APPROVAL SHEET

Title of Thesis: Detecting DDoS Attacks in Software Defined Networks: An Experimental Study of Stream Sampling Methods

Name of Candidate: David Chandler Harris, Jr.
Master of Science, Computer Science, 2017

Thesis and Abstract Approved: Alan T. Sherman

Alan T. Sherman
Professor
Department of Computer Science and Electrical Engineering

Date Approved: April 25, 2017 4-25-17

ABSTRACT

Title of Thesis: Detecting DDoS Attacks in Software Defined Networks: An Experimental Study of Stream Sampling Methods

David Chandler Harris, Jr., Master of Computer Science, 2017

Thesis directed by: Alan T. Sherman, Professor
Department of Computer Science and
Electrical Engineering

I propose and experimentally evaluate a new sampling method for a streaming algorithm to improve Distributed Denial of Service (DDOS) detection in Software Defined Networks (SDNs). My method leverages the SDN architecture of OpenFlow and its novel capabilities to improve detection by analyzing traffic by flow. This approach can lower the cost of gathering data for analysis and improve the detection rate. Using the Mininet emulation environment, I compare the new sampling methods using my adaption of the hierarchical heavy hitter algorithm in a SDN environment and analyze the differences to a possible implementation on a legacy network. My work shows that clear differences can be detected by using per flow sampling to detect hierarchical heavy hitters from traffic that contains heavy flows.

Detecting DDoS Attacks in Software Defined Networks: An Experimental Study of Stream Sampling Methods

by

David Chandler Harris Jr.

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
2017

© Copyright David Chandler Harris, Jr. 2017

I would like to dedicate this work to my family who has supported me every step of the way. Thank you Dad, Mom, and Elana.

ACKNOWLEDGMENTS

I owe my gratitude to all the individuals who have guided and mentored me through the completion of my thesis. I would first like to thank my advisor Dr. Alan Sherman for assisting me in finding an excellent topic to study and research. I would also like to thank Dr. Matthew Carey from the Laboratory for Telecommunication Science for agreeing to be a member of my committee and also offering advice and guidance during my research efforts. Dr. Carey took many hours out of his schedule to visit UMBC and assist in the formulation of this work, and for that I am truly grateful. I would also like to thank the other members of my committee Dr. Anupam Joshi and Dr. Dhanajay Phatak for their support of my research.

I would like to thank the members of the Bridge to the Doctorate program at UMBC for their fellowship and support as I completed this process. I would especially like to express my gratitude to Dr. Renetta Tull who as the coordinator for the program was instrumental in not only my success at UMBC but also in my interest in the university as well.

Finally, I would like to extend a large thank you to the entire CSEE department as a whole, to all the professors whose classes I attended and to the students with whom I was able to study and learn. You have made my student experience truly enjoyable, and I could not have succeeded without your support, advice, and humor.

There have been so many people who have worked to ensure my success, so if I have not mentioned you above please know that your efforts are appreciated and will be remembered.

TABLE OF CONTENTS

DEDICATION . . . . **ii**

ACKNOWLEDGMENTS . . . . **iii**

LIST OF TABLES . . . . **vi**

LIST OF FIGURES . . . . **vii**

Chapter 1 INTRODUCTION **1**

1.1 Overview 1

1.2 Thesis Statement 2

Chapter 2 BACKGROUND **3**

2.1 Software Defined Networks 3

2.2 Distributed Denial of Service Attacks 5

2.3 Hierarchical Heavy Hitter Algorithm 6

Chapter 3 PREVIOUS WORK **8**

Chapter 4 METHODS **12**

4.1 Mininet 12

4.2 Topology 12
4.3 Traffic Generation 14
4.4 Implementation 15

Chapter 5 RESULTS **17**

Chapter 6 DISCUSSION **21**

6.1 Future Questions 23

Chapter 7 CONCLUSION **25**

Appendix A SOURCE CODE **26**

A.1 Hierarchical Heavy Hitter Algorithm for raw statistics 26
A.2 Hierarchical Heavy Hitter Algorithm for stats grouped by flow 32
A.3 Controller for raw stats 39
A.4 Controller for flow grouping 44
A.5 Targeted heavy flow traffic generation 49
A.6 Spread heavy flow traffic generation 58

REFERENCES **67**

LIST OF TABLES

4.1 List of four experiments 16

LIST OF FIGURES

2.1 SDN Architecture diagram 4

2.2 HHH Example 7

4.1 The experimental SDN topology 13

5.1 SDN setups under targeted heavy flows 18

5.2 SDN setups under spread heavy flows 20

Chapter 1

INTRODUCTION

1.1 Overview

Software Defined Networks(SDNs) present innumerable opportunities to create efficient techniques to address many different networking issues. Distributed Denial of Service(DDoS) attacks on networks are a common security problem affecting network architecture today. This thesis is built on addressing this issue by determining if the inherent abilities provided by SDNs can increase a network's ability to detect DDOS attacks, which could lead to improved mitigation of such attacks. Improved methods for detection and mitigation are still under study, but the system-wide scope that can be analyzed by a controller has shown promise. One of the primary methods is the use of advanced algorithms that can quickly classify and analyze incoming data and structure the network software to route the data optimally based on that analysis. Dr. Kalliola described the use of this method in a paper and showed promising results (Kalliola *et al.* 2015). This work seeks to show the effectiveness of such analysis by comparing the methods used to sample incoming data for the analysis algorithms, through leveraging the unique communication abilities inherent in SDNs. I conducted a simulation of normal and heavy traffic and used two different sampling methods. The relative effectiveness of each technique was determined by analyzing the simulation results. I found that when using a SDN protocol's inbuilt characteristics,

I was better able to detect and identify the instances of larger-than-average traffic flows entering my test network efficiently.

1.2 Thesis Statement

The goal of this thesis is to determine if the use of sampling methods that leverage the flexibility of SDNs can improve the heavy flow detection abilities of streaming algorithms. To answer that question, I ran simulations using an emulated SDN test system to analyze its capacity to identify heavy flows out of incoming traffic. The ability to detect heavy flows can be applied to the process of detecting when DDOS attacks are being conducted on a network. The paradigm of SDNs brings some key advantages to solving this problem. One key I focused on exploiting in my research was the ability of SDNs to relocate where decisions are made on where data should be routed. The shift of this decision making process from the switch to the controller, allows lower overhead for the switch and more complex analysis on the controller. Another key is that as a result of the communication paths between controller and switch, anomalous traffic detected at any switch can be targeted by the system for more thorough analysis. Finally, the system wide view the controller has of the network can allow for patterns to be discovered that might not be identifiable by a single switch.

Chapter 2

BACKGROUND

2.1 Software Defined Networks

Software Defined Networks (SDNs) present an emerging and growing paradigm in computer networking. This is due to the foundational characteristics of SDNs being flexible, maintainable, and adaptable. Software defined networks are built from the concept of separating the data and control plane in the network architecture. This separation translates to having a logically centralized controller that can be programmed to direct and then redirect the forwarding of data that are passed through the network's switches.

