Through the separation of control and data plane, network security ideas can be swiftly implemented without concern for understanding a network element's "black box" configurations. These abstractions that SDN provide enable rapid innovation and deployment of large scale experiments that are configurable and easy to take down or build up.

2. *Centralized Logic and View of the Network:*

   The centralized capabilities of the SDN controller allow for the monitoring of all traffic patterns within a given network. This allows the controller to quarantine hosts that are suspected of misbehaving as well as continuously identify security threats that have not been present previously.

3. *Programmability:*

   SDN gives applications such as Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), the capability of programming the network based on its determination of the data collected from the controller. Furthermore, the

current state of the network itself may be provided to northbound applications to further determine how a future state might look.

4. *Software-based Traffic Analysis:*

   Due to the controller being a software-based entity, innovation to how traffic is collected and processed provides an edge over those traditionally implemented on a switch. Software based tools such as databases or IDS/IPS can be utilized to assist in the process of analyzing or detecting traffic of interest. Machine learning algorithms have also been an area of interest to use in conjunction with SDN.

5. *Dynamic Updating of Forwarding Rules:*

   Mitigation can be provided by SDN through the installation of rules that drop traffic deemed malicious by a northbound application, preventing unwanted traffic from propagating into the network. This also provides for an opportunity to reactively install forwarding rules that pipe flows of interest towards entities that are capable of performing Deep Packet Inspection, or towards machine learning applications that have the capability of generating new policies for previously unencountered attacks.

These features and characteristics of SDN are what brings promise in answering the two major obstacles in regards to IoT security; heterogeneity and scalability [20]. SDN provides for the possibility of being able to provide crypto, network, and traffic based security solutions to a variety of devices by way of platform independent SDN applications. Scalability of the network can be addressed through centralized handling of security configurations that are deployed uniformly throughout the network, without needing to worry about potential conflicts that may occur if security policies were implemented in a distributed fashion to individual network devices, such as in

traditional networks. It is envisioned that in the IoT environment, the controller will play a major role in providing these kinds of security solutions for those resource-constrained devices who may not be able to perform security measures themselves.

2.2.1 OpenFlow

The OpenFlow protocol is an example of a Southbound API and is currently the most popular solution for SDN implementation. The OpenFlow architecture consists primarily of two elements, the SDN controller, which is responsible for managing the collection of forwarding devices that make up the data plane, and the OpenFlow enabled forwarding devices themselves. Each OpenFlow switch consists of a channel that enables communication between controller and switch [21]. An OpenFlow switch will consist of one or more flow tables which the controller may configure by adding new flow entries to the table, deleting existing flow entries, or just modifying them. Flow entries are composed of *match* fields that are to be matched by received packets, such as Ethernet addresses, IP addresses, source and destination port numbers, and switch port numbers, to name a few. Upon a match or packet hit, the specified flow action for that match may be applied to the packet allowing for actions like packet forwarding, packet dropping, or sending the packet up to the controller for further processing and decision making. If there exists no flow entry for a packet to match against, it may be sent up to the controller, which then decides to either drop it or create a new flow rule such that any future packets that match the flow will be handled automatically by the switch without controller intervention.

Flow entries also consist of counters that are able to provide the statistics of a given flow. The *received packets* counter gives the number of packets that have

matched the given flow while the *received bytes* counter gives the number of bytes that have matched the given flow. These counters can be polled by the controller in order to monitor the traffic of a given switch. Applications running on top of the controller may also poll for this data in order to provide a more complex service such as QoS.

2.3 Manufacturer Usage Descriptions

Manufacturer Usage Descriptions (MUD) is a proposed Internet Engineering Task Force specification written by Elliot Lear, Ralph Droms, and Dan Romascanu, that describes a component based architecture that provides a means for limited-purpose IoT devices to relay its intended access controls and network needs to a given network. Traditionally, hosts on the Internet have primarily consisted of general purpose computing devices such as desktops and laptops (and in more recent years, smartphones and tablets) that are capable of speaking various types of protocols, utilized for a multitude of different tasks, and equipped with adequate amounts of resources such that they are capable of performing security measures on board. However, when we view the modern Internet in the scope of the IoT, many of these qualities are oftentimes unavailable in IoT devices. Very often are smart Things designed to be limited-purposed type devices, meaning they have been built by the manufacturers with specific intentions and functionality in mind.

As a simple example, let us examine what the most likely intentions of an IP camera should be. Given that the device is a camera, it is likely that the primary function of this device is to survey and record a given environment, such as for home security, and stream the captured frames to a a destination host such as a

local or remote server. The camera most likely will not be able to speak and process routing protocols or allow remote access to itself other than perhaps through a service in the form of a designated cloud service or mobile application. The camera most likely should not have the capabilities of reaching other hosts on the Internet such as Facebook, Google, or Netflix.

MUD can be realized through leveraging the manufacturer's role in describing the intent and communication patterns of their products, such as those described in the example. In the draft, the authors mention that the term *manufacturers* is used loosely to include any entity (organization or local administrator) with a notion of an intended communication pattern for a MUD enabled IoT device. In this thesis, we work on the premise that in general, consumers who buy these MUD enabled devices would likely be content with the default network access recommendations by the manufacturers and delegate the role of implementing these access controls to a more capable entity. A controller would then be responsible to process these policies such that the configurations may be deployed to the network elements which would enforce the described access controls. By securing IoT devices using this whitelist approach, several benefits become apparent.

- The surface of attack for a given device is reduced, the device is consequentially limited to only the intended functions and actions it may perform if compromised
- Network configurations for an ever increasing number of heterogeneous devices becomes more manageable and scalable due to pre-determined polices. We can expect that even devices that are distributed around the world, as long as they are of the same model and from the same manufacturers, they should adhere to the same access controls specified by the manufacturers

- An additional layer of protection is provided to address existing vulnerabilities on devices in a more timely manner than it may take for manufacturer patches to roll out, an especially important point for those devices that are no longer supported

The MUD architecture has a couple limitations to note. The first and foremost is that MUD was conceived with the purpose of addressing network authorization and security issues that IoT networks face. Given a general purpose computing devices such as a desktop, it becomes very difficult and meaningless to define the communication patterns it may have. Unlike a smart camera, a computer may be used for various different types of tasks. This implies that a general purpose computer will also most likely need to speak various types of protocols whereas the camera may only need to speak one or two. In this thesis, we utilize MUD as intended, to provide an additional layer of security to an IoT network comprised of limited-purpose smart devices. Another limitation imposed on MUD is that it currently lacks the capability of examining application layer payloads. As such, MUD is not a "end-all" solution to prevent attacks on an IoT network, but rather serves as an additional component and layer to protect against vulnerabilities in place of patching until they are identified and their fixes can be deployed.

The author's of the MUD proposal state that the MUD architecture is primarily used to address the threats *to* the devices rather than address the issue of the devices as a threat themselves. However, they note that depending on how certain processes are enacted and communicated, there may be offers of some protections against attacking devices. It is apparent that MUD is inherently useful for defending against attacks *on* IoT devices, however, we dedicate the majority of our discussions and this work towards investigating how the MUD architecture can be leveraged to protect

the Internet infrastructure *from* malicious devices. We detail this work more in depth during our discussion on design and implementation of the system. In the following sub-sections, we describe the components that make up the MUD architecture along the order of operations as they are defined in [11].

2.3.1 Components of the MUD Architecture

IoT Devices

In the context of the Manufacturers Usage Descriptions, the *Thing* or IoT device is a resource-constrained and limited-purpose device that is produced by the manufacturer with a specific intent and communication pattern in mind. The Thing is what MUD seeks to secure by reducing the surface of attack through the deployment of policies that define the communication patterns of the device. For the remainder of this thesis, when we refer to the terms *Thing, IoT device* or *device*, we mean them in the context of MUD where the IoT devices are limited-purpose, resource-constrained, and would ideally have an associated MUD policy. This is also synonymous with a term we coin, MUD enabled device, to define a limited-purpose IoT device with a associated MUD policy.

MUD Controller/MUD Manager

The MUD controller is the Policy Decision Point (PDP) of the system. It is responsible for receiving the URL describing the location of where to retrieve a MUD device's communication policy, known as the MUD file, and then retrieving the

actual file itself. Upon retrieval of the MUD file, the MUD controller is expected to validate and parse the file such that abstractions specified in the file (such as certain policy classes) may be resolved and mapped. The device's access controls and configurations are then deployed onto the network elements, including routers or switches. These abstractions and their mappings are then maintained and updated by the controller as needed.

MUD Universal Resource Locator (MUD URL)

The MUD URL is a Universal Resource Locator (URL) that describes where the MUD file (policy) for a MUD device may be retrieved. It is emitted by the IoT device and forwarded to the MUD controller through one of three described methods.

- **DHCP Option**: In this method, the client uses the Dynamic Host Configuration Protocol (DHCP) with the IANA reserved DHCP option 161 that is used to indicate to the DHCP server that it has a MUD policy that can be retrieved at the given URL. The DHCP server may then either process the URL itself if it is capable, or forward it to another entity such as the MUD controller for MUD file retrieval. For the purposes of this thesis, we use this option to implement the functionality and relaying the MUD URL to better portray and demonstrate the process.
- **X.509 Constraint**: The MUD URL can also be communicated to the controller through a certificate based approach by using a defined X.509 non-critical extension containing a single MUD URL to locate the MUD file for the device. Signature checking is performed by validating the device's manufacturer certificate chain and is successful if the certification chain can be validated AND

if the subject field of the device certificate is equal to the subject field of the certificate used to sign the MUD file itself.

- **LLDP Option**: This method leverages the Link Layer Discovery Protocol in order to advertise the identity, capabilities and neighbors of the Things. The main intent of this method is to be able to uniformly identify the types of MUD devices to the network in a standard fashion, without the need to discern devices of different vendors, as LLDP is a vendor-neutral protocol.

The MUD URL must use the HTTPS scheme to ensure that the identity of the manufacturer's MUD file server, which hosts the MUD files, can be verified. This also assists in assuring the MUD file's integrity itself.

Any URL that follows the HTTPS scheme can be a MUD URL. Several examples of what a MUD URL may look like are provided in the proposed specification [11].

- https://things.example.org/product_abc123/v5
- https://www.example.net/mudfiles/temperature_sensor/
- https://example.com/lightbulbs/colour/v1

To better ensure the legitimacy of the file that the URL points to, the URL may be tested against web or domain reputation services after signature validation of the MUD file.

MUD File Server

The MUD File server is simply a web server that hosts a MUD file. These are envisioned to be hosted by the Manufacturers of the Things.

MUD File

The MUD file serves to describe the associated IoT device, as well as its intended communication patterns and access controls. The file itself is a JSON file that is a serialization of a data model instance written in YANG (Yet Another Next Generation), a data modeling language often used to define data that is sent through the network to configure and retrieve the states of network elements. The MUD YANG model itself consists of the use of three defined YANG modules.

- `ietf-access-control-list`: A YANG model defined in [22] that defines a access control list model. MUD uses this module to describe the permitted communication patterns in both directions that are associated with a Thing. Each access control rule is defined as an access entry. When access entries of a device are defined in MUD, one can assume that the only features implemented would be those access controls that match against IPv4, IPv6, TCP, UDP, and ICMP rules as they are defined in [22].
- `ietf-mud`: The `ietf-mud` model itself is structured into three parts. The first component describing metadata relevant to the MUD file itself such as its MUD version, retrieval, validity, last updates, policy names, and model names among others. The second part of this model adds some augmentations to the previously mentioned `ietf-access-control-list` model that describe classes of policies relevant to the use of MUD URLs that may be used within a local environment. The set of classes are meant to abstract away IP addresses that can later be instantiated into actual addresses through local configuration. We summarize the definitions of these classes as they are presented by the authors of the Manufacturer Usage Description draft in Table 2.2. Finally, the

third component of the `ietf-mud` model augments the TCP match container in the `ietf-access-control-list` model so that one may be able to define and match on the direction of initiation for a TCP connection.

- **`ietf-acldns`**: The last model used by the MUD file is the `ietf-acldns`, an extension that augments the `ietf-access-control-list` model to allow the ability of referencing domain names. Both IPv4 and IPv6 matches are augmented with this extension. The data nodes defined in this module include the specification of a source DNS name for inbound flows, and a destination DNS name for outbound flows.

Table 2.2. Classes of MUD policies defined in the `ietf-mud` YANG module

`manufacturer` Class	A class of devices specified by the authority component of the device's MUD URL.
`same-manufacturer` Class	A class of devices identified by the same authority component of their MUD URLs.
`controller` Class	A class of devices that are identified by a specified controller's URI. The controller's URI is expected to be registered with the MUD controller such that it may maintain and update the mappings as needed.
`my-controller` Class	A class of devices that are mapped by a given MUD URL as specified by the MUD controller.
`local` Class	Class of IP addresses scoped within a specified administrative boundary, such as the local subnet.

We do not make extensive use of these classes for the purposes of our thesis, but understand their importance and the scenario that if devices were wrongfully admitted into any of these classes, it could potentially lead to the compromising of the other devices within the class. It is because of this security consideration that we are interested in exploring the ability to detect the devices suspected as being used as attack vectors, despite already being within the MUD specified access controls. Upon the MUD file's retrieval, the MUD controller must be able to validate the signature corresponding to it, which is signed by the MUD device's manufacturer. The MUD controller can retrieve the associated signature by examining the `mud-signature`

field, which specifies the URI resolving to the signature. This serves as a measure to prevent malicious tampering of the file and ensuring the file's integrity.

Listing 2.1 is an example of what a MUD file can look like. This particular example describes two policies that use IPv6, one for outbound traffic from the device and one for inbound traffic to the device from a cloud server. The outbound access control entry defines communication that is initiated by the device using the TCP protocol with the destination port 443 and destination host `service.bms.example.com`. The inbound policy defines communication from the server in response to the initiated communication from the client from source port 443, source host `service.bms.example.com`, and using the TCP protocol.

```
{
   "ietf-mud:mud":{
      "mud-version":1,
      "mud-url":"https://lighting.example.com/lightbulb2000"
         ,
      "last-update":"2018-03-02T11:20:51+01:00",
      "cache-validity":48,
      "is-supported":true,
      "systeminfo":"The BMS Example Lightbulb",
      "from-device-policy":{
         "access-lists":{
            "access-list":[
               {
                  "name":"mud-76100-v6fr"
               }
```

```
            ]
         }
      },
      "to-device-policy":{
         "access-lists":{
            "access-list":[
               {
                  "name":"mud-76100-v6to"
               }
            ]
         }
      }
   },
   "ietf-access-control-list:access-lists":{
      "acl":[
         {
            "name":"mud-76100-v6to",
            "type":"ipv6-acl-type",
            "aces":{
               "ace":[
                  {
                     "name":"cl0-todev",
                     "matches":{
                        "ipv6":{
```

```
                        "ietf-acldns:src-dnsname":"test.
                           com",
                        "protocol":6
                     },
                     "tcp":{
                        "ietf-mud:direction-initiated":"
                           from-device",
                        "source-port":{
                           "operator":"eq",
                           "port":443
                        }
                     }
                  },
                  "actions":{
                     "forwarding":"accept"
                  }
               }
            ]
         }
      },
      {
         "name":"mud-76100-v6fr",
         "type":"ipv6-acl-type",
         "aces":{
            "ace":[
```

```
                {
                    "name":"cl0-frdev",
                    "matches":{
                        "ipv6":{
                            "ietf-acldns:dst-dnsname":"test.
                                com",
                            "protocol":6
                        },
                        "tcp":{
                            "ietf-mud:direction-initiated":"
                                from-device",
                            "destination-port":{
                                "operator":"eq",
                                "port":443
                            }
                        }
                    },
                    "actions":{
                        "forwarding":"accept"
                    }
                }
            ]
        }
    }
]
```

```
85      }
86  }
```

Listing 2.1. Example of a MUD file describing permitted HTTPS communications to and from the device at `service.bms.example.com`

(*Source: Lear, Droms, and Romascanu [11]*)

Network Access Device

The Network Access Device is a router or switch that serves as the first hop onto the local network and as such, serves as the means to forward the MUD URL emitted by the Thing to the MUD controller. It is also the device that is to be configured by the MUD controller based on the MUD policies associated with the Thing, allowing only the type of traffic defined in the MUD file. For our approach, we use the software switch Open vSwitch (OvS) as our NAD in order to build the system in an SDN context. Using OvS allows for the SDN controller to configure the switch by way of flow rules that are then pushed onto the switch's flow tables. These flow rules then serve as whitelists that the traffic must match before it is forwarded to the correct port or device. If there exists no MUD flow that a flow matches with, then an alert is generated and the suspected traffic is forwarded to the controller for further action.