This design allows the control plane to be programmed and thus adapt and change the underlying data movements quickly and efficiently. The commands for changes in the network, flow down from the controller to the switches on the lower level. For example, Figure 2.1 shows such a configuration referred to as a southbound interface. The specific protocol used to implement the software defined paradigm for my network is OpenFlow, while the chosen framework for my network was Ryu. The OpenFlow protocol is one of the main SDN protocols in use because of its ability to organize data by flows. For my research a flow is defined to be any sequence of packets that can be matched to rules in the flow table (Afek *et al.* 2017). In Openflow, a switch consists of one or more flow tables and a group table, these tables perform packet lookups and forwarding. There is also an

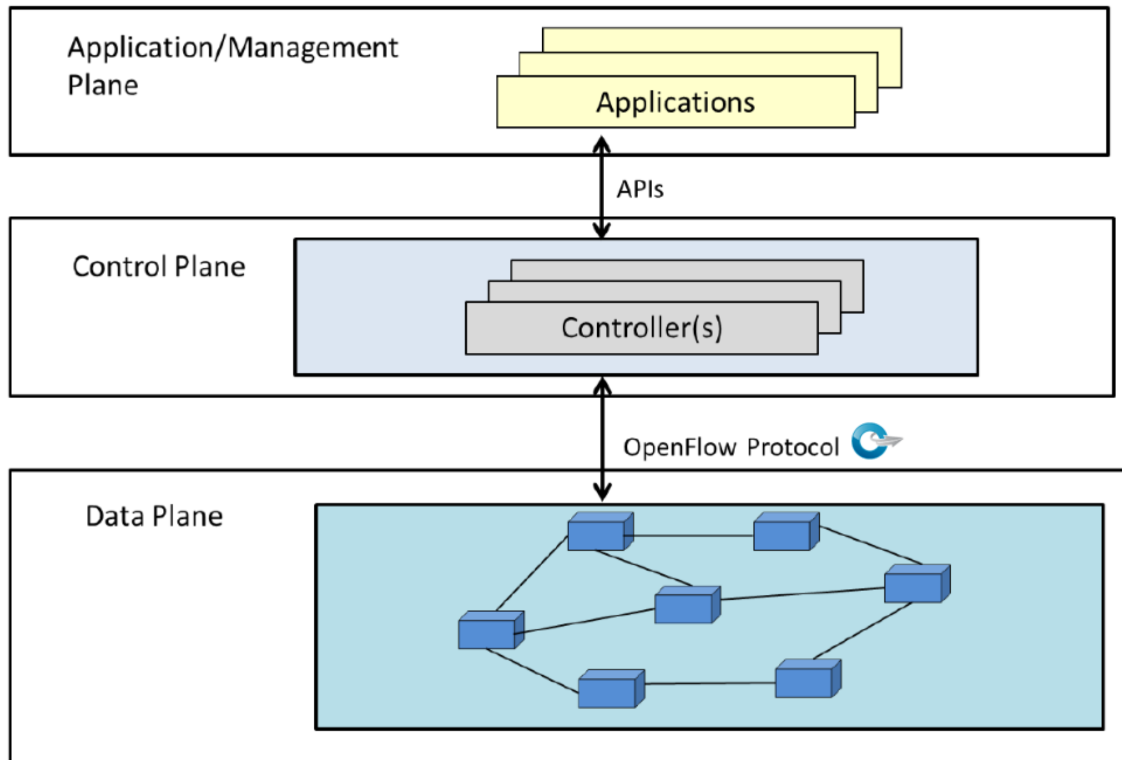

FIG. 2.1. The accepted architecture for software defined networks is pictured above. The separation of the control and data plane is the key aspect of this setup. OpenFlow, is one of the protocols that can be used to facilitate communication between the separated layers.

OpenFlow channel on each switch to provide communication with an external controller. The flow entries in the table can be added, updated, and removed based on instructions from the controller. The controller's instructions for each specific flow are stored in a flow table on each switch along with the match fields/identifiers and counter statistics for that flow. This thesis attempts to demonstrate how using this technology to combat DDOS attacks on virtualized infrastructure holds promise.

2.2 Distributed Denial of Service Attacks

Alongside the increasing growth and promise of web based services and applications, there is increasing growth in cyber attacks on these services. One of the most frequent attack types is Distributed Denial of Service (DDOS) attacks. DDOS attacks are implemented by flooding the connections of a particular web service/site with large amounts of fake traffic and/or bad messages. There can be several motivations for these attacks, including denying service and system access to valid users and processes. Another motivation is simply making the service/site unsustainable through increased costs from having to meet the demand the attacks impose on the servers (Yan *et al.* 2016). This goal is increasingly more common and effective because of the rise of virtualization in many internet applications. Studies have shown that DDOS attacks on virtualized and cloud servers are 15 percent more effective than on traditional hardware server environments (Yan *et al.* 2016). As a result, there is a pressing need for mitigation techniques, especially when those attacks are on virtualized/cloud hosted web services and sites.

There are many different branches of DDOS attack methods available to adversaries for potential attacks on network systems. The most common and simply implemented method of attack is flooding attacks from multiple sources. These attacks consist of data being continuously sent to the network's servers from different infected origin points

(Mirkovic & Reiher 2004).

2.3 Hierarchical Heavy Hitter Algorithm

The streaming algorithm that will be implemented in my system is one that will detect Hierarchical Heavy Hitters (HHH). The Heavy Hitter problem is one that is well studied, for my research we used the definition of a heavy hitter as an entity which accounts for at least a set proportion of the total network activity (Zhang *et al.* 2004). This activity, depending on the situation can be measured in terms of number of packets, bytes, or other attribute. The identification of HHHs is a two part process.

The first part revolves around the identification of heavy hitter traffic. As traffic flows through the system, the algorithm will analyse it based on specified features and will grow a trie or prefix tree based on which data features are the sources of a majority of the traffic. After the trie is constructed, all nodes that are above a certain threshold will be identified as heavy hitters. The second part of locating HHHs involves traversing through all the heavy hitters previously identified in the trie and determining if they are still heavy hitters after subtracting any HHHs that are among their child nodes.

An example of the process for identifying a tree's HHHs is pictured in Figure 2.2. Nodes above the threshold of 10 are double circled to denote their status as Heavy hitters. Heavy Hitters are then colored blue if they are HHH. Although there are 7 nodes pictured that are heavy hitters, only 4 are HHHs including the root node. This is due to the HHH algorithm being able to clarify its results to the level of detail needed to identify the true sources of heavy flows in the system. Thus the use of this particular algorithm for heavy flow detection is well suited (Jose, Yu, & Rexford 2011).

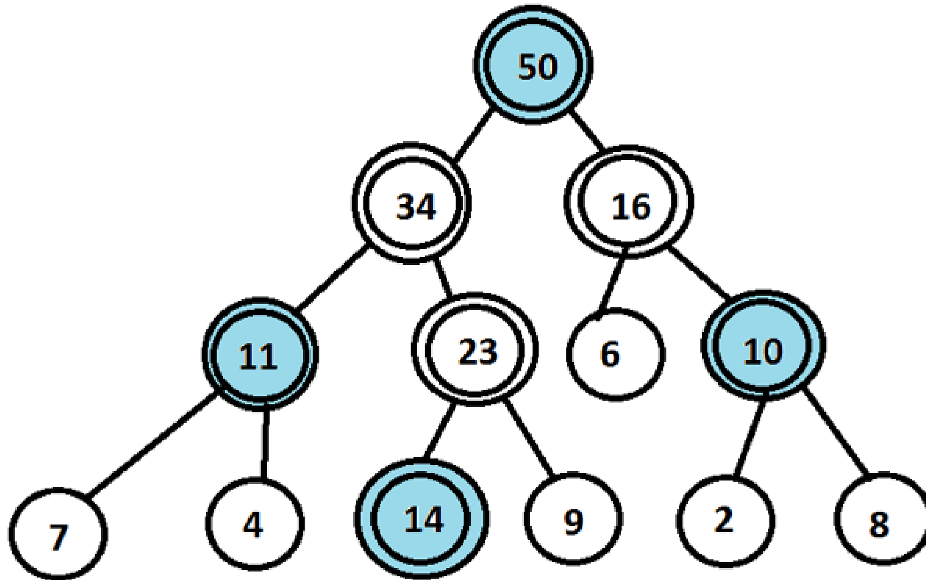

FIG. 2.2. An example of a tree identifying its Hierarchical Heavy Hitters(HHHs). The specified threshold for status as a heavy hitter in this example is 10. Thus all nodes above 10 are double circled to denote their status as heavy hitters. Heavy Hitters that retain a sum of 10 after subtracting the sum of the HHHs from their children if any are shaded blue. Those Heavy Hitters shaded blue can be classified as HHHs.

Chapter 3

PREVIOUS WORK

Previously, there have been multiple research efforts to determine the effectiveness of the unique capabilities of SDNs to detect and mitigate denial of service attacks on systems. Additionally, there have been various approaches developed to address this problem as well. Much of the previous work used setups that leveraged additional switch capabilities to assist in the detection of heavy flows and heavy hitters. However, these approaches did not try to take advantage of the capabilities provided by SDNs, but instead based their implementations on custom hardware or software extensions.

Researchers Afek and Bremler-Barr (Afek *et al.* 2017) worked to define an effective way to sample in an SDN Match and Action model, which is provided by OpenFlow and P4, another SDN protocol. The researchers presented multiple approaches and analyzed their efficacy. This research approach differs in that their simulation did not attempt to measure the performance and accuracy gains acquired from leveraging the capabilities of the SDN. The primary method they used is having the controller insert a rule into the switch to monitor sources of heavy flows more closely with an exact packet count instead of sampling the packets as they go through.

An additional approach can be found in (Giotis *et al.* 2014), which is based on their theory that "OpenFlow's statistics collection and processing overloads the centralized con-

trol plane and introduces scalability issues.” As a result, they decided to enhance and decouple this functionality by using the Sflow tool. They compared the results of this implementation against native SDN approaches that use the inbuilt sampling capabilities. The methods they used to detect anomalies and outlying flows in their traffic were entropy-based calculations developed by Shah and Nucci (Ranjan *et al.* 2007). This method was in contrast to many other methods that made use of the identification of heavy hitting flows as the key to mitigating bad traffic in SDNs.

Shirali-Shahreza and Ganjali (Shirali-Shahreza & Ganjali 2013) improved the sampling capabilities of SDNs while alleviating the demands on inter-network communication paths. This work has shown some promising results. Similar to (Giotis *et al.* 2014), this work also centered on leveraging an external tool to improve the traffic sampling capabilities of a SDN. The rationale behind the attempt in their paper, however is that many network security tools rely on packet-level information in their analysis. Thus OpenFlow, which focuses on flows of traffic rather than on individual packets, might not be able to serve the input needs of these tools adequately. Their system takes advantage of OpenFlow’s built in counters for each flow, such as the received packets counter, to select packets at a specified rate and then forward them to the controller for further analysis.

Additional research has been conducted to improve detection and provide mitigation. Kalliola and Lee (Kalliola *et al.* 2015) focus on detecting DDOS attacks in a SDN environment and mitigating the effects of the attack. For the detection aspect of their system, they made use of the HHH algorithm to cluster incoming data using the source address attribute. The researchers noted that even if source addresses were spoofed, most of the spoofed addresses would not fall into the set of addresses responsible for expected traffic. Thus, the ability to identify bad traffic would be retained. The researchers used multiple different DDOS attack methods to test their system. This work’s experimental results revolved around the application of different mitigation techniques to their test system. The

work does not focus on how to improve the detection methods of their network through OpenFlow's capabilities. They instead leverage software defined networking's adaptability and flexibility to autonomously mitigate bad traffic.

In another work, Yang describes and implements a novel system for detecting heavy hitters in SDNs (Yang, Ng, & Seah 2016). Their research is centered on studying the effectiveness, accuracy, and other aspects of their heavy hitter detection methodology for SDNs. Their work combines the information gained from switch statistics and forwarding table entries to gain a broad and network wide view of the system from the controller. The authors describe the implementation of their system in two parts. The first part involves the collection of aggregate and individual flow statistics from the switches. The next part involves identifying glaring characteristics that can be used to assemble new forwarding table entries that will monitor the found suspicious flows to verify if they are really heavy flows. This two-step process is an effective and novel method to take advantage of a SDN's flexibility to improve detection capabilities. This work's approach to identifying heavy hitters discusses its speed in relation to other heavy hitter identification setups. However it does not discuss the overhead and effectiveness gained from the approach compared to an SDN that does not take advantage of its OpenFlow features or a even a traditional network.

In the final research effort analyzed in this paper, Jose focuses on using the OpenFlow protocol in a software defined network to detect hierarchical heavy hitters (Jose, Yu, & Rexford 2011). This paper focuses on simply using the inbuilt capabilities of the OpenFlow protocol on what can be seen as generic hardware with no custom specifications. This work focuses on finding the most effective implementation of the HHH algorithm on the test setup. They experiment with altering the sampling rate to study the relationship between overhead and accuracy. They also use the threshold for heavy hitter designation as an experimental feature to determine what percentage is most effective. In contrast to this thesis, the accuracy and efficiency gained from implementing this setup are not differentiated

across legacy network and SDN architectures.

Chapter 4

METHODS

4.1 Mininet

Mininet is an emulation environment that allows for easy creation of a realistic virtual network, running real kernel, switch, and application code. This emulation can be performed on a single machine whether it's virtual, cloud, or native. This paper uses Mininet for its built-in abilities to emulate SDN topologies, traffic handling, and applications inexpensively and efficiently (Lantz, Heller, & McKeown 2010). This application has the added advantage of being able to simulate traffic generation at different stochastic rates compared to alternative traffic generating tools.

4.2 Topology

The organization of the SDN used in the experiment will be based on a simple simulation of how a network would be organized in industry. Figure 4.1 illustrates the setup of 10 hosts on each side of two switches connected to a controller. The rationale behind this topology was to ensure that the number of sources for detected heavy hitters would be able to match the number of paths that could be over the threshold. One of the switches will deliver a summary of their traffic data statistics periodically to the controller where the streaming algorithm will be running.

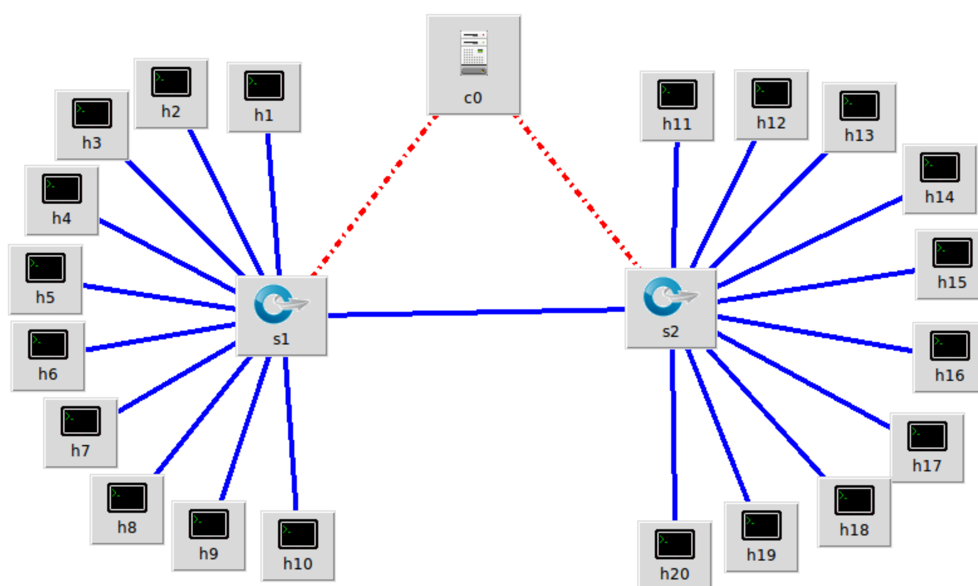

FIG. 4.1. Network topology used for the SDN traffic simulations. There were 10 hosts on each side, with hosts 1-10 transmitting traffic to hosts 11-20. Switches s1 and s2 were connected to a controller c0 and reported their traffic statistics every set time period.