2.3.2 Order of Operations

In general, the order of operations of a MUD implementation will align with the following flow of events.

1. The Thing emits MUD URL
2. The MUD URL is forwarded to the MUD controller using one of the three methods described previously
3. The MUD controller retrieves the MUD file and its associated signature file as described in the file itself. Validation of the policy file is expected to be performed at this step

4. The MUD controller processes the MUD file and resolves any abstractions in the file (DNS names, *same-manufacturer* classes, controller access, etc.)
5. The MUD controller deploys the generated configurations onto the network elements that the Thing has connected to

We use this as a template to define the order of operations that our proof of concept SDN MUD implementation will follow. This process is described more in depth in Chapter 4.

Chapter 3

RELATED WORKS

The use of SDN as part of DDoS defense have been explored in many works, with several of them specific to the IoT landscape. Yan *et al* [19] surveyed various works that used SDN to defend against DDoS attacks and describe the three categories in which SDN DDoS solutions may be classified. We divide the first three sections of this chapter into the general SDN defense approaches (not necessarily pertaining to the IoT), categorized as source-based, then network-based, and finally destination-based. We then reserve a section to discuss the current works in SDN DDoS defense specifically in the IoT scope.

3.1 Source-Based SDN Approaches

To address the increasing number of network devices connected to the Internet through SOHO networks, Feamster [23] proposed a concept in which home network security is outsourced to a third party entity that would be more capable in securing these networks that normally would be managed poorly, if they were managed at all. To accomplish this, it is posed that management should be delegated to an entity with a broad network view and allows for the networks managed by this entity to operate in a "plug and play" fashion, relieving the burden of management from from the end users. The central controller as it were would have access to programmable gateways that reside in the home networks from which statistics, network state, and network activity can be retrieved and processed. Based on the decisions made at the

central controller, it may program the network at large through installation of new forwarding rules, filtering rules, or access controls configurations.

Work proposed by Medhi *et al* [24] carry on the approach of deploying network security solutions at the source of traffic by enabling standardized programmablity through SDN. In their work, it is argued that threat detection is best placed in the home network where algorithms can run at line rates and traffic rates are low. Meanwhile, the delegation of security policies into downstream networks is also achieved through communication of controller and programmable home gateways. To demonstrate their hypothesis, they implement four anomaly detection algorithms faithfully in the SDN context using the NOX controller and OpenFlow protocol. The algorithms that were implemented were *Threshold Random Walk with Credit Based Rate Limiting, Rate Limiting, Maximum Entropy Detector,* and *NETAD* (Network Anomaly Detection), with all four following the main idea of only installing flow rules whenever a connection attempt succeeds. Their results show that they were able to detect anomalies with high true positive and low false positive rates at the hpoity scoopome network as compared to the ISP level. They primarily attribute this to the difficulty of running anomaly detection algorithms that must service thousands of home gateways. While it should be noted that when concerned with general networks, this observation holds true, we observe that in IoT networks if we were to limit the flows to only those that are permitted beforehand, the detection of recruit and exploitation phases (such as occurrence of port scans) of DDoS can simply be handled by the filters. Thus we argue that placing security at the ISP level becomes a more viable option, where configurations are completely handled at the ISP level such that end users can inact the "plug and play" desirability as mentioned in [23].

3.2 Network-Based SDN Approaches

Network-Based SDN approaches can often be implemented as SDN applications and are generally composed of functional modules that perform flow collection, feature extraction, anomaly detection, and a means of attack mitigation [19].

A lightweight approach is proposed by Braga *et al* [25] that leverages the global visibility that SDN provides in order to monitor the switches within a SDN and classify the traffic they handle as either benign or malicious using a artificial neural network running on the controller. Three modules are described and implemented in their work. The first is the Flow Collector, used to periodically request flow entries from all flow tables of all registered switches. This is then fed into the *Feature Extractor* module, which computes and produces a 6-tuple of features, each associated with the corresponding switch that statistics were pulled from, that are to be used by the *Classifier* (anomaly detector) that analyzes whether the given 6-tuple corresponds to a DDoS flooding attack or legitimate traffic. Mitigation is performed after an attack alert from the classifier if malicious flows are identified. Classification is done through the use of Self Organizing Maps (SOM), an unsupervised artificial neural network that transforms n-dimensional data into 1 or 2 dimensional map or grid. The 6 features that make up the 6-tuple (Average of Packets per flow, Average of Bytes per flow, Average of Duration per flow, Percentage of Pair-flows, Growth of Single-flows, and Growth of Different Ports) are those commonly present during a DDoS occurrence. It was demonstrated that aggregating features of interest from the flows yielded low overhead compared to traditional approaches of preprocessing packets. The work was also able to show high detection rates on switches within the SDN while at the same time keeping false positives low.

The aggregation of features through flow statistics is again utilized in a approach by Yang *et al* [26] where they demonstrate the detection of DDoS occurrence by measuring the presence of volumetric and asymmetric features that are common during in DDoS attacks. They argue that the overhead and bottleneck of southbound communication during flow polling increases if networks become large and therefore the solution is to implement lightweight detection on the switch, or data plane itself through the implementation of a Field-programmable gate array (FPGA) based OpenFlow switch. This lightweight mechanism works in cooperation with a module which they dub the Fine-grained Detection Interface that lies on the control plane to perform controller-based detection methods such as those involved with learning and classifications. Detection is done through a lightweight flow monitoring algorithm that aggregates flow metrics from each flow and generates a 4-tuple (*Byte Count per Second, Packet Count per Second, Byte Count Asymmetry, Packet Count Asymmetry*) every t_n seconds. The algorithm then predicts the four metrics for the flow at time t_{n+1} by computing a weighted moving average (WMA) on a list of the n most recent vectors collected for that particular flow. The predicted output is then compared with the actual output via a ratio function. To account for the deviation between history records and the actual vector at time t_{n+1}, two thresholds are defined by taking the ideal average of collected vectors and then adding and subtracting 3 times the standard deviation to obtain the upper limit and lower limit respectively. These two values are also used as parameters to generate the upper and lower ratio limits. Abnormal flows are detected when all four actual ratio metrics in the 4-tuple fall out of range of the upper and lower limits. If any of these metrics fall within the computed thresholds, then it is determined to be acceptable.

3.3 Destination-based SDN Approaches

Lim *et al* [27] demonstrate a SDN based blocking scheme for botnet-based DDoS attacks, specifically those that target the application layer. In their system architecture, the victim server resides in an SDN controlled network and install flow rules in for communication in both directions whenever a connection is made with the victim. An application running on top of the controller is responsible for continuously monitoring the number of traffic flows on the switches while at the same time keeping a pool of available IP addresses used to "move" the protected service in the scenario of an attack. The server itself monitors metrics for indications of possible DDoS attacks. In the event that the server has determined an attack is occurring, the SDN application is notified and a new IP address is provided for the server to which the server relocates to. The server then sends an HTTP message that is explicitly NOT a HTTP 3xx message is then sent to prospective clients seeking to reach the service that a relocation has occurred and gives information on how to reach it before closing all the existing connections. This message is protected by a CAPTCHA in order to prevent bots from decoding this message. In the meantime, the controller installs instructions that allow flows towards the server with the new IP address. Clients that continue to attempt connection with the old address are deemed bots after a certain threshold, at which point their associated flow rules are simply modified by the controller to be dropped.

3.4 SDN Defense Mechanisms Specific to the IoT

After the wake of the Mirai attacks in 2016, several works with a focus towards addressing IoT based DDoS have emerged. Many of these approaches build on top of those explored in the previous section, but a shared characteristic among the proposals surveyed demonstrate a trend of moving security features such as monitoring, inspection, analysis, and mitigation services, closer to the source of the traffic.

Three of these works ([6], [28], [29]) have incorporated elements of the emerging edge computing paradigm in this transition effort to enable real-time computationally intensive security applications right at the network edge. The use of edge computing serves two primary purposes in these works. The first is to address the scaling problems that the IoT inherently generates due to the sheer volume of connected devices. With edge computing, SDN capabilities can be placed at the network edge rather than the core of networks, significantly reducing data volume that are inspected and mitigated by providers such as those in the cloud. The second is to extend and push cloud capabilities such as computational and storage resources towards the network edge to further maximize the capabilities of learning applications near the sources of traffic rather than in the network core.

Özçelik *et al* [6] use this concept with SDN to create a "closer-to-the-edge" defense architecture that they call *Edge-Centric Software-Defined IoT Defense*. They illustrate a topology in which the SDN controller resides on a edge computing node as an extension of the cloud. Each of these edge computing nodes may service one or more IoT networks, thus providing defense towards the endpoints of the network. They utilize the combination of two algorithms in order to detect the occurrence of recruitment and exploitation phases of DDoS, these two algorithms being Threshold

Random Walk with Credit Based Limiting (TRW-CB) and Rate Limiting (RL). The TRW-CB algorithm leverages the observation that a benign host is more likely to successfully complete a connection and thus the hosts that repeatedly fail connection attempts are increasingly suspected to be more likely to be an infected node. To accomplish this, the system maintains a queue of TCP SYN messages for each host and track the connection state of each one. Every time a three way handshake goes uncompleted or a timeout occurs, the message is dequeued and the likelihood of infection is increased for that particular node. The RL algorithm works off the assumption that a malicious host will be more likely to attempt a large number of connections than compared to a benign host. Legitimate connections made are also more likely to be to the same destinations previously made before. In this algorithm, two sets are maintained, one containing a list of recently contacted hosts and another that contains the list of attempted connections called the delay queue. When a new connection is attempted and the destination has not been recorded in the working set, the forwarding of this traffic is delayed and the connection is sent to the delay queue. The connection is allowed to proceed after d seconds and removed from the delay queue. If the connection succeeds, it is then moved to the working set. However, if the delay queue grows beyond its predetermined size n, an alarm is raised, indicating the presence of a malicious host on the network.

Aggarwal and Srivastava [28] propose two security models that utilize SDN and edge computing, one of which addresses the scenario in which IoT devices connect to the Internet via network devices such as routers or switches. In this scenario, the model described consists of three major components; OpenFlow enabled routers and switches representing the data plane, a *IoT Gateway Controller* that is placed at the network edge between IoT networks and ISP, and the *Main SDN Controller*

that resides at the ISP. The IoT Gateway Controller (GC) is the main component that provides the security measures for its designated zone comprised of the connected IoT networks. Due to its location of placement, it is capable of receiving all inbound and outbound traffic, a feature beneficial for the deployment of applications such as Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and Deep Packet Inspection (DPI). In the scheme proposed, traffic received by the GC is analyzed using DPI techniques or run through an IDS such as Snort in order to determine whether the traffic is benign or malicious. In the case that a threat is detected, the GC is capable of modifying the flow tables in the respective switches residing in the culprit IoT network such that the traffic matching a particular flow is dropped or re-routed. The Main SDN Controller on the other hand is responsible for the management of each GC continuously collecting data such as traffic throughput and bandwidth management. Packets that are cleared by a GC are forwarded to the Main SDN Controller which may apply further actions such as transmitting requests to the appropriate DNS servers or routing the packet to the next hop of its destination.

Bull *et al* [29] continue the trend of arguing for the use of SDN gateways in playing the security role for the connected IoT device by performing services such as packet inspection and traffic analysis. Similar to the previous works discussed, it is argued that moving security functionality to the network edge is critical for performance requirements, more rapid detection and mitigation at the traffic source, and load reduction between controller and switch communications. A method in which the traffic patterns of IoT devices are monitored and analyzed is used by the authors n order to determine whether they are perpetrators acting in an attack or victims on the receiving end of an attack. A statistics manager component is used to collect and

store data received on the SDN gateway in order to take into account historical flow features during analysis. Upon detection of an anomalous flow, three possible actions are defined:

- Blocking a flow by adding a flow rule to the Blocked Flows table and applying the DELETE OpenFlow action to the suspected flow
- Forwarding a flow to an isolated zone of the network for deeper inspection of traffic to generate a more robust decision in regards how to handle the device
- Applying QoS to limit impact of an attack if decision is not clear or in the even that blocking a single source is not possible

3.5 Points of Consideration

We find that while the previous works surveyed are invaluable building blocks for the future works in DDoS defense, several characteristics related specifically to IoT networks are not sufficiently addressed and what may work well in a traditional computing environment does not necessarily translate to success over in the IoT realm. When concerning attacks stemming from IoT sources, we believe methods such as the one described in Section 3.3 are not appropriate. Destination-based approaches delegate intensive defense tasks such as detection to the victim which must work in keeping up with incoming traffic. Given the large number of devices that an IoT botnet may consist of, this certainly results in overhead and creates a bottleneck due the requirement of processing a large volume traffic data. Consequently, a mechanism may resort to packet sampling but as shown by works in [24], this results in a large number of false positives as well, further creating conflict when legitimate hosts attempt to connect with a service. Network-based methods described in 3.2 serve to

be a better alternative towards detecting attacks from IoT sources, as traffic monitoring used in conjunction with learning or statistical methods can determine whether a particular flow is misbehaving or not. These methods however may find difficulty scaling as the size of networks increases along with the volume of data stemming form IoT sources that must be processed. This potentially leads to communication overhead between controller and switch communication during polling of statistics, leading to potential denial of service on the SDN controller itself. To alleviate this overhead, the work by [26] demonstrate a more lightweight approach by placing detection mechanisms on a custom prototype OpenFlow switch, such that abnormal traffic may be detected directly on the data plane before alerting applications on the control plane for deeper analysis. However, we disagree with the use of detection mechanisms on the data plane of the network. The reason is that we feel this approach is contradictory to the fundamental concepts of Software Defined Networks which states that logic and control should be delegated to the control plane and the data plane serves simply as the mechanisms of forwarding the traffic based on decisions from the control plane. To address scalability and overhead issues, we believe that utilizing a proactive strategy in which explicit access control rules in the form of flow rules are enforced, we can minimize the communications between switch and controller to only those that require controller intervention. In doing so, the detection mechanisms that were originally implemented on the dataplane can be re-delegated back to the control plane and handled by the software controller. We discuss more on the issue of potential vulnerabilities in SDN, and how we address them in our implementation in Chapter 4.

As explored by works in [6], [28], and [29], the emergence of extending intelligence to the edge of networks is proceeding with the requirements of reducing device

and controller communication overhead in mind, furthering the argument that traffic monitoring is still best placed at the control plane level where the global view of the network allows for detection of abnormal traffic within not just a single switch but all that reside in a controller's designated SDN. There are concerns with source-based approaches being a difficult domain to enact security due to the multitude of sources that need to be detected and filtered correctly [14]. However we argue that with the use of global policies describing the intent of each individual device as specified in MUD, a standardized method of determining filters and communication patterns of every IoT devices is achievable, making the determination of whether a device is misbehaving or acting accordingly to expected patterns more practical.

Chapter 4

APPROACH DESCRIPTION AND PROCEDURE

In this chapter we describe the design and implementation of our proposed approach to detect suspected attacking IoT devices using SDN and Manufacturer Usage Descriptions. With our approach, we seek to identify those devices that are emitting traffic which display the volumetric and asymmetric characteristics commonly present in flooding attacks. We achieve this by leveraging Software Defined Networks to implement the MUD controller where the central controller is capable of translating the MUD file for a particular device into OpenFlow rules. The controller can then programmatically configure the managed networks by pushing the generated MUD flow rules to the flow tables of the network switches the IoT device to inform how the switches should handle the permitted traffic. These sets of MUD rules essentially act as whitelists, in which traffic received and emitted from the IoT device must be matched with. In the case where there are no flow rules that the traffic form a device matches with, the packet will be forwarded to the controller for further action (such as deep packet inspection), due to the implication of unpermitted behaviors. With the established set of expected communication patterns in the form of MUD flow rules, we begin to continuously monitor each outbound MUD flow through another SDN application, and build a statistical model around the averages of four flow features which we define in Subsection 4.1.2.1. The computed averages at time t_n are then used to define a range of acceptable values which we expect to observe from the captured flow at time t_{n+1}. The core of our methodology can be summarized by two main ideas:

1. The generation of a set of permitted MUD flow rules for the connected IoT devices that were translated from the access control rules specified by the device's MUD file. MUD flows are defined terms of the OpenFlow Match fields (< *Protocol, SrcIP, SrcPort, DestIP, DestPort* >), which traffic to and from the device must match with.
2. Monitoring the flow statistics from the set of MUD flows and extracting their measurements at each interval. We use these metrics to compute the expected or "average" behaviors of an IoT device in terms of four IP flow features. Features that deviate greatly from a determined threshold based on the weighted averages are considered "flows of interest" and will be handled by the controller through either modifying a MUD rule to drop the traffic, or to forward it to another entity for further actions such as Deep Packet Inspection.