4.3 Traffic Generation

To simulate a software defined network undergoing a denial of service attack, the first step is to simulate regular traffic coming into the system. I used TCP traffic resembling the profile of normal web browsing for the average flows in the experiment's simulations (Gamer & Mayer 2011). The tool used to simulate the traffic in Mininet is the Distributed Internet Traffic Generator (D-ITG). D-ITG is "a platform capable of producing traffic at packet level, accurately replicating appropriate stochastic processes for both IDT (Inter Departure Time) and PS (Packet Size) random variables" (Botta, Dainotti, & Pescapè 2012). The profile for normal web traffic involved the transmitting hosts sending 1 packet every second with each packet size being 200 bytes. The profile for DDOS traffic involved the same protocol but has a higher frequency of transmitted packets. For many DDOS traffic attacks, the protocol usually used is UDP (Mirkovic & Reiher 2004). However, to avoid the regressive case of simply filtering traffic by protocol for detection purposes, I used the same protocol for both traffic types. The goal of the systems' detection was showing that it could detect the differences between a set level of normal traffic and the same traffic that contained sources with heavier flows. The traffic tests were split into two cases, in the first, the traffic flowed on a one-to-one path where there will be 10 data paths in total. The traffic generated for this pattern was 4 MB per second before the heavy flows began and 20 MB per second with the heavy flows. This traffic was denoted as targeted heavy flows. The next part of the tests featured one-to-many flows of traffic where each transmitting host will send traffic to each receiving host for a total of 100 data paths. The traffic generated for this pattern was 40 MB per second before the heavy flows began and 104 MB per second with the heavy flows. This set was denoted as spread heavy flows.

4.4 Implementation

I ran the SDN using Mininet in a virtual system. The traffic used by the sampling methods was generated from D-ITG over two time periods of equal length. The traffic will run for 150,000 milliseconds during the first period. During the second time period there was a mixture of regular web traffic and simulated DDOS traffic from multiple hosts. Three of the transmitting hosts will become malicious and begin transmitting heavy flows to their targets. There will be two sets of generated traffic run on each SDN setup. This traffic was analyzed using the selected streaming algorithm to detect HHHs. The algorithm will be based on tries tailored to the data provided by the OpenFlow switches. The analyzed attribute for the traffic will be the destination address for each detected flow. Using the OpenFlow interface provided by the Ryu controller, the relevant data can be quickly retrieved from the sampled messages with the controller's stats request message. This process will give the algorithm sufficient information to detect the set of hierarchical heavy hitters in the traffic set. The threshold that will be used to determine if a node is a heavy hitter will be ten percent. Jose in his work determined this threshold to be an effective setting for such traffic simulations (Jose, Yu, & Rexford 2011). This process will be repeated using two different sampling methods to determine the differences between each one's effect on the detection of HHHs. The true set of HHHs will be predetermined from the setup for the link flooding simulator. I will determine the change in accuracy per each method using the difference. Additionally, the amount of resources used by each sampling method will be tracked during the simulation. The first trial will be done using a software defined system with switch and controller communication provided by OpenFlow to sample the incoming data. The second simulation will use the enhanced flow tracking abilities of the Ryu controller to detect HHHs on a per flow basis, where the flows are being matched by their destination addresses. For my simulation, I assumed an uncompromised network with

Table 4.1. Four Simulations were performed in this thesis, two SDN configurations against two different traffic patterns

Experiments	
Experiment Description	Experiment Purpose
Targeted heavy flows analyzed with raw stats	Illustrate how HHHs can be identified from basic traffic patterns.
Targeted heavy flows analyzed by flow grouping	Illustrate how the advanced per flow analysis is also effective at identifying HHHs.
Spread heavy flows analyzed with raw stats	Illustrate the ineffectiveness of naively analyzing raw stats when detecting HHHs for complex traffic patterns.
Spread heavy flows analyzed by flow grouping	Display the effectiveness of flow grouping for identifying HHHs when traffic patterns are complex.

trusted receiving hosts and controller. The transmitting hosts were representations of users of the system that were communicating with the hosts. The hosts that were used as vehicles for the heavy flow traffic were representations of compromised parties. In my simulations, I hope to observe how effective each system is at identifying the number of HHHs that can be identified from the traffic after each update to the switch. Furthermore, I want to track the amount of overhead each system produces during the simulation. Tracking the overhead allows me to characterize the scalability of each configuration.

Chapter 5

RESULTS

I evaluated the effectiveness of the HHH algorithm through the two configurations using multiple measures. The results of each simulation are included in the following graphs in this chapter. The X axis represents an update of traffic statistics sent to the controller from a SDN switch in the emulation. The Y axis represents the number of HHH traffic sources determined to exist from the information the controller received in the update. My analysis of the data is based on how accurately each configuration could determine the true number of HHHs from the traffic flows and how quickly it could detect any changes in that number as the simulation continued. The flows for the traffic come from data sent from the left hosts to the right hosts and the response traffic as well, response traffic could only be gathered from using the TCP protocol in the simulation. For the UDP protocol the results of the traffic analysis would have been different. For the first configuration using the emulated software defined system, I obtained the traffic statistics from every switch update sent to the controller. This setup featured an OpenFlow environment that did not take advantage of using per flow statistics. The HHHs detected for each update of the switches statistics are displayed in Figure 5.1.

Figure 5.1 shows that for the destination addresses, the number of HHHs initially discovered were at the locations of the 10 incoming data paths, which then shifted to the

FIG. 5.1. The results from the targeted heavy flow traffic simulation on both SDN configurations. The upper graph represents raw traffic statistics analyzed by the HHH algorithm, while the lower figure represents analysis done by flow

three data paths after the change in traffic type. For the next setup that shows the use of flow matching, the results are shown in the lower graph in Figure 5.1. In this figure, the identification of three hierarchical heavy hitters after the introduction of denial of service sources was much more apparent. The analysis for the second setup used an iteration of the HHH algorithm for every flow that was sent to the controller as part of the switch statistic update.

The next traffic set tested on both network configuration was the spread heavy flows set. The difference between the two networks steups under these traffic conditions can be seen in Figure 5.2. The expected shift in detected heavy hitters previously seen in 5.1 never occurs for the first network setup. In the third setup however, there is a visible shift in detected heavy hitters when the spread heavy flow traffic set begins its switch in the simulation. On the second graph of Figure 5.2 there is a recognizable return to the chart patterns of Figure 5.1 while taking into account the distributed traffic that is being run through the system to 10 destination hosts.

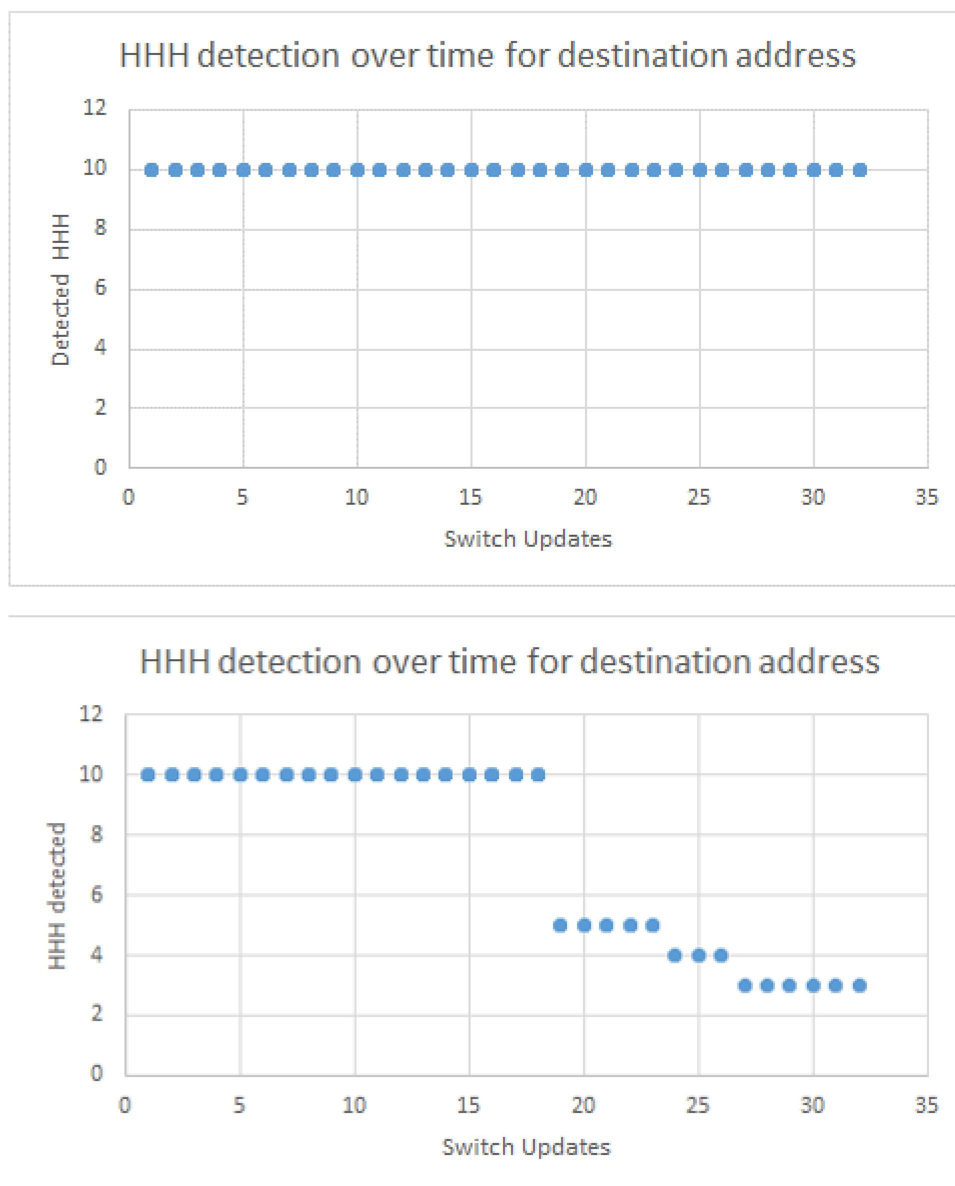

FIG. 5.2. The results from the spread heavy flow traffic simulation on both SDN configurations. The graph on the left represents the results from the controller using the HHH algorithm on raw stats, while the left took advantage of flows

Chapter 6

DISCUSSION

The results of the simulations clearly show differences between the detection capabilities of each setup. For each set of traffic patterns, the correct output for each graph should be 10 HHHs detected initially before a switch to three HHHs after the traffic sets start sending heavy flows. Examining the left graphs in Figures 5.1 and 5.2, it can be seen that for the targeted heavy flows, both configurations do an adequate job of correctly identifying the expected number of HHHs, although the first configuration is somewhat slower adapting to the changes in the number of HHHs than the second one. The results for this traffic set show that the HHH algorithm implemented was easily able to separate the ten equal flows based on their destination. Consequently, when the three heavy flows began to enter the system, they stood out from the rest of the seven average flows and constituted a majority of the traffic throughput. For the spread heavy flows traffic set, Figure 5.2 shows absolutely no change when the traffic set begins sending heavy flows. Yet for the other side of Figure 5.2, the shift in detected HHHs, although not as immediate as in Figure 5.1, is still there as it settles onto three HHHs detected by the end of the simulation. The reason that the first configuration did not react to the addition of heavy flows into the system is due to its method of analyzing traffic merely through switch updates; it had no method to group the incoming data by flows. As a result, simply basing its analysis on the destination

of all packets, the flows spread the bytes evenly across all ten destination hosts, leading to ten HHHs. In contrast, for the second configuration, where the controller matched flow information gathered from the switch updates by destination address, the system was able to determine the true number of HHHs after the heavy flows were introduced into the system.

Additionally in the final setup, due to the leveraging of OpenFlow's matching capability, the controller matched all reported flows by their destination address. Thus there were differences in overhead between the two configurations. The code for analysis through matching flows had an O(n^2) run time. Consequently, for the targeted heavy flow traffic, the first configuration had slightly lower overhead due to there being no overlap of destination flows. For the second pattern of spread heavy flows however, there was a significant difference in overhead in favor of the second configuration. For the first configuration where there was no flow matching; all 100 flows had to be analyzed and input into the HHH algorithm.

In the second configuration, where there was flow matching, all the flows to each destination were grouped together and then input into the HHH algorithm resulting in a fraction of calls to the HHH algorithm. Thus for a likely scenario of network traffic where there are exponentially more source hosts from outside the system sending data to a network with limited destination hosts, this approach is viable. The simulation results indicate that the technique of leveraging OpenFlow's flow matching capabilities were more effective and efficient in detecting potential denial of service attack vectors.

When comparing the effectiveness of using the flow matching capability in my SDN network, it must be mentioned that even legacy networks have the capability to track data entering its systems using a flow based approach. Netflow provided by Cisco is one such tool that network administrators can use for such a purpose. Researchers have explored how Netflow can be used to sample flows and gain an understanding of system traffic (Hofstede *et al.* 2014). The information gained from Netflow's aggregated flows however, can only

be utilized and acted on when an administrator accesses the Netflow Collector's storage. In contrast, for the system implemented in this paper, mitigation techniques can be activated at the onset of the detection of a change in the number of HHHs. The rapid reaction and flexibility that is available to a SDN is what makes the system more effective in addressing such attacks when compared to legacy network environments.

6.1 Future Questions

The results of this thesis presented new and interesting questions as well as further areas to explore. The next steps of this research should include studying the efficacy of alternative streaming algorithms when used in this setup. The experiments I performed in this thesis made use of the HHH algorithm to identify heavy flows, but there are alternative streaming algorithms, and determining if the gains in their effectiveness if any that could come from leveraging the inherent abilities of SDNs is a pertinent question.

Another step, would be the implementation of the flow tracking method on a legacy network to quantify the advantages that can be gained from leveraging the separation of the data and control planes. Additionally, a problem for the future is better understanding of how scalable the system implemented in this thesis is. Another future point, is how to further leverage the capabilities provided by SDNs by using the network viewpoints provided by multiple switches in concert. This thesis focused on gathering flow data from a single switch for analysis, but many networks must track incoming data through multiple switches, and then reroute the data through secondary switches before it reaches the final hosts destination. Consequently, being able to take advantage of the data coming from all the switches in this process to gain a more holistic view could lead to more effective analysis.