While the MUD architecture inherently makes hijacking devices and launching attacks more difficult for the adversary, we want to be able to take into account the possibility that MUD devices in the network may be compromised through some other channel.This may be possible if a MUD device is, for some reason, capable of lying about what it is and gains additional network access by being wrongfully admitted into a class of network accessibility, these classes being one of those that are described in Chapter 2 in regards to the MUD architecture. Let us posit the case in which a device is inappropriately admitted into a class of other IoT devices for one reason or another (such as perhaps the *my-controller* or *same-manufacturer* class), wherein the IoT devices already in this class also have access control rules stating they may speak to an external Internet host. Then it is possible that because this rogue device is allowed to communicate with all devices within this class, potentially having the capability of also taking them over and use them to launch attacks. It is

in this scenario in which we would like to maintain the ability to determine which MUD flow within the set of preinstalled flows are misbehaving.

We describe a flow as "misbehaving" if they adhere to the flow rules as they are defined, but exhibiting of behavior such as the emitting of large volumes of bytes, packets, or excessive communications that were not apparent in previously recorded measurements. As an example, consider a rule that specifies that a IoT device may only speak using the TCP protocol towards a host `cloud.example.com` that is listening on port `5555`. This rule can still be abided by an adversary but abused through the deployment of a TCP SYN attack that targets port `5555`. Our goal is thus to use the method described above to identify and filter the malicious flows within the set that we have allowed.

We divide the remainder of this chapter into three sections to describe the methodology and procedures of our implementation, as well as how it addresses the concerns stated above. In Section 4.1, we describe the two primary modules that make up the system along with their functionalities. In Section 4.2 we describe how the SDN environment was set up for testing and experimentation. Finally, in Section 4.3, we describe the security concerns and vulnerabilities in regards to SDN, particularly the centralized nature of the controller, and detail how inherent access control functionality of MUD alleviates them to the best of the system's abilities.

4.1 Implemented SDN Modules

There are two primary modules that make up the proposed methodology. The first is a simplified implementation of the MUD controller component, the policy decision point of the MUD architecture, whose functionality we implement as an SDN

application. The MUD controller is responsible for retrieving, translating, and deploying the access control policies to the managed network devices. The second module is another SDN application that functions as the flow monitor which periodically polls for flow counters from each permitted flow that have been preemptively installed on the OpenFlow switches.

4.1.1 MUD Controller

The MUD controller is an SDN application that can be viewed as the entity that is responsible for programming the network through the installation of flow rules derived from the MUD files. The moment an IoT device is connected to a network and its MUD URL is received by the controller, the MUD controller will extract the URL and retrieve the resource at the specified URL, which would be the MUD file that is written in a JSON format. The MUD controller is then responsible for parsing the file appropriately, translating the access controls described by the MUD file into flow rules. As previously mentioned, the MUD file is a JSON file generated from a YANG data model. As such, parsing a MUD file ideally would be done through the use of a YANG validator in order to ensure that all the elements in the file are in accordance those defined in the `ietf-mud`, `ietf-access-control-list`, and `ietf-acldns` YANG models. For the purposes of this thesis, we choose to just parse the JSON directly and assume that the elements and file itself are all valid. We base this assumption off the fact that all MUD files used for experimentation were generated through https://mudmaker.org/alpha/, a web application created by the authors of the MUD draft that can be used for generating example MUD files .

Parsing of the MUD file begins upon the MUD controller receiving the resource

located at the specified URL. Two lists are generated; one for containing the access control entries specifying *to-device*, or inbound flows, and another for the associated *from-device*, or outbound flows. The parser then scans through the file and for every access control entry in the MUD file, a vector V_{ace} is generated containing the rules specifying how a flow should look like. V_{ace} is defined as follows.

Definition 4.1.1. $V_{ace} =< DNSname,\ Protocol,\ SrcPort,\ DstPort >$

The values in the vector are then used by the MUD controller to construct OpenFlow rules from the access control policies described by the MUD file by specifying the OpenFlow Match fields that the packet headers from the associated MUD device must match in order to continue to be forwarded. These rules are pushed by the controller to the flow table on the OpenFlow switch the device is connected to.

Definition 4.1.2. *Inbound Flows:* (*Protocol, DstHost, SrcHost, SrcPort, DstPort*)

Definition 4.1.3. *Outbound Flows:* (*Protocol, SrcHost, DstHost, SrcPort, DstPort*)

Access control entries that correspond to inbound traffic are resolved first. Therefore, the *DstHost* field of the inbound flow should be that of the IoT device, which we represent as the MUD device's MAC address. Meanwhile, the *SrcHost* field of the flow is set to the associated IP address of a hostname representing the entity seeking to make a connection with the device. For purposes of this thesis, we assume that the host associated with the *DNSname* element of V_{rule} is always an Internet host or cloud service that is presumably in control by the manufacturer. The IP of the host is resolved by the MUD controller through DNS resolution of the *DNSname*, allowing us to specify the Match source field as the IP address of the host. The next item to be extracted from V_{rule} is the value of *Protocol* which can be one of three values; *any, TCP,* or *UDP*. If the value of *Protocol* is *any* this signifies that the manufacturer has

allowed either TCP or UDP protocol to be used in a flow. If *TCP* is specified, then the flow is expected to use the TCP protocol. The same logic applies if the value is equal to *UDP*. Finally, the source and destination port fields of the flow rule are set equal to those extracted from V_{rule}. The result is a list that contains the permitted inbound flows for a IoT device.

The same process is performed to generate the list of permitted outbound flows. The only difference in this procedure is that source and destination fields are flipped, with the *DstHost* flow field being set to the IP address resolved from *DNSname*, and the *SrcHost* field being set equal to the MAC address of the device. To the best of our knowledge, recent IoT malware, such as Mirai, do not have the feature of spoofing hardware addresses. Thus this implementation serves as a simple measure to ensure that any communications coming out of a specific device will be matched against the MUD flow(s) that has been installed in association with it. What this implies is even with IP spoofing, the increase in flow metrics will still be captured as we do not match by IP, but by hardware address. Any divergence from the expected behaviors of a device, such as attempting to connect to a host that is not specified, will find no flows that it can match against in the switch's flow table, raising an alert to the controller where further analysis and decisions can be made for the suspected device. To demonstrate more easily the flow of how the MUD architecture works, we opt to use the DHCP option that was specified in Section 2.3.1, and as such we assume that the MAC address of a device is indeed associated with that device. It should be noted however, that there are several security considerations if an implementation should choose to use this scenario (and the LLDP case), such as the possibility that a device may lie to the controller about what it is and gain unauthorized or unintended network access. It is acknowledged in a real world scenario that the implementation

should support the option for the X.509 certificate method to authenticate the device and preserve integrity of the MUD file.

For every outbound flow, a 64-bit value is generated and set as the flow's cookie, a value assigned by the controller to a flow for identification purposes. The inbound flow that is correspondent to the ID'd outbound flow is also assigned a 64-bit value that is equivalent to the outbound flow's cookie-1. At this point, the ability to identify pair flows is now possible. We use this capability in the Flow Monitor module to compute the asymmetry of bytes and asymmetry of packets features between pair flows as part the detection process. Once a MUD file is parsed and the set of inbound and outbound flow rules are constructed, the controller pushes them to the associated datapath of the switch the IoT device has connected to. An example of a pair of MUD flows representing inbound and outbound rules for a particular device are shown highlighted in Figure 4.1.

```
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=15.069s, table=0, n_packets=11, n_bytes=995, idle_age=2, priority=20,udp,tp_dst=53 actions=output:3
 cookie=0x3e6355d16ebc4062, duration=12.257s, table=0, n_packets=15, n_bytes=1030, idle_age=2, priority=20,tcp,dl_dst=00:00:00:00:00:01,nw_src=192.168.3.2,tp_src=1234 actions=NORMAL
 cookie=0x3e6355d16ebc4063, duration=12.257s, table=0, n_packets=25, n_bytes=1790, idle_age=2, priority=20,tcp,dl_src=00:00:00:00:00:01,nw_dst=192.168.3.2,tp_dst=1234 actions=NORMAL
 cookie=0x0, duration=12.257s, table=0, n_packets=0, n_bytes=0, idle_age=12, priority=20,arp,dl_src=00:00:00:00:00:01,dl_dst=aa:bb:cc:dd:ee:00 actions=output:3
 cookie=0x0, duration=12.257s, table=0, n_packets=1, n_bytes=42, idle_age=12, priority=20,arp,dl_src=aa:bb:cc:dd:ee:00,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=12.257s, table=0, n_packets=11, n_bytes=1075, idle_age=2, priority=20,udp,dl_dst=00:00:00:00:00:01,tp_src=53 actions=output:1
 cookie=0x0, duration=15.069s, table=0, n_packets=3, n_bytes=1170, idle_age=12, priority=10,udp,tp_src=68,tp_dst=67 actions=CONTROLLER:65535
 cookie=0x0, duration=15.069s, table=0, n_packets=2, n_bytes=692, idle_age=12, priority=10,udp,tp_src=67,tp_dst=68 actions=NORMAL
 cookie=0x0, duration=15.069s, table=0, n_packets=1, n_bytes=42, idle_age=12, priority=10,arp,dl_dst=ff:ff:ff:ff:ff:ff actions=FLOOD
 cookie=0x0, duration=15.069s, table=0, n_packets=0, n_bytes=0, idle_age=15, priority=0 actions=CONTROLLER:65535
```

Figure 4.1. A pair of MUD flows (highlighted) in a switch's flow table

4.1.2 Flow Monitor

The Flow Monitor module is also an SDN application that is responsible for periodically collecting the Byte and Packet counters from every MUD flow that reside on each managed OpenFlow switch. It then will compute a vector of four features for each MUD flow, which is used to determine whether there is indication of that particular flow being abnormal. We ultimately decide for these four features to represent the *Byte count per second*, *Packet count per second*, *Byte count asymmetry*, and the *Packet count asymmetry*, of a given MUD flow at time t_n. We begin this subsection by detailing the justification for the selection of these four features, which are based on the reasonings presented by the works of Braga *et al* [25] and Yang *et al* [26].

4.1.2.1 Selection of Flow Features

The idea of using IP flow features that we can generate through available statistics in an OpenFlow switch can be attributed to the work laid down by [25]. As we had summarized in the previous chapter, their approach is a network-based mechanism that utilizes a Self Organizing Map in order to classify whether a 6-tuple of IP flow features belonging to a particular switch in the network is indicative of normal traffic or traffic belonging to that of DDoS attacks. This 6-tuple of features contain values representing the *Average of Packets per flow, Average of Bytes per flow, Average of Duration per flow, Percentage of Pair-flows, Growth of Single-flows* and *Growth of Different Ports*. While the authors of this work were able to demonstrate high quality results using these six features, we found that the benefits of said features do not translate over well into our approach. The primary reason of this is due to the

goal of the original detection methodology, which was meant to work specifically in the detection of DDoS occurrence within a network rather than detection near the traffic sources. As such, the method seeks to identify the OpenFlow *switches* that are highly indicative of DDoS traffic rather than identifying the individual *flows* that are responsible for the attack, which is the goal of our approach. We argue that detection near the source of attack traffic is preferred when concerned with IoT based DDoS. The vast number of IoT devices and the large volume of traffic that they can generate creates the need to be able to respond and block the sources of attack traffic before proliferation into the network, where further resources are wasted. By moving these defense mechanisms towards the source of traffic, we can stop this proliferation of noise into the network.

Despite these differences, we wanted to investigate whether these features can still be utilized for detection of abnormal traffic at the per-flow level, and if not, what were the reasons such that they were unnecessary for our use. We find that the use of MUD in our approach limits the benefits that these features can provide. As we defined in Section 4.1.1, the types of access control that MUD policies enforce are those such as specific port numbers used for connection or communication, the range of hosts a device is allowed to communicate with, and the types of protocols a device is allowed to speak. These limitations effectively make a majority of these features that were originally meant for the network-based DDoS detection, not the most suited for our use. We summarize the reasons why these features are not as effective in our scheme in Table 4.1.

Table 4.1. Summary of unused IP flow features

Average of Packets per flow	Intended to find the average of packets of *all* flows for a switch, not on a per flow level
Average of Bytes per flow	Intended to find the average of bytes of *all* flows for a switch, not on a per flow level
Average of Duration per flow	Intended to find the average duration of time a flow resides in a switch's flow table. This value is not applicable for our use by our application due to the assumption that a flow is only generated when the MUD controller is able to define a predetermined flow for a device. Furthermore, the duration of this flow is to reside in the switch for as long as the device's connection or until it needs to be updated, making this value inaccurate for our use
Percentage of Pair-flows	Intended to verify the number of pair-flows occurring in the flow stream during a certain interval. Because in our scheme flows are associated with devices, there will always be an inbound and outbound pair of flows for each MUD policy described for the device. The ratio between flows with a pair flow and flows without is thus meaningless in our scenario
Growth of Single-flows	Intended to capture beginnings of flooding attack in which number of flows can skyrocket (such as when IP spoofing is used). We acknowledge this feature as an important characteristic in detecting DDoS due to its ability to capture the occurrence of a device generating outbound flows without any correlating inbound flows. However because we predetermine these flows before hand and associate the flows with a device's MAC address, the occurrence of single "new" flows being emitted from the device is not possible, as they would all match with the MUD flows that were pre-installed. To mitigate this, we choose to use the features *Asymmetry of packets* and *Asymmetry of bytes* to capture this feature
Growth of Different Ports	Intended to capture the growth of ports during an attack (such as when port scans occur). In MUD, ports that a device is allowed to speak from or to are defined explicitly and are incorporated into the flow rules. As such, the growth of ports is a feature that is unusable as it is prevented from occurring. If a device should deviate from speaking through the specified ports, it is captured by the controller and an alert would be raised

In the publication by Yang *et al* [26], the authors observe that there are fundamental differences between normal traffic and attack DDoS traffic, specifically the presence of volumetric and asymmetric features. In a DDoS flooding attack, a large rate of traffic is deployed in order to improve the efficacy of flooding the netim. Furthermore, there is usually a large difference between the rate of traffic going towards the victim than responses from the victim during a DDoS attack, due to the use of IP spoofing in many attack cases. For these two observations, the authors define a 4-tuple of IP flow features that describe the *Byte Count per Second, Packet Count per*

Second, Byte Count Asymmetry, and *Packet Count Asymmetry* at time t_n. Because these four features are the result of aggregated statistics for each flow rather than all flows of a switch, we argue that we are able to use these features to detect changes at a *per-flow* level, enabling the capability of identifying abnormal flows from those that are not, a capability more in line with the goals of our approach. We ultimately decide on the use and measurements of these four features as part of the methodology. We argue our reasons for selecting these flows in Table 4.2.

Table 4.2. Summary of selected IP flow features

Byte Count per Second at time t_n	Intended to measure the rate of bytes per second sent from a particular MUD flow within a given time interval. Large deviations from the calculated weighted average of monitored bytes $(+-)$ a defined threshold results in an alert of suspected flooding behavior
Packet Count per Second at time t_n	Intended to measure the rate of packets per second sent from a particular MUD flow within a given time interval. Large deviations from the calculated weighted average of monitored packets $(+-)$ a defined threshold results in an alert of suspected flooding behavior
Byte Count Asymmetry at time t_n	Intended to measure the asymmetry of bytes between sent messages and responses. The closer to 0 this value is implies that more data is being sent out to the destination than received. The closer to ∞ this value is, implies that more data is being sent to the device than being sent to the destination. This feature is meant to capture the asymmetric feature of flooding attacks (in conjunction with the *Packet Count Asymmetry* feature), in which a balance of sent and received data may be skewed due to occurrences of IP spoofing
Packet Count Asymmetry at time t_n	Intended to measure the asymmetry of packets between sent messages and responses. The closer to 0 this value is implies that more packets are being sent out to the destination than be replied to. The closer to ∞ this value is, implies more packets are being sent to the device than being sent to the destination. This feature is meant to capture the asymmetric feature of flooding attacks (in conjunction with the *Byte Count Asymmetry* feature), in which a balance of sent and received packets may be skewed due to occurrences of IP spoofing.

Flow features selected were based on the work by Yang *et al* [26]

Having selected the features we want to use for differentiating between normal and abnormal traffic, we define the vector of features for a particular MUD flow at time t_n in Definition 4.1.4. We show how these features can be computed in Equations 4.1 to 4.4.