Another key aspect of SDNs that was discussed in this thesis but not applied in the

experiment, is the ability of a SDN to adapt to traffic coming into the network to more efficiently handle it. A future application of this process could be to split incoming flows to the system, so that suspicious and anomalous flows are passed through more rigorous sampling and analysis to determine whether they are malicious or benign. Finally, studying the gains in mitigation efficiency that come from quicker and more accurate detection from the techniques proposed in this thesis also needs further study.

Chapter 7

CONCLUSION

This thesis is focused on identifying the advantages that can be realized from leveraging the inherent abilities of software defined networks when using streaming algorithms to detect heavy flows. Detecting Distributed Denial of Service attacks on a network often involves the detection of heavy flows or other anomalous traffic. I evaluated this process through simulating network traffic on an emulated software defined network, while running multiple setups that used varying levels of SDN capabilities. From the data, I was able to see clear differences in how effective the streaming algorithm was under each circumstance. In conclusion, I can state that the use of the Hierarchical Heavy Hitter algorithm in combination with periodic per flow updates from software defined switches can improve the ability to detect certain types of denial of service attacks.

Appendix A

SOURCE CODE

A.1 Hierarchical Heavy Hitter Algorithm for raw statistics

```
import json


pertot = 0
entry = 0

class Trie:

    def __init__(self, maxdepth, maxchild):
        self.maxdepth = maxdepth
        self.maxchild = maxchild
        self.child = [None] * self.maxchild
           # arry of pointers to children in
           tree
        self.depth = 0  # depth of the node
        self.fringe = True      # true iff
           subtrie < T_split
```

```
        self.volume = 0 # volume of traffic
           trapped at node
        self.pastent = []
        self.pastmem = []
        self.pastval = []

H = [ [None]*13 for i in range(2) ]
T_split = 20
tries = [Trie(1,10), Trie(12,16)]

def get_Nth_bit(key, depth):
    #get the nth bit in the key
    return key[depth-1]

def prefix(key, leng):
    #return the matching prefix
    return key[0:leng]

def get_child(n, index):
    #get the specified child
    return n.child[index]

def printTrie(n, ind):
    print '   ' * n.depth + ind
    for i, ch in enumerate(n.child):
        printTrie(ch, i)
```

```
def match_bit(mem, dep, maxd):
    if dep >= len(mem):
        print str(maxd) + " " + str(dep) + " "
           + mem
    return mem[dep]


def create_children(n):
    #create a new trie
    for i in range(0,n.maxchild):
        n.child[i] = Trie(n.maxdepth,
           n.maxchild)
        n.child[i].fringe = True
        n.child[i].volume = 0
        n.child[i].depth = n.depth + 1
        #fill the trie with the flows that made
           up it's volume
        for j in range(0, len(n.pastmem) - 1):
            if i ==
               int(match_bit(n.pastmem[j],n.child[i].depth-1,
               n.maxdepth),16) and n.pastent[j]
               not in n.child[i].pastent:
                n.child[i].volume =
                   n.child[i].volume +
                   n.pastval[j]
                n.child[i].pastent.append(n.pastent[j])
                n.child[i].pastmem.append(n.pastmem[j])
                n.child[i].pastval.append(n.pastval[j])
```

```
def Update(trieroot, key, value, ent):
    #return the depth of the node
    n = trieroot
    while 1==1:
        if ent not in n.pastent:
            n.pastent.append(ent)
            n.pastmem.append(key)
            n.pastval.append(value)
        if n.fringe:
            if (n.volume + value / ( pertot *
               1.0 )) * 100 < T_split:
                n.volume = n.volume + value
                return n.depth - 1
            else:
                n.fringe = False
                if n.depth == n.maxdepth:
                    n.volume = value + n.volume
                    return n.depth
        else:
            if n.depth == n.maxdepth:
                n.volume = n.volume + value
                return n.depth
        newdep = n.depth + 1
        index = get_Nth_bit(key, newdep)
        c = get_child(n,int(index, 16)) ##takes
           a single character and puts it as an
```

```
            int so dec maps to dec and hex maps
            to hex
        if c == None:
             create_children(n)
             c = get_child(n,int(index, 16))
        n = c
        n.volume = n.volume + value

def Update_CP(portkey,edestkey, value):
    #update function for each dimension
    global pertot
    global entry
    global tries
    entry = entry + 1
    pertot = pertot + value
    portkey = str(portkey)
    edestkey = edestkey.translate(None, ':')
    len1 = Update(tries[0], portkey, value,
       entry)
    len2 = Update(tries[1], edestkey, value,
       entry)

def HHHdet(trieroot, f):
    n = trieroot
    hhhsum = 0
    if n.depth != n.maxdepth and get_child(n,0)
       != None:
```

```
    for i in range(0,len(n.child)):
        hhhsum = hhhsum +
            HHHdet(get_child(n, i), f)
    val = (n.volume / ( pertot * 1.0 )) *
        100
    if val > 10 and val - hhhsum > 10:
        json.dump(entry, f)
        f.write(',')
        json.dump(n.volume, f)
        f.write(',')
        json.dump(n.pastmem[0][0:n.depth+1],
            f)
        f.write(',')
        json.dump(len(n.pastmem), f)
        f.write('\n')
        hhhsum = hhhsum + val
    return hhhsum
val = (n.volume / ( pertot * 1.0 )) * 100
if val > 10:
    json.dump(entry, f)
    f.write(',')
    json.dump(n.volume, f)
    f.write(',')
    json.dump(n.pastmem[0][0:n.depth+1], f)
    f.write(',')
    json.dump(len(n.pastmem), f)
    f.write('\n')
```

```
        hhhsum = hhhsum + val
    return hhhsum

def checkHHH():
    if pertot != 0:
        f = open('HHH2filep.txt', 'a')
        HHHdet(tries[0], f)
        f.close()
        f = open('HHH2filed.txt', 'a')
        HHHdet(tries[1], f)
f.close()
```