Definition 4.1.4. $V_{flow} =< ByteCountPerSecond,\ PktCountPerSecond,\ ByteAsym,\ PktAsym >$

$$Byte\ Count\ per\ Second\ at\ time\ t_n = \frac{ByteCount_{t_n} - ByteCount_{t_{n-1}}}{t_n - t_{n-1}} \tag{4.1}$$

$$Packet\ Count\ per\ Second\ at\ time\ t_n = \frac{PktCount_{t_n} - PktCount_{t_{n-1}}}{t_n - t_{n-1}} \tag{4.2}$$

$$Byte\ Count\ Asymmetry\ at\ time\ t_n = \frac{ByteCount_{t_n}^{flow_{in}} - ByteCount_{t_n}^{flow_{in}}}{ByteCount_{t_n}^{flow_{out}} - ByteCount_{t_n}^{flow_{out}}} \tag{4.3}$$

$$Packet\ Count\ Asymmetry\ at\ time\ t_n = \frac{PktCount_{t_n}^{flow_{in}} - PktCount_{t_n}^{flow_{in}}}{PktCount_{t_n}^{flow_{out}} - PktCount_{t_n}^{flow_{out}}} \tag{4.4}$$

4.1.2.2 Determining Abnormality of Flows

We define abnormal flows as the MUD rules whose behaviors, in terms of the four flow features described, have deviated far enough from the computed range of acceptable values determined for the specific MUD flow. These abnormal flows are those with flow features whose values display the volumetric and asymmetric characteristics common in DDoS flooding attacks. We model an average for the normal traffic flow of a MUD device by way of forecasting, specifically through the

computation of the Exponentially Weighted Moving Average (EWMA) technique for each of the four flow features measured at each interval. The Moving Average is a statistical calculation that is used to summarize past data points that have been recorded in specified periods or intervals [30]. The EWMA specifically, is a type of Weighted Moving Average in which exponentially decreasing weights are assigned to the set of observations. with the sum of these weights equaling a value that is very close to 1. The rate in which the value the weights are decayed is specified with a smoothing constant set to a value between 0 and 1. A big advantage of the EWMA is its simplicity and low memory requirements, needing only two pieces of data for computation, the observation of the current iteration and the previous iterations computed EWMA. The EWMA can be computed with with the equation shown in 4.5.

$$S_k = \alpha Y_k + (1 - \alpha)(S_{k-1}), \quad 0 < \alpha <= 1, \quad \text{for } k = 2, ..., m \tag{4.5}$$

where α is the smoothing constant used to determine the speed in which older responses are dampened, Y_k is the observed value at period k, and S_{k-1} is the previously computed EWMA at period $k - 1$. In our implementation we initialize the value of S_1 to the average of the first 5 observations collected for each flow, which is a general rule of thumb [30].

The following example demonstrates the formula's exponential property which we

take from an excerpt in the NIST Statistics Handbook [30].

$$
\begin{aligned}
S_k &= \alpha Y_k + (1-\alpha)(S_{k-1}) \\
&= \alpha Y_k + (1-\alpha)[\alpha Y_{k-1} + (1-\alpha)(S_{k-2})] \\
&= \alpha Y_k + \alpha(1-\alpha) Y_{k-1} + (1-\alpha)^2 (S_{k-2}) \\
&= \alpha Y_k + \alpha(1-\alpha) Y_{k-1} + (1-\alpha)^2 [\alpha Y_{k-2} + (1-\alpha)(S_{k-3})] \\
&= \alpha Y_k + \alpha(1-\alpha) Y_{k-1} + \alpha(1-\alpha)^2 Y_{k-2} + (1-\alpha)^3 (S_{k-3}) \\
&\ldots
\end{aligned}
$$

As each previous EWMA equation is substituted into the formula, it can be seen that the value of the weights are equal to the value $\alpha(1-\alpha)^k$, which decrease in a geometric fashion as k increases. The parameter α controls the speed in which the weights of previous observations are dampened, with a value of 1 meaning that only the most recent observation has impact on the EWMA, and a value closer to 0 giving more weight to the older collected data. We choose to use the EWMA as the method of representing what the normal metrics of a MUD flow may look like, taking into account both past and potential future deviations. An exponentially weighted moving average is computed for each feature in the MUD flow vector, V_{flow}, at period k. The computed averages are then used to determine the upper and lower bounds that the features computed at the next period $k+1$ should fall within. The method in which the bounds are determined derive from the work of [26], where the three-sigma rule of the Gaussian Distribution is used in order to account for changes between historical and current measurements. Using the assumption that the four traffic features of a benign MUD flow should fall within some degree of the computed mean (in the manner of a normal distribution), we define the upper bound in Equation 4.6 and lower bound in Equation 4.7 as follows.

$$Upper_i = \mu_{EWMA_i} + 3 * \sigma \tag{4.6}$$

$$Lower_i = \mu_{EWMA_i} - 3 * \sigma \tag{4.7}$$

where the subscript $i = 1, 2, 3, 4$ and represents each element in the MUD flow vector V_{flow}, μ_{EWMA_i} is the EWMA of the ith element, and std_i is the weighted standard deviation of the i'th element. To compute std_i, we take the square root of the exponentially weighted moving variance of element i, represnted as σ_k^2. We use the equation presented by [31]:

$$\sigma_k^2 = (1 - \alpha)(\sigma_{k-1}^2 + \alpha(Y_k - \mu_{EWMA_{k-1}})^2) \tag{4.8}$$

where σ_{k-1}^2 is the square of the standard deviation that was computed in the previous period $k - 1$, Y_k is the observed value at period k, and $\mu_{EWMA_{k-1}}^2$ is the EWMA computed in the previous period $k - 1$. The initial standard deviation and variance are computed normally using the first five observations, same as the EWMA. Assuming that the EWMA and acceptance range was computed at period k, we take the observations made at time $k + 1$ and compare each element in the generated vector V_{flow} with the computed bounds. If any of the values of the four elements are found to be within their respective bounds, then we determine this traffic as normal and begin computing the next iteration of feature values that will be used in iteration $k + 2$. Otherwise, if all four elements fall out of bounds, then we determine the MUD flow to be abnormal, to which additional action may be taken. Categorization of a flow as abnormal is only done when all four elements fall out of bounds, thus demonstrating that both the required volumetric characteristics (*Bytes per Second*, *Packets per Second*) and asymmetric characteristics (*Asymmetry of Bytes*, *Asymmetry of Packets*)

are present. It is entirely possible in cases where slight deviations may demonstrate a spike in traffic that resembles that of an attack. However, if the asymmetry features are within an expected range, then we can assume that because the server is responding to the device for every request sent, the asymmetric characteristic is not present, thus that flow is not indicative of an attack. In our implementation, we react to a detection alert installing a flow rule such that it will drop all traffic stemming from said device, preventing the attack traffic from propagating into the network. Another option could possibly be the modification of the MUD rule for redirecting or forwarding the traffic to another entity to perform Deep Packet Inspection, furthering the capabilities of building additional profiles for a network's Intrusion Detection System (IDS) or Intrusion Prevention System (IPS). The procedure of the Flow Monitor can be summarized in the Figure 4.2 below.

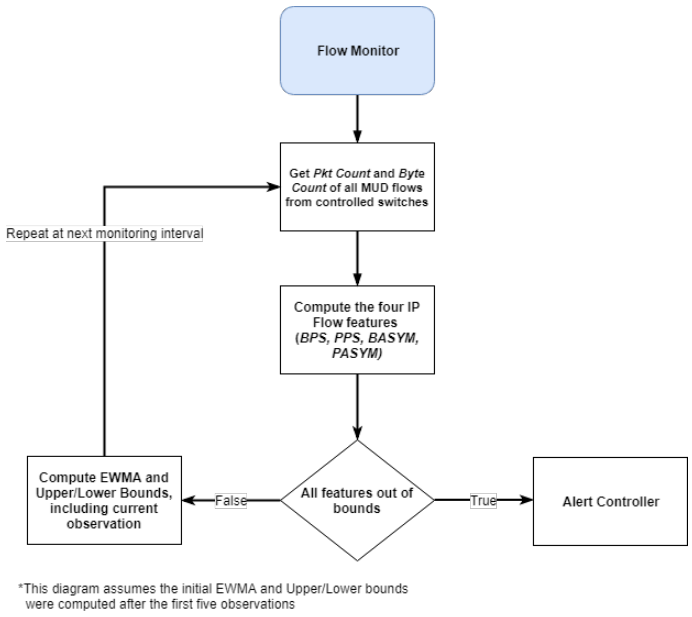


Figure 4.2. SDN Topology in Mininet

4.2 Testbed Design and Network Topology

A custom SDN topology, pictured in Figure 4.3, was built using the network emulation orchestration system, Mininet [32]. The virtual testbed is designed with the scenario where the MUD architecture is deployed as a security service located near the sources of the IoT traffic and has the capability of orchestrating and configuring the SDN switches in the respective "private" networks. We aim to demonstrate how MUD could fit into a *plug in and forget* type of design, similar to that mentioned in the work of [23], in which end users connect their IoT devices and can assume this third party will configure the network with the device's given policies automatically. It is also on this testbed where we enact our attack and defense simulations in order to evaluate the approach.

The overall topology consists of four networks. The first network is the physical network of our machine, or the local host, that is separated from the Mininet virtual network. The Ryu SDN controller [33] resides on this network and has control management over the three other virtual networks in the topology. Additionally, we run our two SDN applications, the MUD controller and the Flow Monitor, on top of the controller. This network represents a third party entity (such as possibly an ISP) which has network vision over the individual "private" networks managed such that it may perform the security measures for them by way described in the MUD architecture, coupled with our approach. We also place the MUD server on this network which we implement in Python by modifying the Python module, SimpleHTTPServer, such that TLS is enabled for the network sockets used by the webserver. The MUD server, as previously stated, is envisioned to be hosted by the manufacturer of the products, so normally it is expected that they reside on the Internet at some remote

host. However, due to the Mininet network by default being separated from the physical LAN, we choose to simplify the setup by not introducing an additional VM to act as a "Internet" server and just have the MUD server reside on the same network as the controller.

The next two networks represent two private or small office IoT networks that are comprised of virtual hosts which are meant to be portrayed as limited-purpose IoT devices that all have MUD policies associated with it. The first IoT network is assigned the 192.168.1.x block, and the second is assigned the 192.168.2.x block. The hosts in each network are connected to a corresponding OpenFlow enabled software switch that Mininet is capable of using called Open vSwitch [34]. These switches in the IoT networks are represented as *s1* and *s2*. We treat each virtual host created by Mininet as a MUD enabled IoT device, in which a MUD policy has been generated for each. To communicate the MUD URL for the MUD controller to retrieve the policy, we modify the `dhclient.conf` file used by the DHCP client, dhclient [35] [36], in order to enable the capability of sending DHCP 161 option to signify the support for MUD that the device has.

Finally, a fourth network is assigned the 192.168.3.x block and is one we loosely represent as the *Internet*. We use this network to simulate where an external host would reside, and in our scenario, we implement a TCP echo server using the ncat tool [37], to function as an arbitrary IoT application for the MUD IoT devices to communicate with. This server sits on 192.168.3.2 and listens on port 1234 for data from the "devices" in the IoT network, which it simply echos or replies back when received. While a third software switch, *s3*, exists in this network and is technically still managed by the SDN controller, for our intents and purposes, we do not push any flow rules to the *s3* switch in order to allow the node in the `192.168.3.x` network

to act as independently as possible. In this design, the MUD controller would reside on this network as an SDN application and through the SDN controller, have the capabilities of managing each of the IoT networks within its zone.

To enable communication between all three networks, we add a host, *r1*, that is connected to *s1, s2,* and *s3*, and effectively functions as a router to forward packets between the networks. We assign the address 192.168.1.1 to the *r1-eth0* interface to serve *IoT Network 1*, and the 192.168.2.1 address is assigned to the *r1-eth1*to serve *IoT Network 2*. Two applications run on *r1*, a static DHCP server that serves addresses to *IoT Network 1* and *IoT Network 2*, and a DNS server that resolves the remote controller's name residing on 192.168.3.2. We make use of the open-source Python DHCP server implementation known as staticDHCPd [38] and the Dnsmasq software as the DNS server [39]. The 192.168.1.1 address is set to be the default gateway, DHCP server, and DNS server for the devices in *IoT Network 1* and the same is done for *IoT Network 2* with the address 192.168.2.1.

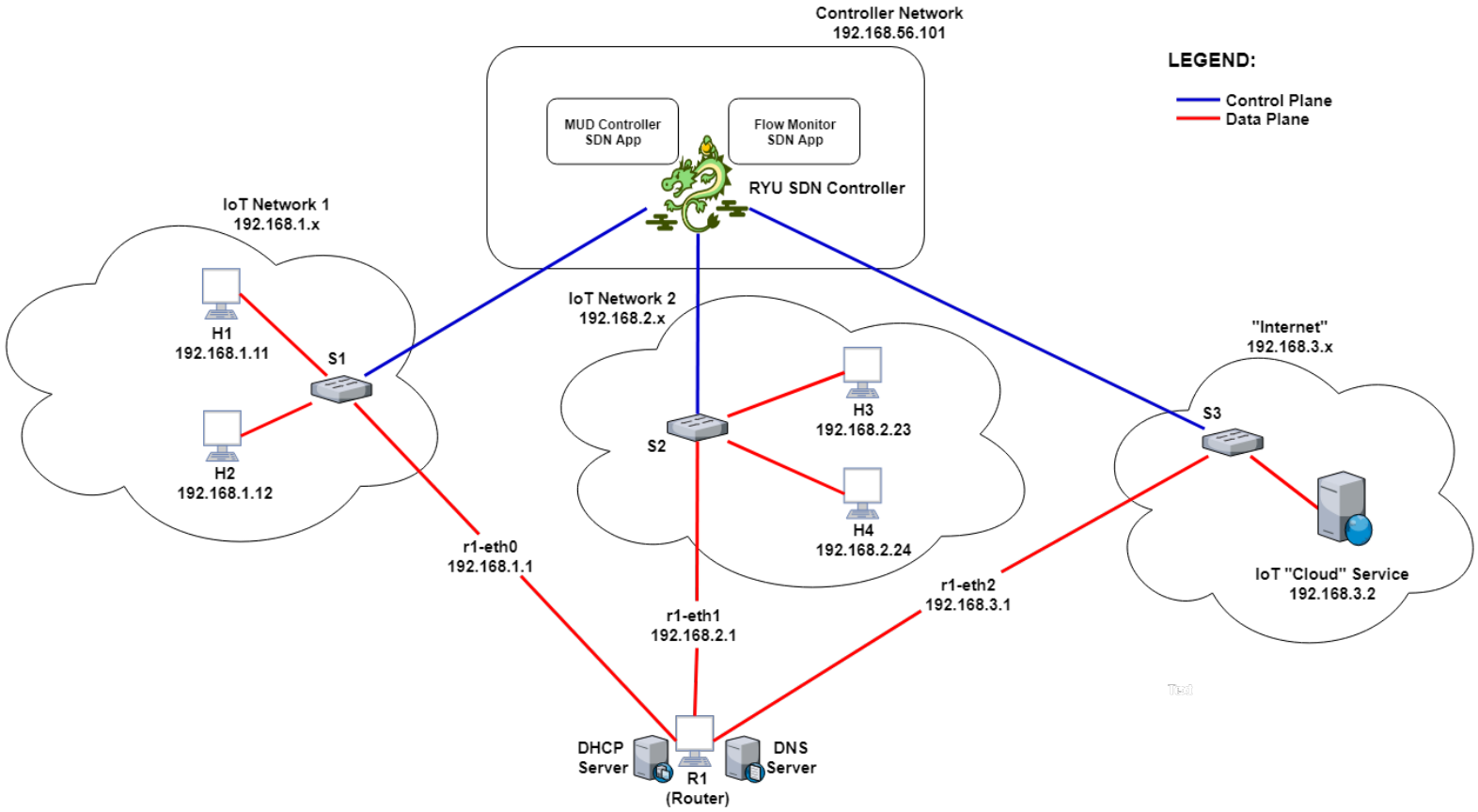


Figure 4.3. SDN Topology in Mininet

4.3 Security Considerations

The use of Software Defined Networks in our approach means that we inherit all the vulnerabilities and security flaw associated with SDN and the selected APIs, specifically the OpenFlow protocol. One of the primary concerns when using SDN is the possibility of malicious attackers committing a DDoS attack against the controller itself. The centralized nature of the controller means that it is a point of failure for the network as without its availability, the entire network functionality ceases to operate [19]. Attackers can achieve this through the launching of controller plane attacks such as *Packet In Swamping* in which large volumes of OpenFlow `PACKET_IN` events are generated. Oftentimes in SDN networks, when a new packet is received by a switch with no rules to match it with, this packet is sent to the controller for processing and decision handling, generating a `PACKET_IN` event. Because most DDoS attacks use IP spoofing of sorts, flooding a SDN network with spoofed addresses will generate a large volume of rule misses, causing the switch to send the packet to the controller. Given a large enough volume of traffic sent to the controller for processing, the bandwidth between the controller and switch channel can be compromised, along with the controller's processing power. The consequence is the eventual resource exhaustion of the controller that causes legitimate traffic to be unprocessed.

The consequences of such kinds of DDoS attacks on the SDN controller itself can be alleviated through our proactive approach of using MUD to define the permitted communications to and from devices. As described in the work [40], the use of access control as a SDN security solution is low cost and feasible for networks in which traffic is likely to go outbound and the inbound traffic is generally trusted and well known to the network. While our use of MUD access controls is meant to be

specify the permitted communications to and from the MUD enabled IoT devices, the SDN controller also inherits the benefits of the flow rule enforcement that help in alleviating the load upon itself. For example, when a MUD device is connected to the OpenFlow switch, inbound and outbound rules are installed in correspondence to the permitted communications. Should a MUD device act out of the defined set of rules, the device will be blocked with the installation of a new flow rule that is matched to the MAC address of the device. Because we identify devices by their MAC addresses, we are able to effectively aggregate multiple possible (permitted) flows from a particular device to one flow rules such that even if the device is rapidly emitting spoofed IP addresses, the flows will be matched to the device itself, and thus match with the blocking rule that has been installed for it. This means that the spoofed traffic is not sent to the controller for processing, alleviating the load placed on the controller. On the other hand, should a device attempt to connect to a host in the SDN IoT network that is unexpected, this will be dropped as well due to the communications not matching with the expected inbound flow rules, allowing for the aversion of inbound attack traffic that may be being sent to the SDN network from needing to be processed by the SDN controller.

Chapter 5

EVALUATION AND RESULTS

5.1 MUD Process

All experiments that were conducted were run on top of the custom Mininet topology created. Upon launching the Mininet topology, several events take place. When the Ryu SDN controller establishes a session with an OpenFlow switch, four default flow rules are pushed into the switch's flow tables by the MUD controller application; one for ARP (Address Resolution Protocol) broadcasts such that the IP address of the default gateway may be found, one for forwarding DHCP (Dynamic Host Configuration Protocol) client flows, one for forwarding DHCP server flows, and finally a flow for forwarding DNS queries. Depending on which of these default rules are hit, we apply different OpenFlow defined forwarding actions to be applied to the matching flows. If the ARP broadcast rule is hit, we apply the `OFPP_FLOOD` forwarding action that uses the normal pipeline of the switch to flood the request to the network. If the DNS query rule is hit, then we forward it to the port in which the DNS server is connected to (*r1*). When a DHCP packet from source port 68 to destination port 67 is matched with the DHCP client flow, this packet is sent to the controller to see if the MUD option is present, and to retrieve and process the MUD policies if it is. Responses from the DHCP server back to the client are forwarded normally by applying the `OFPP_NORMAL` forwarding action.

The Mininet hosts representing the MUD enabled IoT devices will then initiate the DHCP protocol by running the dhclient DHCP client software. We modified the

configuration file used by dhclient `dhclient.conf` to add a custom DHCP option representing MUD support. This option is represented with the value of 161 as described and reserved by the authors of the MUD draft to the Internet Assigned Numbers Authority (IANA) [11]. Since the DHCP Discover and DHCP Request will both match with the default client flow, both packets will be sent to the controller for processing. The controller will forward the initial DHCP Discover message as it indicates the client broadcasting a request to find the DHCP server. After receiving the DHCP Offer from the DHCP server residing on address 192.168.3.2, the client will send a DHCP Request message. This message will again be sent to the controller, however this time we inspect to see if the option 161 is present. Assuming it is, the MUD controller will begin initiating the MUD process so that the policies for the connecting MUD IoT device may be retrieved, then translated into flow rules. The MUD file retrieval is begun through sending an HTTP GET request by the MUD controller to the MUD URL that the MUD controller extracted from the DHCP Request packet. Assuming that the MUD URL is resolvable, the MUD server that is contacted will return the MUD file serialized in a JSON format, containing the MUD information and access control lists for the given device. Normally the MUD controller would also retrieve the signature file to validate the MUD file, but as stated in the Section 4.1.1, we assumed that the MUD file is valid so we skip this part of the MUD specification.

Upon receiving the MUD file from the MUD server, a *MUD entry* object is initialized, containing the inbound and outbound access control lists with resolved domain names (if any) for the connecting MUD IoT device. The inbound and outbound access control lists are then passed to another function to be translated into OpenFlow rules to be pushed into the corresponding OpenFlow switch or switches. The parsing

of the MUD file into flow rules are laid out in Chapter 4, Section 4.1.1. For every MUD enabled device that is connected to the OpenFlow switch, there are $2x+3$ flow entries that are installed into the switch flow table. The variable x represents the number of access control entries that a manufacturer has defined. This value is multiplied by two to accommodate for both inbound and their corresponding outbound access control entries. Finally, three additional rules are added for each MUD device; one for the IoT device to respond to the gateway during ARP, one for the gateway to reach back to the IoT device during ARP, and finally a flow rule for DNS query responses from the DNS server. Figure 5.1 summarize the flow of these events.

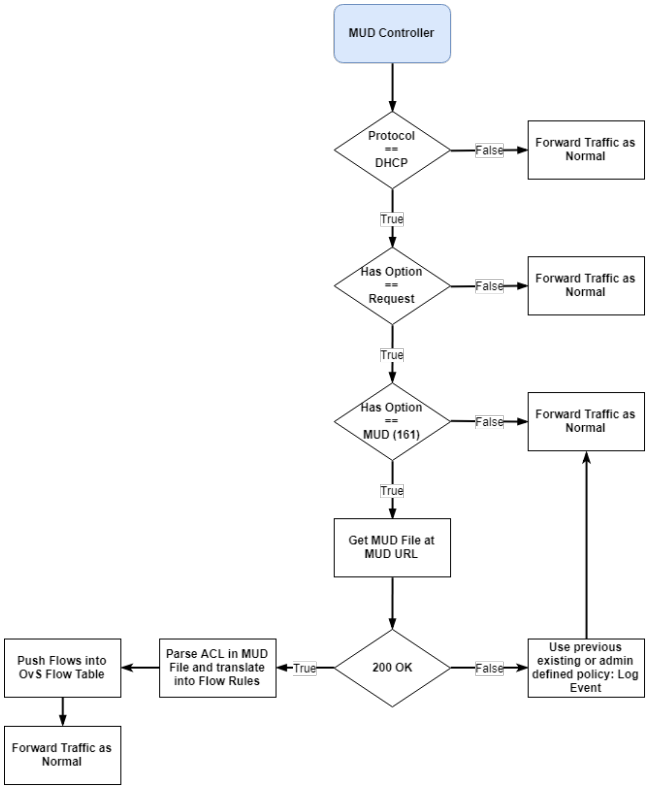


Figure 5.1. MUD Controller process flow using MUD DHCP Method

5.2 Impact of the EWMA Tuning Parameter

A series of trials were conducted to identify how the α parameter used in the Exponentially Weighted Moving Average (EWMA) function impacted the detection of deviating values for the four flow features computed at each monitoring interval. We test five values for α; 0.05, 0.10, 0.15, 0.25, and 0.50. Due to the α parameter used to determine the weights of the data used for the moving average, we hypothesized that using a larger value of α would result in higher false positives due to the bounds computed becoming tighter as a result of the newer data contributing more heavily to the calculated moving average, and thus contributing more to the computed standard deviation of the moving average. An α value of 1 can be thought of as taking only the newest observation into consideration, and within the scope of our approach, it can be interpreted as the expectation of every subsequent traffic flow features observed to be the same as the last observed iteration. Depending on the type of environment whether traffic is stable or dynamic, this value is highly variable and application specific and we seek to find a value that is able to provide low false positive occurrences in general.

For each α value, we ran 10 trials, each lasting 10 minutes, on the custom Mininet topology. A simple Python program was executed by each Mininet host representing the MUD IoT devices in order to generate periodic benign TCP traffic towards the permitted destination as processed by the MUD controller. TCP traffic was sent every second with random data generated to be between 2 and 512 bytes. Within the Mininet topology, bidirectional links between the three switches and the router node *r1* is set to 15 Mbps bandwidth, 2ms delay, and 2% chance of packet loss with the purpose to introduce a degree of variance to the observed traffic. Each trial was ran

for approximately 10 minutes in which the occurrences of feature values, exponentially weighted moving averages of each features, upper/lower bounds, and false positive occurrences were recorded at each monitoring interval, set at 3 seconds. We make use of equation 5.1, that was used in the work by Braga *et al*[25] to calculate the rate of false positive occurrences, which was found by taking the occurrences of False Positives (FP) over the sum of False Positives and legitimate traffic or True Negatives (TN).

$$FPR = \frac{FP}{TN + FP} \tag{5.1}$$

As hypothesized, as α increases, the occurrence of false positives becomes greater, with almost an 88% false positive rate over 10 trials when $\alpha = 0.50$. The rate of false positives lowers significantly to 20% as the value of the tuning parameter equals 0.25, and when $\alpha = 0.15$, the false positive rate is 5% over 10 trials. This correlation of increasing alpha value and false positive rate can be attributed to a larger alpha value translating into a smaller sample window that make up the average. If more weight is given to the more recent data sample, then the upper and lower bounds calculated for the next observation are going to expect it to be strictly similar to the previous, thus causing a false positive alert even if the deviation is in reality, quite small. These results are summarized in Table 5.1 below.

Table 5.1. Normal TCP Traffic False Positive Frequency Based on α

α	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**	FPR **(%)**
0.05	0	0	0	0	0	0	0	0	0	0	0.00
0.10	0	0	0	0	0	0	0	0	1	0	2.50
0.15	0	0	0	0	0	2	0	0	0	0	5.00
0.25	1	1	0	0	1	0	1	1	2	1	20.00
0.50	3	3	3	3	4	4	4	3	4	4	87.50

Often times the false positives occurred during small spikes or dips in traffic that were not very significant when compared with all previously observed flow features, but enough so such that the values fell out of bounds for the *Bytes per Second*, *Packets per Second*, *Asymmetry of Bytes* and *Asymmetry of Packets* features. By setting a higher α value, the upper and lower bounds are generally tighter and adjusted as according to the newest observed data, thus leading to a higher occurrence of false positives early on during the monitoring stages. Visualization of the differences in the bounds computed for different α values are shown in the following eight graphs below. These graphs represent the flow features captured for a false positive occurrence in Trial 6 when $\alpha = 0.15$ and another false positive occurrence in Trial 6 when $\alpha = 0.50$. In both of these scenarios, the Mininet host representing IoT device H4 were falsely flagged as malicious.