A.2 Hierarchical Heavy Hitter Algorithm for stats grouped by flow

```
import json

pertot = [0,0]
entry = 0

class Trie:

    def __init__(self, maxdepth, maxchild):
        self.maxdepth = maxdepth
        self.maxchild = maxchild
        self.child = [None] * self.maxchild
           # arry of pointers to children in
           tree
        self.depth = 0  # depth of the node
```

```
        self.fringe = True      # true iff
           subtrie < T_split
        self.volume = 0 # volume of traffic
           trapped at node
        self.pastent = []
        self.pastmem = []
        self.pastval = []

H = [ [None]*13 for i in range(2) ]
T_split = 20
portbyttrie = Trie(1,10)
destbyttrie = Trie(12,16)
portpacktrie = Trie(1,10)
destpacktrie = Trie(12,16)

def get_Nth_bit(key, depth):
    #get the nth bit in the key
    return key[depth-1]

def prefix(key, leng):
    #return the matching prefix
    return key[0:leng]

def get_child(n, index):
    #get the specified child
    return n.child[index]
```

```
def printTrie(n, ind):
    print '   ' * n.depth + ind
    for i, ch in enumerate(n.child):
        printTrie(ch, i)

def match_bit(mem, dep, maxd):
    if dep >= len(mem):
        print str(maxd) + " " + str(dep) + " "
           + mem
    return mem[dep]

def create_children(n):
    #create a new trie
    for i in range(0,n.maxchild):
        n.child[i] = Trie(n.maxdepth,
           n.maxchild)
        n.child[i].fringe = True
        n.child[i].volume = 0
        n.child[i].depth = n.depth + 1
        #fill the trie with the flows that made
           up it's volume
        for j in range(0, len(n.pastmem) - 1):
            if i ==
               int(match_bit(n.pastmem[j],n.child[i].depth-1,
               n.maxdepth),16) and n.pastent[j]
               not in n.child[i].pastent:
                n.child[i].volume =
```

```
                        n.child[i].volume +
                        n.pastval[j]
                    n.child[i].pastent.append(n.pastent[j])
                    n.child[i].pastmem.append(n.pastmem[j])
                    n.child[i].pastval.append(n.pastval[j])

def Update(trieroot, key, value, ent, totind):
    #return the depth of the node
    n = trieroot
    while 1==1:
        if ent not in n.pastent:
            n.pastent.append(ent)
            n.pastmem.append(key)
            n.pastval.append(value)
        if n.fringe:
            if (n.volume + value / (
               pertot[totind] * 1.0 )) * 100 <
               T_split:
                n.volume = n.volume + value
                return n.depth - 1
            else:
                n.fringe = False
                if n.depth == n.maxdepth:
                    n.volume = value + n.volume
                    return n.depth
        else:
            if n.depth == n.maxdepth:
```

```
                n.volume = n.volume + value
                return n.depth
        newdep = n.depth + 1
        index = get_Nth_bit(key, newdep)
        c = get_child(n,int(index, 16)) ##takes
           a single character and puts it as an
           int so dec maps to dec and hex maps
           to hex
        if c == None:
            create_children(n)
            c = get_child(n,int(index, 16))
        n = c
        n.volume = n.volume + value

def Update_CP(portkey,edestkey, bytvalue,
   pacvalue):
    #update function for each dimension
    global pertot
    global entry
    entry = entry + 1
    pertot[0] = pertot[0] + bytvalue
    pertot[1] = pertot[1] + pacvalue
    portkey = str(portkey)
    edestkey = edestkey.translate(None, ':')
    len1 = Update(portbyttrie, portkey,
       bytvalue, entry, 0)
    len2 = Update(destbyttrie, edestkey,
```

```
        bytvalue, entry, 0)
    len3 = Update(portpacktrie, portkey,
        pacvalue, entry, 1)
    len4 = Update(destpacktrie, edestkey,
        pacvalue, entry, 1)

def HHHdet(trieroot, totind, f):
    n = trieroot
    hhhsum = 0
    if n.depth != n.maxdepth and get_child(n,0)
        != None:
        for i in range(0,len(n.child)):
            hhhsum = hhhsum +
                HHHdet(get_child(n, i), totind,
                f)
        val = (n.volume / ( pertot[totind] *
            1.0 )) * 100
        if val > 10 and val - hhhsum > 10:
            json.dump(entry, f)
            f.write(',')
            json.dump(n.volume, f)
            f.write(',')
            json.dump(n.pastmem[0][0:n.depth+1],
                f)
            f.write(',')
            json.dump(len(n.pastmem), f)
            f.write('\n')
```

```
            hhhsum = hhhsum + val
        return hhhsum
    val = (n.volume / ( pertot[totind] * 1.0 ))
       * 100
    if val > 10:
        json.dump(entry, f)
        f.write(',')
        json.dump(n.volume, f)
        f.write(',')
        json.dump(n.pastmem[0][0:n.depth+1], f)
        f.write(',')
        json.dump(len(n.pastmem), f)
        f.write('\n')
        hhhsum = hhhsum + val
    return hhhsum

def checkHHH():
    if pertot[0] != 0:
        f = open('HHH3filepb.txt', 'a')
        HHHdet(portbyttrie, 0, f)
        f.close()
        f = open('HHH3filedb.txt', 'a')
        HHHdet(destbyttrie, 0, f)
        f.close()
        f = open('HHH3filepp.txt', 'a')
        HHHdet(portpacktrie, 1, f)
        f.close()
```

```
        f = open('HHH3filedp.txt', 'a')
        HHHdet(destpacktrie, 1, f)
f.close()
```

A.3 Controller for raw stats

```
#import statements
from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import
   CONFIG_DISPATCHER, MAIN_DISPATCHER,
   DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib import hub
import Update2

class
   SamplerSwitch(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SamplerSwitch,
           self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread =
```

```
        hub.spawn(self._monitor) #hub that
        will query switches

@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER,
               DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in
           self.datapaths:
            self.logger.debug('register
               datapath: %016x',
               datapath.id)
            self.datapaths[datapath.id] =
               datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister
               datapath: %016x',
               datapath.id)
            del self.datapaths[datapath.id]

def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
```

```
            self.clear_flows(dp)
        hub.sleep(10)

def _request_stats(self, datapath):
    if datapath.id == 0000000000000001:
        self.logger.debug('send stats
           request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        req =
           parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

def clear_flows(self, datapath):
    if datapath.id == 0000000000000001:
        self.logger.debug('send stat clear:
           %016x', datapath.id)
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto
        actions =
           [parser.OFPActionOutput(ofproto.OFPP_NORMAL,
           0)]
        inst = [
           parser.OFPInstructionActions(
           ofproto.OFPIT_APPLY_ACTIONS,
           actions)]
        for i in range(1,12):
```

```
            req = parser.OFPFlowMod(
               datapath, 0, 0, 0,
               ofproto.OFPFC_MODIFY, 0, 0,
                 1, ofproto.OFP_NO_BUFFER,
               ofproto.OFPP_ANY,
               ofproto.OFPG_ANY,
               ofproto.OFPFF_RESET_COUNTS,
               parser.OFPMatch(in_port= i
               ), inst)
            datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply,
   MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):

    body = ev.msg.body

    self.logger.info('datapath         '
                     'in-port  out-port
                        eth-dst            '
                     'eth-src
                        packets  bytes   '
                     'ip-src
                        ip-dst            ')
    self.logger.info('---------------- '
                     '--------  --------
                        ----------------- '
```

```
                                    '-----------------
                                       -------- --------'
                                    '-----------------
                                       -----------------')

                 for stat in sorted([flow for flow in
                    body if flow.priority == 1]):
                     src = None
                     if 'eth_src' in stat.match:
                         src = stat.match['eth_src']
                     self.logger.info('%016x %8x %8x
                        %17s %17s %8d %8d',
                                       ev.msg.datapath.id,
                                       stat.match['in_port'],
                                       stat.instructions[0]
                                  .actions[0].port,
                                       stat.match['eth_dst'],
                                       src,
                                       stat.packet_count,
                                          stat.byte_count)#,
#
   stat.match['ipv4_src'],
   stat.match['ipv4_dst'])
            Update2.Update_CP(
               stat.instructions[0].actions[0].port,
               stat.match['eth_dst'],
               stat.byte_count)
```

```
Update2.checkHHH()
```