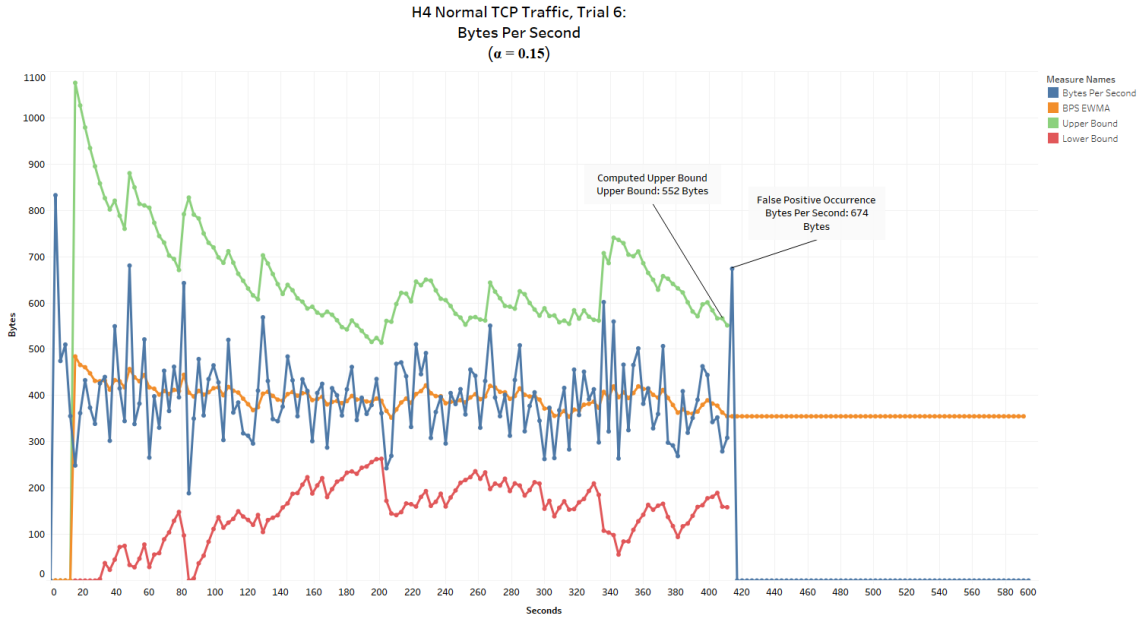


Figure 5.2. H4: Bytes Per Second when $\alpha = 0.15$

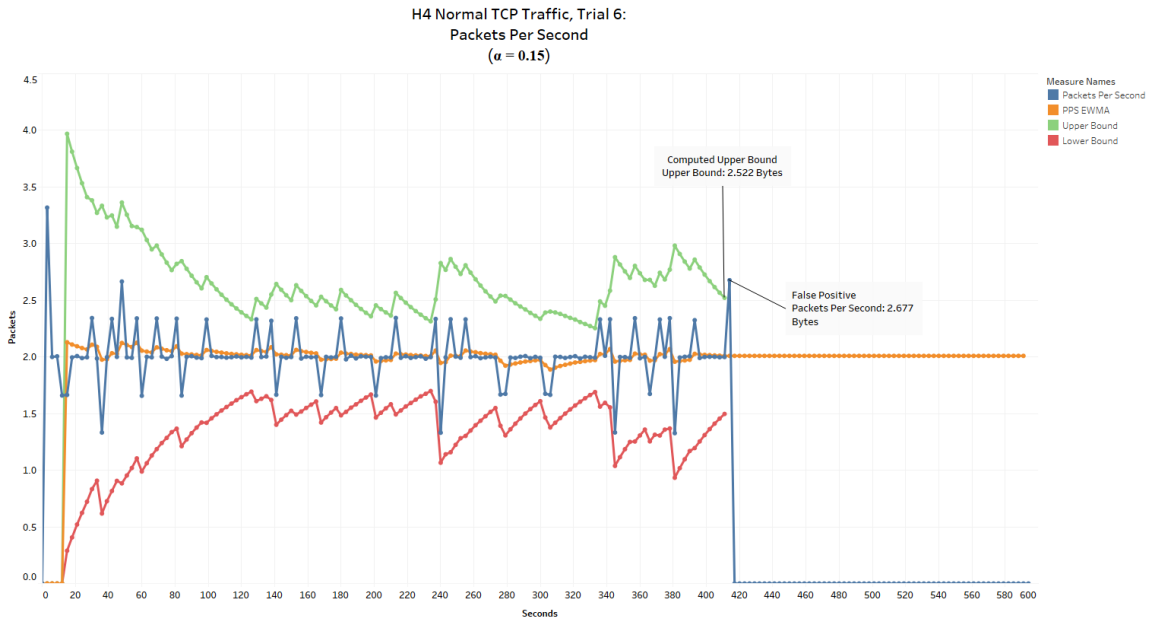


Figure 5.3. H4: Packets Per Second when $\alpha = 0.15$

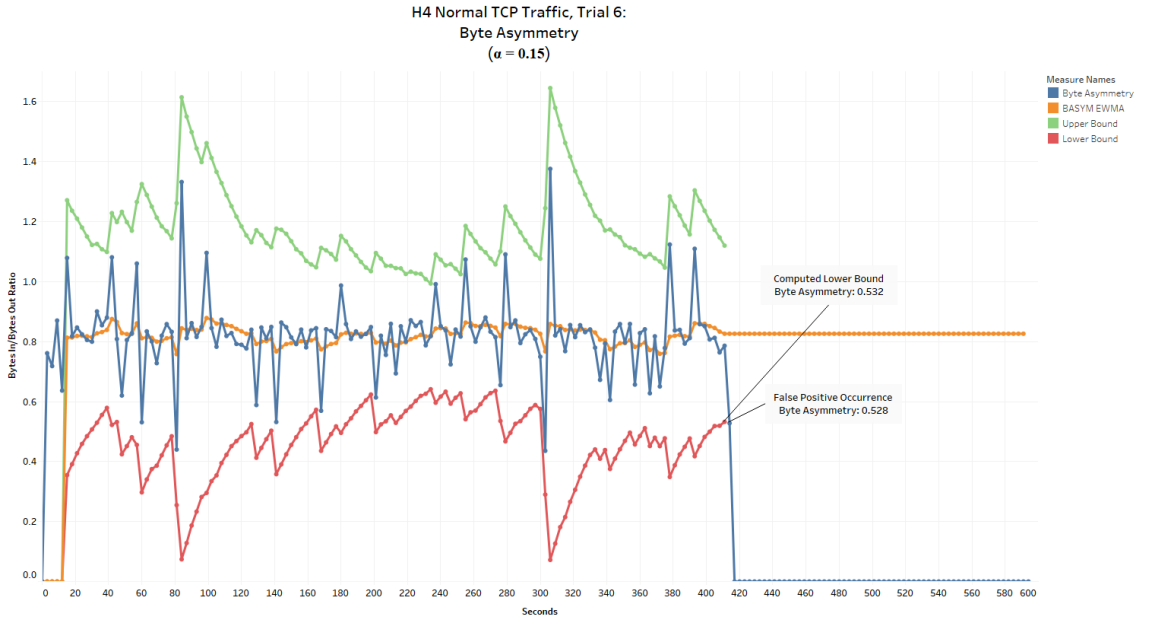


Figure 5.4. H4: Bytes Asymmetry when $\alpha = 0.15$

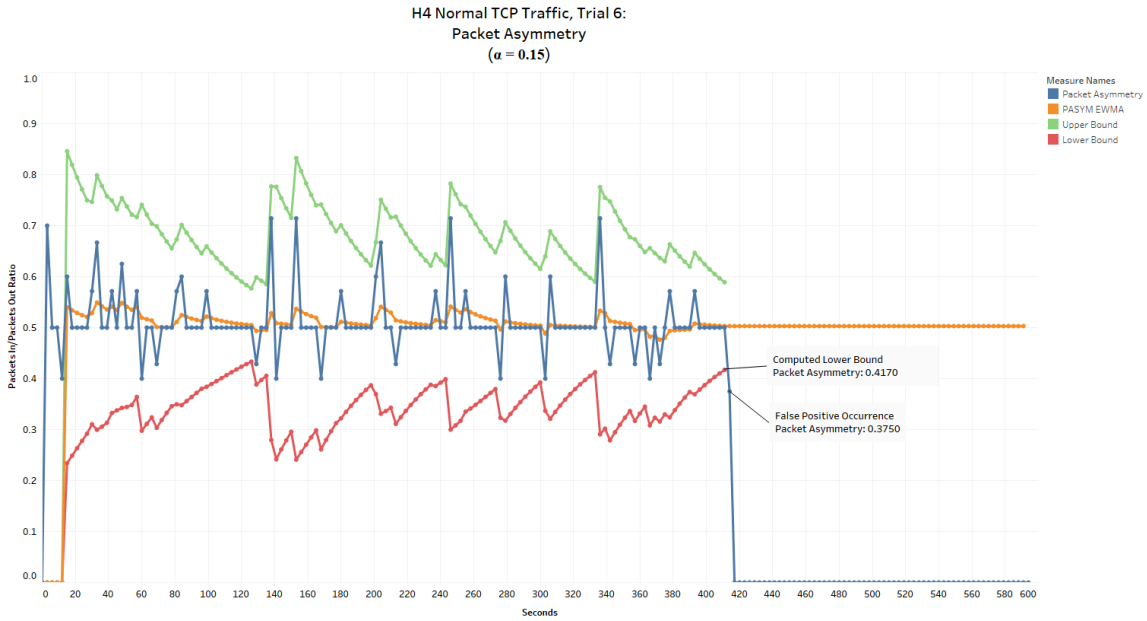


Figure 5.5. H4: Packets Asymmetry when $\alpha = 0.15$

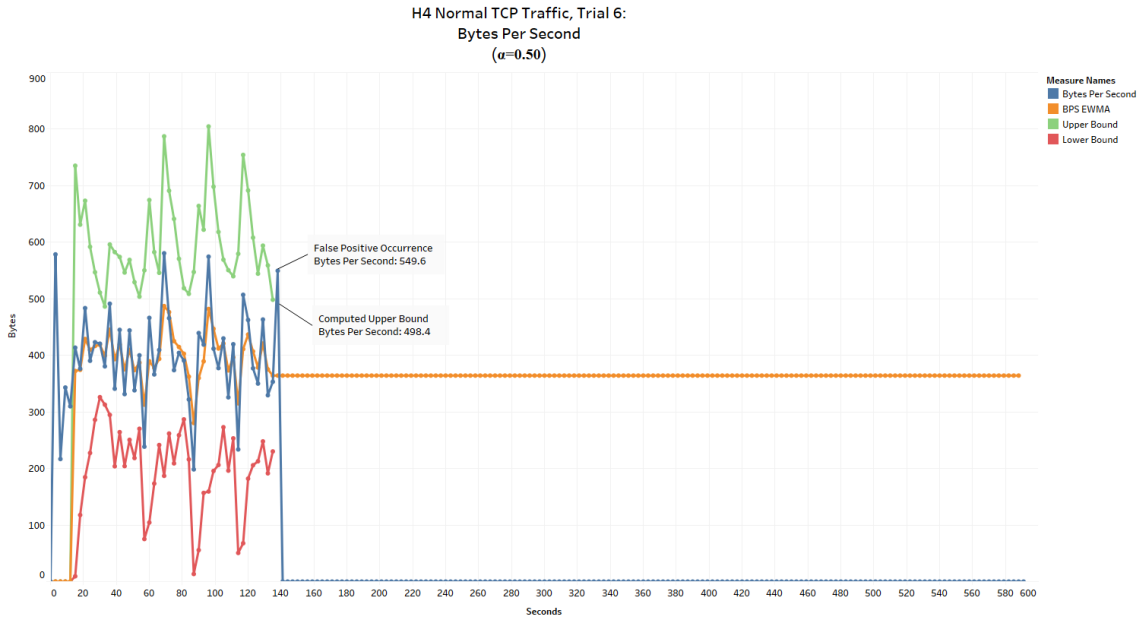


Figure 5.6. H4: Bytes Per Second when $\alpha = 0.50$

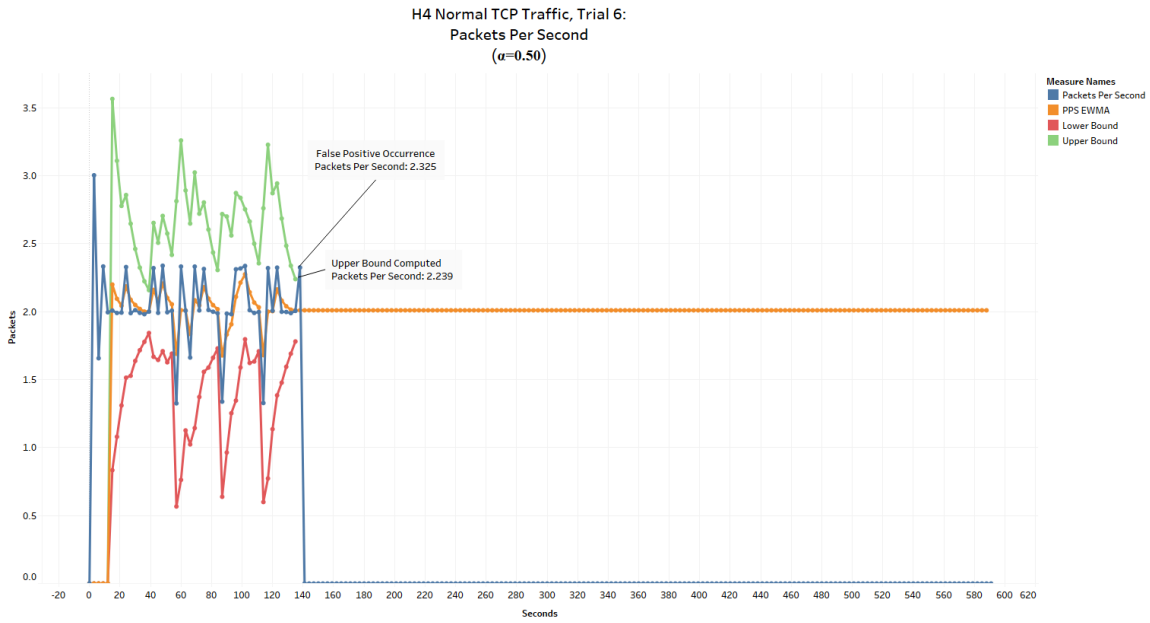


Figure 5.7. H4: Packets Per Second when $\alpha = 0.50$

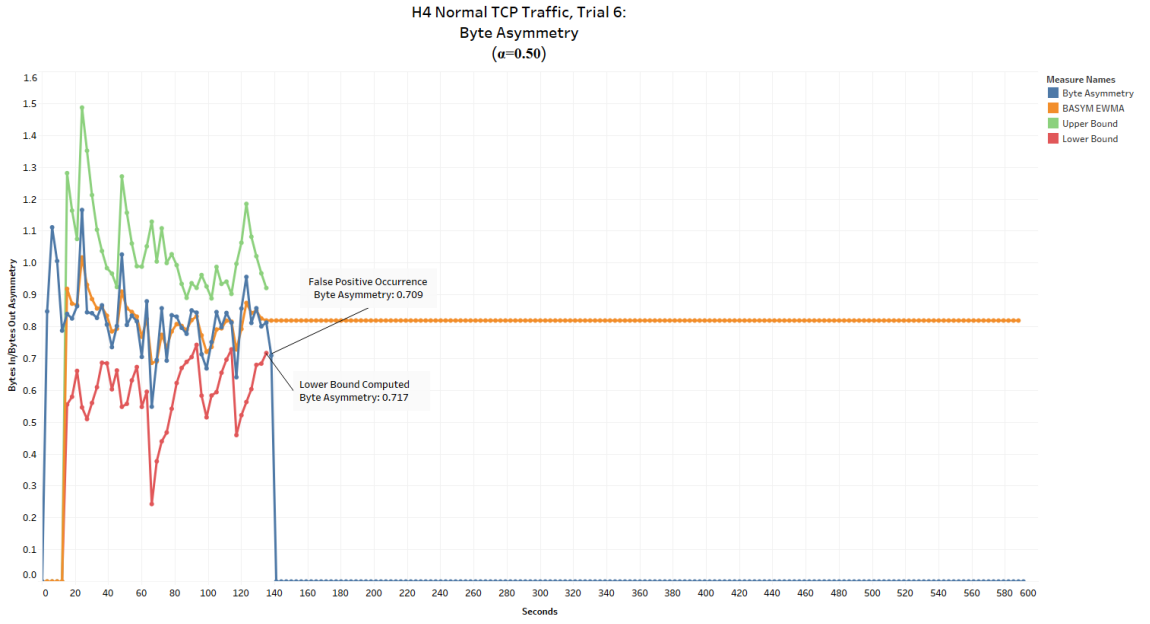


Figure 5.8. H4: Bytes Asymmetry when $\alpha = 0.50$

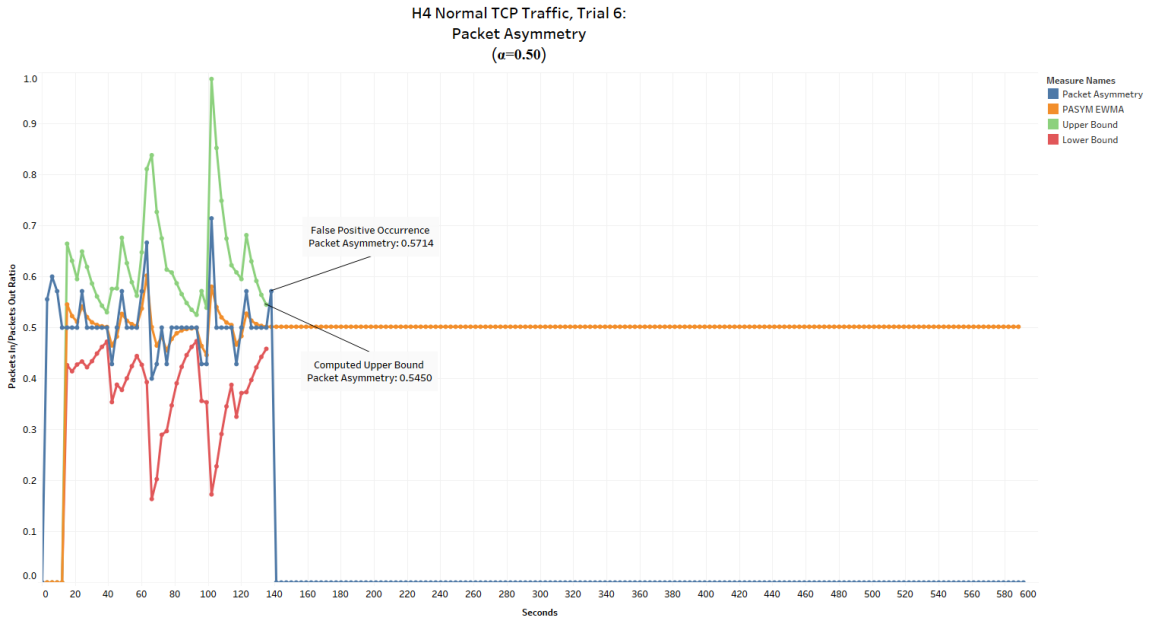


Figure 5.9. H4: Packets Asymmetry when $\alpha = 0.50$

5.3 TCP SYN Flood Attack

TCP SYN Flood attacks are one of the most common form of DDoS flooding attacks and is a type of attack that the IoT malware Mirai is capable of initiating. In this attack, exploitation of the TCP three-way handshake is performed by malicious actors sending large volumes of TCP SYN packets, usually spoofed, towards the victim in an attempt to use up all the server's resources due to their allocation and binding to the half-opened connections awaiting the client's SYN-ACK reply. TCP can be considered a symmetric protocol due to all data needing to be acknowledged by each party before sending the subsequent sets of data. This means that a skew in the ratio of traffic going out versus coming in along with the conjunction of a large volume of packets and bytes captured per flow, will indicate that an attack is most likely occurring.

A MUD file was generated using https://mudmaker.org/alpha/ to specify the

communications permitted to and from the Mininet hosts representing the MUD IoT devices. For this specific test, we specified that in addition to the default rules described in Section 5.1, the hosts may only be allowed to send out TCP traffic to the remote controller identified by the URL `service1.example-iot-service1.com` that is running a simple echo server listening on destination port 1234. For to-device traffic, the host was only permitted to receive TCP traffic from the remote controller and source port 1234. As mentioned in Chapter 4, Section 4.2, this URL will resolve to the IP address 192.168.3.2 which resides on the remote network. The resulting flow rules that were generated by the MUD controller from the MUD file were then pushed into the host's corresponding OpenFlow switch, which were then capable of enforcing these access controls.

Table 5.2. Match Fields of From-Device Flow Rules (Trial 1)

Host	**Ether Src**	**IP Dst**	**IP Proto**	**Src Port**	**Dst Port**
H1	00:00:00:00:00:01	192.168.3.2	TCP	*	1234
H2	00:00:00:00:00:02	192.168.3.2	TCP	*	1234
H3	00:00:00:00:00:03	192.168.3.2	TCP	*	1234
H4	00:00:00:00:00:04	192.168.3.2	TCP	*	1234

Note: H1 and H2 rules reside on Switch *s1* in IoT Network 1 while H3 and H4 rules reside on Switch *s2* in IoT Network 2. '*' indicates wildcard.

Table 5.3. Match Fields of To-Device Flow Rules (Trial 1)

Host	**IP Dst**	**Ether Src**	**IP Proto**	**Src Port**	**Dst Port**
H1	192.168.3.2	00:00:00:00:00:01	TCP	1234	*
H2	192.168.3.2	00:00:00:00:00:02	TCP	1234	*
H3	192.168.3.2	00:00:00:00:00:03	TCP	1234	*
H4	192.168.3.2	00:00:00:00:00:04	TCP	1234	*

Note: H1 and H2 rules reside on Switch *s1* in IoT Network 1 while H3 and H4 rules reside on Switch *s2* in IoT Network 2. '*' indicates wildcard.

The tests were ran on our the MUD SDN Mininet topology where bidirectional links between the three SDN switches and the router (*r1*) were set such that band-

width between them was 15 Mbps, with 2ms delay and 2% probability of packet loss, to introduce variance of the flow observations. We chose hosts H1 (192.168.1.11) and H3 (192.168.1.21) to be the attackers in our experiments. Upon each host's connection to the OpenFlow switch and the resolution of the MUD process, all four hosts would begin generating normal TCP traffic. H1 and H3 were programmed to generate normal traffic for period 60 seconds upon connecting to their respective networks. Upon the end of this period of normal traffic, H1 and H4 would then begin flooding TCP SYN messages to the victim at 192.168.3.2 while H2 and H4 would continue emitting normal traffic. We used the network tool hping3 [41] to perform the DDoS attack. We conducted several experiments with varying monitoring intervals, time of attack, and α values. Each scenario that was tested consisted of 30 trials, each lasting 3 minutes. Along with Equation 5.1 to calculate the False Positive Rate of our approach, we also used Equation 5.2 to compute the Detection Rate, which can be found by taking the occurrences of True Positives (TP) over the sum of True Positive and attack traffic that was misinterpreted as legitimate, or False Negatives (FN). Our results can be summarized in the following two tables, which show the Detection Rate and False Positive Rate for the different combinations of α, time of attack, and monitoring interval chosen.

$$DR = \frac{TP}{TP + FN} \tag{5.2}$$

Table 5.4. TCP SYN Flood Detection Results (Attack Started at 40 Seconds)

	3 Sec Interval		**5 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**
$\alpha = 0.10$	98.31	3.39	100.00	0.00
$\alpha = 0.15$	100.00	6.45	100.00	0.00

Table 5.5. TCP SYN Flood Detection Results (Attack Started at 60 Seconds)

	3 Sec Interval		**5 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**
$\alpha = 0.10$	100.00	1.64	100.00	0.00
$\alpha = 0.15$	100.00	3.33	100.00	0.00

Our results show accurate Detection Rates for TCP SYN Flood attacks launched at both 40 and 60 seconds for alpha values 0.10 and 0.15. An increase in monitoring time provides for more accurate Detection Rate. However, the trade off in accuracy suggests that in the worst case, the attack will be delayed by n seconds, where n is the selected monitoring interval. However, selecting a shorter n also has the implications of increased overhead between controller and switch communication due to the more frequent polling of statistics from the OpenFlow switches. The false positives occur infrequently in comparison to correct abnormal flow detection due to our use of the four flow features and the condition that all four observed features must fall out of bounds to constitute an alert. It is entirely possible that more data or packets is sent outbound, going above the computed upper bound. However, if the byte and packet symmetry is maintained by receiving the expected ratio of inbound traffic per outbound traffic, then the alert is not generated and the EWMA and upper/lower bounds are adjusted to match the recently observed flow features.

Another TCP SYN Flood scenario was conducted to observe any possible degradation in Detection Rate in larger networks versus the smaller counterparts. In these new topologies, we increased the size of each network from two hosts to five hosts, and

increased the total number of networks managed to 4 and 8. The same bidirectional link parameters were maintained, while an α value of 0.10 was used. The detection results of the experiments conducted on this larger network are shown in Table 5.6.

Table 5.6. TCP SYN Flood Detection Results (Larger Topology)

	3 Sec Interval		**5 Sec Interval**		**10 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**	**DR(%)**	**FP(%)**
4 Networks, 20 Hosts	98.32	1.91	100.00	0.66	100.00	1.25
8 Networks, 40 Hosts	99.13	7.78	100.00	21.76	100.00	21.28

When the approach is applied to a larger network, it was observed that there is not much degradation in regards to the Detection Rate and that these values seem to be aligned with the smaller topology counterpart. However, a noticeable and significant degradation in False Positive Rate can be observed, specifically when the monitoring interval is increased to 5 and beyond that the FPR rose about 20%. In the scenario in which we tested our approach on the topology consisting of 8 networks, it can be seen that the FPR reached as high as approximately 22%. In this specific case, a large majority of false positives were traffic samples that fell below the computed average and lower bounds, in regards to the flow features of *Bytes Per Second* (BPS) and *Packets Per Second* (PPS). We re-tested our approach on the same 8 network topology, this time removing the lower bound limitation. We argue that a lower bound applied to the *Bytes Per Second* and *Packets Per Second* features are not necessary for capturing volumetric features of DDoS flooding attacks. As shown in Table 5.7, this change allowed us to drastically reduce the FPR to about 2%, a difference of 20%.

Table 5.7. TCP SYN Flood Experiment (Removed BPS and PPS Lower Bounds)

	3 Sec Interval		**5 Sec Interval**		**10 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**	**DR(%)**	**FP(%)**
8 Networks, 40 Hosts	99.13	0.60	100.00	1.19	100.00	1.76

5.4 UDP Flood Attack

UDP Flood attacks are another common type of DDoS flooding attack in which malicious actors flood (usually spoofed) UDP packets targeting random destination ports of the victim. The result is the victim being forced to send out large volumes of ICMP Destination Unreachable messages replies to the spoofed addresses, causing the victim to be unavailable to legitimate users. We simulated this attack again using the program hping3, with the same link parameters in the custom Mininet topology shown in Figure **??**. We again conducted 30 trials for each selected scenario, with each trial lasting 3 minutes. Another MUD file was generated to permit UDP traffic towards the remote UDP echo server at domain `service1.example-iot-service1.com` using any destination port. We chose to allow the use of any destination port to better simulate the UDP flooding attack, allowing traffic sent to be directed towards random ports instead of just a specific one. A correlating inbound rule was described to permit UDP responses from `service1.example-iot-service1.com` back to the MUD device.

Because UDP is a connectionless protocol, the acknowledgement of data sent and received by each party, such as in the case of TCP, is not implemented and thus poses a problem for two of the features used, *Asymmetry of Bytes* and *Asymmetry of Packets*. To make up for this, the generation of UDP traffic was achieved by using a simple Python socket program where UDP packets were sent periodically every second to the server and waited 1 second to receive a response. If no response was

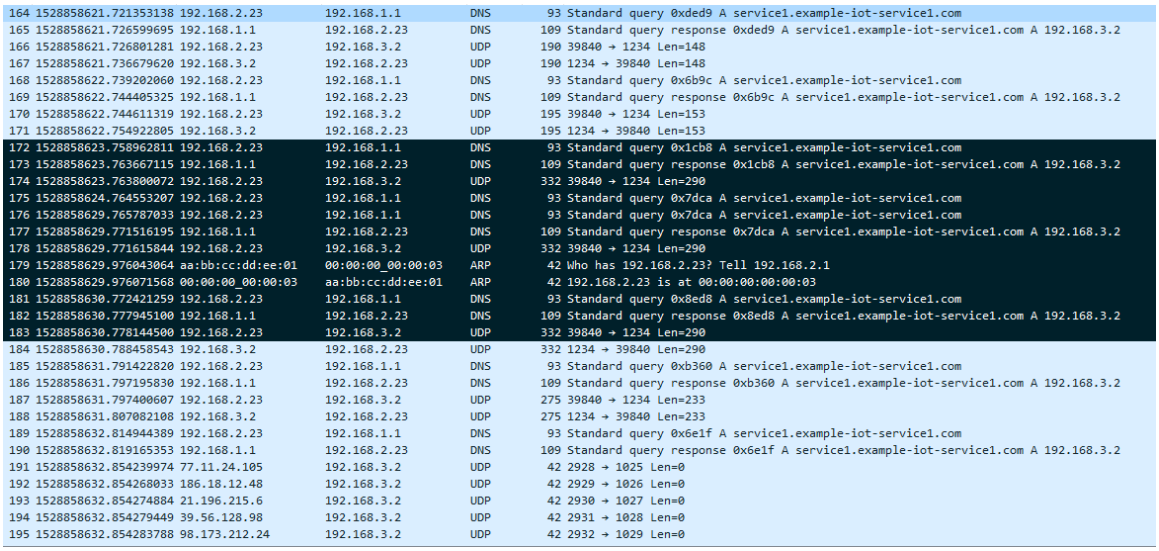
received due to packet drops either at the sending side or receiving side, the same data was transmitted again until a response was received from the server.

Results shown in Table 5.8 demonstrate that our approach sees a significantly noticeable drop in detection performance during the case of UDP Flood Attacks in comparison to our results in Section 5.3, at worst case of about 10% when the monitoring interval is set to 3 seconds and the attack is initiated at 60 seconds. When the attack is started at 30 seconds, we find that Detection Rate was best achieved when the monitoring interval was shorter, such as when it was set to 3 seconds. This is because if the monitoring interval was to be set at 5 or 10 seconds, the initial EWMA and upper/lower bounds would not been initialized as quickly and may take the attack traffic into account when computing the initial values for subsequent EWMA and bound computations. Consequently, when the attack occurred, the initial values used to compute the next iteration's EWMA would believe that the observed attack traffic was within the acceptable bounds, thus the degradation in detection when monitoring interval was increased from 3 seconds to 10 seconds. We see this phenomena flip slightly when the attack was initialized at 40 seconds, in which Detection Rate was improved with an increase of the monitoring interval from 3 to 5 seconds. However, when the interval was set to 10 seconds, the Detection Rate again degraded drastically. The approach worked best when there was significant time between observing "normal" traffic behavior of a device and attack initiation. The attack launched at 60 seconds best reflected this, as we see a steady increase in Detection Rate from 3 second monitoring to 10 second monitoring.

Table 5.8. UDP Flood Detection Results ($\alpha = 0.10$)

	3 Sec Interval		**5 Sec Interval**		**10 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**	**DR(%)**	**FP(%)**
Attack at 30 sec	92.50	7.50	80.00	0.00	0.00	5.00
Attack at 40 sec	87.50	0.00	92.11	0.00	0.00	0.00
Attack at 60 sec	90.00	1.67	94.83	1.72	100.00	0.00

We attribute the decrease in DR and increase in FPR for UDP Flooding attacks, compared to the TCP SYN Flood attack, due to the inherent unreliability of UDP and its susceptibility to packet drops. It is entirely possible that during an unlucky streak, multiple UDP packets sent outbound from the device could be dropped, causing the client to continuously send more packets than it would receive. This of course skews all four flow features such that it may occur in a false positive alert. On the other hand, if the Byte and Packet count features remain in range of the computed bounds, the lack of responses will still skew the asymmetry features. This will not result in an alert as all four features must fall out of bounds to satisfy the detection condition, but will instead bring down the average of the asymmetry features down, and thus also bring down the lower bound to values that are common during occurrences of DDoS attacks. This is exactly what had occurred in Trial 16 of our experimentation, in which a sequence of responses that were not received cause more bytes and packets to be sent than received, reducing the value of the lower bound to approximately 0. Consequently in this scenario, when the flooding attack occured and the spoofed packets sent resulted in more packets going out of a device versus packets received by the device, the 0 values of the asymmetry features were still to be considered legitimate and within the computed bounds, causing the True Negative to occur. These sequences of events leading to this case can be observed in the Wireshark traffic capture of the attacking device, H3, in Figure 5.10.



164	1528858621.721353138	192.168.2.23	192.168.1.1	DNS	93 Standard query 0xded9 A service1.example-iot-service1.com
165	1528858621.726599695	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0xded9 A service1.example-iot-service1.com A 192.168.3.2
166	1528858621.726801281	192.168.2.23	192.168.3.2	UDP	190 39840 → 1234 Len=148
167	1528858621.736679620	192.168.3.2	192.168.2.23	UDP	190 1234 → 39840 Len=148
168	1528858622.739202060	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x6b9c A service1.example-iot-service1.com
169	1528858622.744405325	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0x6b9c A service1.example-iot-service1.com A 192.168.3.2
170	1528858622.744611319	192.168.2.23	192.168.3.2	UDP	195 39840 → 1234 Len=153
171	1528858622.754922805	192.168.3.2	192.168.2.23	UDP	195 1234 → 39840 Len=153
172	1528858623.758962811	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x1cb8 A service1.example-iot-service1.com
173	1528858623.763667115	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0x1cb8 A service1.example-iot-service1.com A 192.168.3.2
174	1528858623.763800072	192.168.2.23	192.168.3.2	UDP	332 39840 → 1234 Len=290
175	1528858624.764553207	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x7dca A service1.example-iot-service1.com
176	1528858629.765787033	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x7dca A service1.example-iot-service1.com
177	1528858629.771516195	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0x7dca A service1.example-iot-service1.com A 192.168.3.2
178	1528858629.771615844	192.168.2.23	192.168.3.2	UDP	332 39840 → 1234 Len=290
179	1528858629.976043064	aa:bb:cc:dd:ee:01	00:00:00_00:00:03	ARP	42 Who has 192.168.2.23? Tell 192.168.2.1
180	1528858629.976071568	00:00:00_00:00:03	aa:bb:cc:dd:ee:01	ARP	42 192.168.2.23 is at 00:00:00:00:00:03
181	1528858630.772421259	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x8ed8 A service1.example-iot-service1.com
182	1528858630.777945100	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0x8ed8 A service1.example-iot-service1.com A 192.168.3.2
183	1528858630.778144500	192.168.2.23	192.168.3.2	UDP	332 39840 → 1234 Len=290
184	1528858630.788458543	192.168.3.2	192.168.2.23	UDP	332 1234 → 39840 Len=290
185	1528858631.791422820	192.168.2.23	192.168.1.1	DNS	93 Standard query 0xb360 A service1.example-iot-service1.com
186	1528858631.797195830	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0xb360 A service1.example-iot-service1.com A 192.168.3.2
187	1528858631.797400607	192.168.2.23	192.168.3.2	UDP	275 39840 → 1234 Len=233
188	1528858631.807082108	192.168.3.2	192.168.2.23	UDP	275 1234 → 39840 Len=233
189	1528858632.814944389	192.168.2.23	192.168.1.1	DNS	93 Standard query 0x6e1f A service1.example-iot-service1.com
190	1528858632.819165353	192.168.1.1	192.168.2.23	DNS	109 Standard query response 0x6e1f A service1.example-iot-service1.com A 192.168.3.2
191	1528858632.854239974	77.11.24.105	192.168.3.2	UDP	42 2928 → 1025 Len=0
192	1528858632.854268033	186.18.12.48	192.168.3.2	UDP	42 2929 → 1026 Len=0
193	1528858632.854274884	21.196.215.6	192.168.3.2	UDP	42 2930 → 1027 Len=0
194	1528858632.854279449	39.56.128.98	192.168.3.2	UDP	42 2931 → 1028 Len=0
195	1528858632.854283788	98.173.212.24	192.168.3.2	UDP	42 2932 → 1029 Len=0

Figure 5.10. H3 UDP Flood Wireshark Capture, Trial 16: True Negative Occurrence

These sequence of events can be summarized as follows:

- Line 172: DNS query
- Line 173: DNS response
- Line 174: UDP packet sent (Result: dropped response or request)
- Line 175: DNS request (Result: dropped response or request)
- Line 176: DNS query
- Line 177: DNS response received
- Line 178: UDP packet sent (Result: dropped response or request)
- Line 179 to 180: ARP (ARP cache timeout)
- Line 181: DNS query
- Line 182: DNS response
- Line 183: UDP packet sent (Response is received back on this attempt on Line 184)

This sequence represents the series of traffic right before the attack is initiated (seen

on Line 191), and spans approximately 7 seconds, which can be equated to two monitoring intervals in which no response was received. During these intervals, the skewed *Asymmetry of Bytes* and *Asymmetry of Packets* features resulted in the lower bound being pushed further down to 0. Consequently, the DDoS attack was not detected because the values for these two features during the attack was considered within range of the bounds. Although the *Bytes per Second* and *Packets per Second* features surge drastically from the computed upper bounds during the next observation during the occurrence of an attack, the asymmetry features are well within range, since the lower bound had been pushed to 0, and thus it lead our method into believing the traffic to have been legitimate.

We again compared the detection performance of our approach on larger network topologies of 4 and 8 networks, each network consisting of 5 hosts. The bandwidth, delay, and packet loss probability, and alpha value remained the same due to time constraints. Table 5.9 show the performance results when monitoring intervals of 3 seconds, 5 seconds, and 10 seconds were selected.

Table 5.9. UDP Flood Detection Results (Larger Topology, $\alpha = 0.10$)

	3 Sec Interval		**5 Sec Interval**		**10 Sec Interval**	
	DR(%)	**FP(%)**	**DR(%)**	**FP(%)**	**DR(%)**	**FP(%)**
4 Networks, 20 Hosts	91.30	7.50	91.47	2.50	99.15	0.00
8 Networks, 40 Hosts	96.09	2.89	99.57	5.29	100.00	7.06

Despite the results of the larger network UDP flood experiment lacking in accuracy when compared with its TCP SYN Flood counterpart, the increased accuracy as a result of increased monitoring time can be observed. The trade off again is that we sacrifice earlier detection time for increased accuracy in DR performance. At worst case, the attack is detected x seconds after its initiation, where x is the value of the set monitoring interval. Despite the removal of the lower bound constraint from

the *Bytes per Second* and *Packets per Second* features, FPR in our UDP flooding experiments remain relatively high, as seen in the scenario of 8 networks.

Overall, two major shortcomings are identified in our evaluation. The first is the relatively high rate of false positives in comparison with some other approaches. In various tests we are able to reduce the false positive occurrences by lowering α value, but it also seems that size of the network makes a significant impact on this value as well. While we try several approaches in reducing the FPR such as lowering alpha values and eliminating the lower bound constraints on *Bytes per Second* and *Packets per Second*, it is noted that more work is required to find a more consistent approach towards reducing the occurrence of false positives, especially in small surges of traffic that are not significantly higher than the computed averages and upper bounds, but still resemble the volumetric and asysmmetric behaviors of DDoS attacks. The other current shortcoming of our approach is the inability to detect application layer attacks such as HTTP flooding attacks. One of the cornerstones of our approach is the ability to find deviations from normal expected behavior through the measurement of flow features at each observation. However, HTTP attacks oftentimes do not engage in flooding that is entirely obvious and oftentimes the attack traffic is very hard to distinguish from normal HTTP traffic [14]. It can be envisioned however, that our approach can still play a role in detecting these kinds of attacks through first flagging suspected attacking nodes and then forwarding it to the SDN controller to be run through IDS or IPS application as a second step.

Chapter 6

CONCLUSION

In this thesis, we examined how the Manufacturer Usage Descriptions (MUD) proposed standard implemented in an Software Defined Networking (SDN) context can be leveraged to detect DDoS flooding attacks that are launched from IoT sources through periodic observations of the device flows, in order to detect any unexpected deviations. We generated flow rules that were translated from the MUD files that detailed the permitted traffic patterns and behaviors, and used these flow rules as a means of identifying the traffic coming in and out of the associated device. We then used the centralized capabilities of SDN to periodically monitor all the MUD flows installed on each connected OpenFlow switch of each IoT network that is managed by the SDN controller. The Exponentially Weighted Moving Average (EWMA) function was utilized to take into account slight deviations of past and future traffic, along with being used to generate an upper and lower bound to define an acceptable range of observed flow features. Detection of attacks sought to identify the presence of the volumetric and asymmetric characteristics of DDoS flooding attacks through observing large deviations in *Bytes Per Second*, *Packets Per Second*, *Asymmetry of Bytes*, and *Asymmetry of Packets* from their respective EWMAs. Our approach serves as a simple method of detecting common flooding attacks such as TCP flooding and UDP flooding (albeit with a bit more difficulty) from IoT sources. We demonstrate that given sufficient time between attack and the initialization of the first EWMA computation, we are able to achieve a high detection rate. While the values of False Positive Rate is a factor to be improved upon, for traffic that follow an expected be-

havior (as defined by their respective MUD rules), we are able to consistently achieve above 98% detection rate performance for TCP SYN attacks while a 90% detection rate performance for UDP attacks. We are able to improve overall accuracy of the approach by setting the monitoring interval to a larger value, the trade off being the delay in detection time. The improvement in accuracy can be demonstrated when the interval times for the UDP attacks are increased to 10 seconds, increasing the detection rate from 90% to 100%.

REFERENCES

[1] A. Mosenia and N. K. Jha, "A Comprehensive Study of Security of Internet-of-Things," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, Oct. 2017.

[2] G. Kambourakis, C. Kolias, and A. Stavrou, "The Mirai Botnet and the IoT Zombie Armies," in *Proc. MILCOM 2017 - 2017 IEEE Military Communications Conf. (MILCOM)*, Oct. 2017, pp. 267–272.

[3] M. E. Ahmed and H. Kim, "DDoS Attack Mitigation in Internet of Things Using Software Defined Networking," in *Proc. IEEE Third Int. Conf. Big Data Computing Service and Applications (BigDataService)*, Apr. 2017, pp. 271–276.

[4] E. Lear and B. Weis, "Slinging MUD: Manufacturer usage descriptions: How the network can protect things," in *Proc. Int. Conf. Selected Topics in Mobile Wireless Networking (MoWNeT)*, Apr. 2016, pp. 1–6.

[5] Intel, *A Guide To the Internet of Things*, Accessed: 2018-03-23. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/images/iot/guide-to-iot-infographic.png.

[6] M. Özçelik, N. Chalabianloo, and G. Gür, "Software-Defined Edge Defense Against IoT-Based DDoS," in *Proc. IEEE Int. Conf. Computer and Information Technology (CIT)*, Aug. 2017, pp. 308–313.

[7] B. Krebs, *Who Makes the IoT Things Under Attack?* Accessed: 2018-05-22, Oct. 2016. [Online]. Available: https://krebsonsecurity.com/2016/10/who-makes-the-iot-things-under-attack/.

[8] ——, *Who is Anna-Senpai, the Mirai Worm Author?* Accessed: 2018-03-23, Jan. 2018. [Online]. Available: https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/.

[9] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," *Computer*, vol. 50, no. 7, pp. 80–84, Jul. 2017.

[10] S. Hilton, *Dyn Analysis Summary Of Friday October 21 Attack*, Accessed: 2018-03-23, Oct. 2016. [Online]. Available: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[11] E. Lear, R. Droms, and D. Romascanu, "Manufacturer Usage Description Specification," IETF Secretariat, Internet-Draft (work in progress) draft-ietf-opsawg-

mud-22, May 2018, https://tools.ietf.org/html/draft-ietf-opsawg-mud-22. [Online]. Available: https://tools.ietf.org/html/draft-ietf-opsawg-mud-22.

[12] NIST, *Software Defined Virtual Networks*, Accessed: 2018-03-07, Oct. 2017. [Online]. Available: https://www.nist.gov/programs-projects/software-defined-virtual-networks.

[13] J. Mirkovic and P. Reiher, "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, Apr. 2004, ISSN: 0146-4833. [Online]. Available: http://doi.acm.org/10.1145/997150.997156.

[14] S. T. Zargar, J. Joshi, and D. Tipper, "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks," *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013, ISSN: 1553-877X.

[15] H. Sinanović and S. Mrdovic, "Analysis of Mirai malicious software," in *Proc. Telecommunications and Computer Networks (SoftCOM) 2017 25th Int. Conf. Software*, 2017, pp. 1–5.

[16] T. Degroote, *WATER TORTURE: A SLOW DRIP DNS DDOS ATTACK*, Accessed:2018-05-06, Feb. 2014. [Online]. Available: https://secure64.com/water-torture-slow-drip-dns-ddos-attack/.

[17] M. Kumar, *New Mirai Okiru Botnet targets devices running widely-used ARC Processors*, Accessed: 2018-05-06, Jan. 2018. [Online]. Available: https://thehackernews.com/2018/01/mirai-okiru-arc-botnet.html.

[18] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015, ISSN: 0018-9219.

[19] Q. Yan, F. R. Yu, Q. Gong, and J. Li, "Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 602–622, 2016, ISSN: 1553-877X.

[20] K. Kalkan and S. Zeadally, "Securing Internet of Things (IoT) with Software Defined Networking (SDN)," *IEEE Communications Magazine*, pp. 1–7, 2017, ISSN: 0163-6804.

[21] F. Hu, Q. Hao, and K. Bao, "A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation," *IEEE Communications Surveys Tutorials*, vol. 16, no. 4, pp. 2181–2206, 2014, ISSN: 1553-877X.

[22] M. Jethanandani, L. Huang, S. Agarwal, and D. Blair, "Network Access Control List (ACL) YANG Data Model," IETF Secretariat, Internet-Draft draft-ietf-netmod-acl-model-18, Mar. 2018, http://www.ietf.org/internet-drafts/draft-ietf-netmod-acl-model-18.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-netmod-acl-model-18.txt.

[23] N. Feamster, "Outsourcing Home Network Security," in *Proceedings of the 2010 ACM SIGCOMM Workshop on Home Networks*, ser. HomeNets '10, New Delhi, India: ACM, 2010, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/1851307.1851317.

[24] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting Traffic Anomaly Detection Using Software Defined Networking," in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2011, pp. 161–180.

[25] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," in *Proc. IEEE Local Computer Network Conf*, Oct. 2010, pp. 408–415.

[26] X. Yang, B. Han, Z. Sun, and J. Huang, "SDN-Based DDoS Attack Detection with Cross-Plane Collaboration and Lightweight Flow Monitoring," in *Proc. GLOBECOM 2017 - 2017 IEEE Global Communications Conf*, 2017, pp. 1–6.

[27] S. Lim, J. Ha, H. Kim, Y. Kim, and S. Yang, "A SDN-oriented DDoS blocking scheme for botnet-based attacks," in *Proc. Sixth Int. Conf. Ubiquitous and Future Networks (ICUFN)*, Jul. 2014, pp. 63–68.

[28] C. Aggarwal and K. Srivastava, "Securing IOT devices using SDN and edge computing," in *Proc. 2nd Int. Conf. Next Generation Computing Technologies (NGCT)*, Oct. 2016, pp. 877–882.

[29] P. Bull, R. Austin, E. Popov, M. Sharma, and R. Watson, "Flow Based Security for IoT Devices Using an SDN Gateway," in *Proc. IEEE 4th Int. Conf. Future Internet of Things and Cloud (FiCloud)*, Aug. 2016, pp. 157–163.

[30] NIST, *NIST/SEMATECH e-Handbook of Statistical Methods*, Accessed: 2018-03-24, Oct. 2013. [Online]. Available: http://www.itl.nist.gov/div898/handbook/.

[31] T. Finch, "Incremental calculation of weighted mean and variance," University of Cambridge, Tech. Rep., Feb. 2009.

[32] B. Lantz, B. Heller, N. Handigol, and V. Jeyakumar, *Mininet*, Accessed: 2018-05-28, Apr. 2018. [Online]. Available: https://github.com/mininet/mininet.

[33] NTT Labs, *Ryu*, Accessed: 2018-05-28, Jun. 2018. [Online]. Available: https://github.com/osrg/ryu.

[34] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, *Open vSwitch*, Accessed: 2018-05-28, Jun. 2018. [Online]. Available: https://github.com/openvswitch/ovs.

[35] E. Poger, T. Lemon, and Internet Systems Consortium, *Dhclient*, Accessed: 2018-05-28, Mar. 2018. [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man8/dhclient.8.html.

[36] Internet Systems Consortium, *Dhclient.conf*, Accessed: 2018-05-28, Mar. 2018. [Online]. Available: http://manpages.ubuntu.com/manpages/xenial/man5/dhclient.conf.5.html.

[37] C. Gibson, K. Katterjohn, Mixter, and Fyodor, *Ncat*, Accessed: 2018-05-28, Apr. 2018. [Online]. Available: https://github.com/nmap/nmap/tree/master/ncat.

[38] N. Tallim, *StaticDHCPd*, Accessed: 2018-05-28, Dec. 2017. [Online]. Available: https://github.com/flan/staticdhcpd.

[39] S. Kelley, *Dnsmasq*, Accessed: 2018-05-28, Oct. 2017. [Online]. Available: http://www.thekelleys.org.uk/dnsmasq/doc.html.

[40] R. Klöti, V. Kotronis, and P. Smith, "Openflow: A security analysis," in *Proc. 21st IEEE Int. Conf. Network Protocols (ICNP)*, Oct. 2013, pp. 1–6.

[41] S. Sanfilippo, *Hping3*, Accessed: 2018-06-14, Feb. 2014. [Online]. Available: https://tools.kali.org/information-gathering/hping3.

APPENDIX A

GENERATED MUD FILES

```
{
  "ietf-mud:mud": {
    "mud-version": 1,
    "mud-url": "https://www.example-mud-server.com/toaster/v
       1/mudfile.json",
    "last-update": "2018-04-29T22:06:59+02:00",
    "mud-signature": "https://www.example-mud-server.com/
       toaster/v1/mudfile.json.sig",
    "cache-validity": 48,
    "is-supported": true,
    "systeminfo": "IoTCompany toasterv1",
    "mfg-name": "IoTCompany",
    "model-name": "toasterv1",
    "from-device-policy": {
      "access-lists": {
        "access-list": [
          {
            "name": "mud-92922-v4fr"
          }
        ]
      }
    },
    "to-device-policy": {
      "access-lists": {
        "access-list": [
          {
            "name": "mud-92922-v4to"
          }
        ]
      }
    }
  },
  "ietf-access-control-list:access-lists": {
    "acl": [
      {
        "name": "mud-92922-v4to",
        "type": "ipv4-acl-type",
        "aces": {
          "ace": [
```

```
            {
              "name": "cl0-todev",
              "matches": {
                "ipv4": {
                  "ietf-acldns:src-dnsname": "service1.
                     example-iot-service1.com",
                  "protocol": 6
                },
                "tcp": {
                  "source-port": {
                    "operator": "eq",
                    "port": 1234
                  }
                }
              },
              "actions": {
                "forwarding": "accept"
              }
            }
          ]
        }
      },
      {
        "name": "mud-92922-v4fr",
        "type": "ipv4-acl-type",
        "aces": {
          "ace": [
            {
              "name": "cl0-frdev",
              "matches": {
                "ipv4": {
                  "ietf-acldns:dst-dnsname": "service1.
                     example-iot-service1.com",
                  "protocol": 6
                },
                "tcp": {
                  "destination-port": {
                    "operator": "eq",
                    "port": 1234
                  }
                }
```

```
                },
                "actions": {
                  "forwarding": "accept"
                }
              }
            ]
          }
        }
      ]
    }
}
```

Listing A.1. Generated TCP MUD File

```
{
  "ietf-mud:mud": {
    "mud-version": 1,
    "mud-url": "https://www.example-mud-server.com/toaster/v
       1/mudfile.json",
    "last-update": "2018-04-29T22:06:59+02:00",
    "mud-signature": "https://www.example-mud-server.com/
       toaster/v1/mudfile.json.sig",
    "cache-validity": 48,
    "is-supported": true,
    "systeminfo": "IoTCompany toasterv1",
    "mfg-name": "IoTCompany",
    "model-name": "toasterv1",
    "from-device-policy": {
      "access-lists": {
        "access-list": [
          {
            "name": "mud-92922-v4fr"
          }
        ]
      }
    },
    "to-device-policy": {
      "access-lists": {
        "access-list": [
          {
            "name": "mud-92922-v4to"
          }
```

```
        ]
      }
    }
  },
  "ietf-access-control-list:access-lists": {
    "acl": [
      {
        "name": "mud-92922-v4to",
        "type": "ipv4-acl-type",
        "aces": {
          "ace": [
            {
              "name": "cl0-todev",
              "matches": {
                "ipv4": {
                  "ietf-acldns:src-dnsname": "service1.
                     example-iot-service1.com",
                  "protocol": 6
                },
                "tcp": {
                  "source-port": {
                    "operator": "eq",
                    "port": 1234
                  }
                }
              },
              "actions": {
                "forwarding": "accept"
              }
            }
          ]
        }
      },
      {
        "name": "mud-92922-v4fr",
        "type": "ipv4-acl-type",
        "aces": {
          "ace": [
            {
              "name": "cl0-frdev",
              "matches": {
```

```
                    "ipv4": {
                      "ietf-acldns:dst-dnsname": "service1.
                         example-iot-service1.com",
                      "protocol": 6
                    },
                    "tcp": {
                      "destination-port": {
                        "operator": "eq",
                        "port": 1234
                      }
                    }
                  },
                  "actions": {
                    "forwarding": "accept"
                  }
                }
              ]
            }
          }
        ]
    }
}
```

Listing A.2. Generated UDP MUD File

APPENDIX B

CONFIGURATION FILES

```
# Configuration file for /sbin/dhclient.
#
# This is a sample configuration file for dhclient. See dhclient.conf's
#       man page for more information about the syntax of this file
#       and a more comprehensive list of the parameters understood by
#       dhclient.
#
# Normally, if the DHCP server provides reasonable information and does
#       not leave anything out (like the domain name, for example), then
#       few changes must be made to this file, if any.
#

option rfc3442-classless-static-routes code 121 = array of unsigned integer 8;

### MUD Options Added ####

option option-mud-url-v4 code 161 = text;
send option-mud-url-v4 = "https://www.example-mud-server.com/toaster/v1/mudfile.json";
#send option-mud-url-v4 = "https://www.example-mud-server.com/toaster/v2/mudfile.json";

############################


send host-name = gethostname();
request subnet-mask, broadcast-address, time-offset, routers,
        domain-name, domain-name-servers, domain-search, host-name,
        dhcp6.name-servers, dhcp6.domain-search, dhcp6.fqdn, dhcp6.sntp-servers,
        netbios-name-servers, netbios-scope, interface-mtu,
        rfc3442-classless-static-routes, ntp-servers;
```

Listing B.1. Modified `dhclient.conf` for MUD Support