A.4 Controller for flow grouping

```
#import statements
from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import
   CONFIG_DISPATCHER, MAIN_DISPATCHER,
   DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib import hub
import Update3

class
   SamplerSwitch(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SamplerSwitch,
           self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread =
           hub.spawn(self._monitor) #hub that
           will query switches
```

```
@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER,
               DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in
           self.datapaths:
            self.logger.debug('register
               datapath: %016x',
               datapath.id)
            self.datapaths[datapath.id] =
               datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister
               datapath: %016x',
               datapath.id)
            del self.datapaths[datapath.id]

def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
            self.clear_flows(dp)
        hub.sleep(10)
```

```
def _request_stats(self, datapath):
    if datapath.id == 0000000000000001:
        self.logger.debug('send stats
           request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        req =
           parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

def clear_flows(self, datapath):
    if datapath.id == 0000000000000001:
        self.logger.debug('send stat clear:
           %016x', datapath.id)
        parser = datapath.ofproto_parser
        ofproto = datapath.ofproto
        actions = [parser.OFPActionOutput(
           ofproto.OFPP_NORMAL, 0)]
        inst =
           [parser.OFPInstructionActions(
           ofproto.OFPIT_APPLY_ACTIONS,
           actions)]
        for i in range(1,12):
            req = parser.OFPFlowMod(
               datapath, 0, 0, 0,
               ofproto.OFPFC_MODIFY,  0, 0,
```

```
                1, ofproto.OFP_NO_BUFFER,
                ofproto.OFPP_ANY,
                ofproto.OFPG_ANY,
                ofproto.OFPFF_RESET_COUNTS,
                parser.OFPMatch(in_port= i
                ), inst)
            datapath.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply,
   MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    body = ev.msg.body
    flows = []
    self.logger.info('datapath         '
                     'in-port  out-port
                        eth-dst            '
                     'eth-src
                        packets  bytes   '
                     'ip-src
                        ip-dst           ')
    self.logger.info('---------------- '
                     '--------  --------
                        ----------------- '
                     '-----------------
                        -------- --------'
                     '-----------------
                        ----------------')
```

```
        for stat in sorted([flow for flow in
           body if flow.priority == 1]):
            src = None
            if 'eth_src' in stat.match:
                src = stat.match['eth_src']
            self.logger.info('%016x %8x %8x
               %17s %17s %8d %8d',
                             ev.msg.datapath.id,
                             stat.match['in_port'],
                             stat.instructions[
                                0].
                        actions[0].port,
                             stat.match['eth_dst'],
                             src,
                             stat.packet_count,
                                stat.byte_count)#,
#
   stat.match['ipv4_src'],
   stat.match['ipv4_dst'])
            flows.append(stat)

        while len(flows) > 0:
            flowcount = 1.0
            bytecount = 0;
            packetcount = 0;
            i = 1
            while i < len(flows):
```

```
                if flows[0].match['eth_dst'] ==
                   flows[i].match['eth_dst']:
                    bytecount =
                       flows[i].byte_count +
                       bytecount
                    packetcount =
                       flows[i].packet_count +
                       packetcount
                    flowcount = flowcount + 1
                    del flows[i]
                    i = i-1
                i = i + 1
            Update3.Update_CP(
               flows[0].instructions[0].actions[0].port,
               flows[0].match['eth_dst'],
               (flows[0].byte_count +
               bytecount) / flowcount,
               (packetcount +
               flows[0].packet_count) /
               flowcount)
            del flows[0]
Update3.checkHHH()
```

A.5 Targeted heavy flow traffic generation

```
from mininet.net import Mininet
from mininet.node import RemoteController
```

```
from mininet.cli import CLI
import time

def dosSim():
    net = Mininet( controller=RemoteController )

    net.addController ('c0' ,
       controller=RemoteController)

    ##build network
    # Add hosts and switches
    leftoneHost = net.addHost( 'h1' )
    lefttwoHost = net.addHost( 'h2' )
    leftthreeHost = net.addHost( 'h3' )
    leftfourHost = net.addHost( 'h4' )
    leftfiveHost = net.addHost( 'h5' )
    leftsixHost = net.addHost( 'h6' )
    leftsevenHost = net.addHost( 'h7' )
    lefteightHost = net.addHost( 'h8' )
    leftnineHost = net.addHost( 'h9' )
    lefttenHost = net.addHost( 'h10' )
    rightoneHost = net.addHost( 'h11' )
    righttwoHost = net.addHost( 'h12' )
    rightthreeHost = net.addHost( 'h13' )
    rightfourHost = net.addHost( 'h14' )
    rightfiveHost = net.addHost( 'h15' )
    rightsixHost = net.addHost( 'h16' )
```

```
rightsevenHost = net.addHost( ’h17’ )
righteightHost = net.addHost( ’h18’ )
rightnineHost = net.addHost( ’h19’ )
righttenHost = net.addHost( ’h20’ )
leftSwitch = net.addSwitch( ’s1’ )
rightSwitch = net.addSwitch( ’s2’ )
centerSwitch = net.addSwitch( ’s3’ )

# Add links
net.addLink( leftoneHost, leftSwitch )
net.addLink( lefttwoHost, leftSwitch )
net.addLink( leftthreeHost, leftSwitch )
net.addLink( leftfourHost, leftSwitch )
net.addLink( leftfiveHost, leftSwitch )
net.addLink( leftsixHost, leftSwitch )
net.addLink( leftsevenHost, leftSwitch )
net.addLink( lefteightHost, leftSwitch )
net.addLink( leftnineHost, leftSwitch )
net.addLink( lefttenHost, leftSwitch )
net.addLink( rightoneHost, rightSwitch )
net.addLink( righttwoHost, rightSwitch )
net.addLink( rightthreeHost, rightSwitch )
net.addLink( rightfourHost, rightSwitch )
net.addLink( rightfiveHost, rightSwitch )
net.addLink( rightsixHost, rightSwitch )
net.addLink( rightsevenHost, rightSwitch )
net.addLink( righteightHost, rightSwitch )
```

```
    net.addLink( rightnineHost, rightSwitch )
    net.addLink( righttenHost, rightSwitch )
    net.addLink( leftSwitch, centerSwitch )
    net.addLink( rightSwitch, centerSwitch )

    ##start network
    net.start()
    CLI( net )

    ##en network
    net.stop()

##traffic command
def dostrafcmd( self, line ):
    "this command executes the scripts for
       traffic generation"
    net = self.mn
    h1 = net.get('h1')
    h2 = net.get('h2')
    h3 = net.get('h3')
    h4 = net.get('h4')
    h5 = net.get('h5')
    h6 = net.get('h6')
    h7 = net.get('h7')
    h8 = net.get('h8')
    h9 = net.get('h9')
    h10 = net.get('h10')
```

```
h11 = net.get('h11')
h12 = net.get('h12')
h13 = net.get('h13')
h14 = net.get('h14')
h15 = net.get('h15')
h16 = net.get('h16')
h17 = net.get('h17')
h18 = net.get('h18')
h19 = net.get('h19')
h20 = net.get('h20')
# send normal traffic through system
h11.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h1.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.11 -c 200 -C 1 -t
   150000 -l sender1.log -x receiver1.log
   &')
h12.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h2.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.12 -c 200 -C 1 -t
   150000 -l sender2.log -x receiver2.log
   &')
h13.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
```

```
    &' )
h3.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.13 -c 200 -C 1 -t
    150000 -l sender3.log -x receiver3.log
    &')
h14.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h4.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.14 -c 200 -C 1 -t
    150000 -l sender4.log -x receiver4.log
    &')
h15.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h5.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.15 -c 200 -C 1 -t
    150000 -l sender5.log -x receiver5.log
    &')
h16.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h6.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.16 -c 200 -C 1 -t
    150000 -l sender6.log -x receiver6.log
    &')
h17.cmd(
```

```
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h7.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.17 -c 200 -C 1 -t
    150000 -l sender7.log -x receiver7.log
    &')
h18.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h8.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.18 -c 200 -C 1 -t
    150000 -l sender8.log -x receiver8.log
    &')
h19.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h9.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.19 -c 200 -C 1 -t
    150000 -l sender9.log -x receiver9.log
    &')
h20.cmd(
    '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &' )
h10.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.20 -c 200 -C 1 -t
    150000 -l sender10.log -x
    receiver10.log')
```

```
##mixed traffic
h11.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h1.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.11 -c 200 -C 1 -t
   150000 -l sender1.log -x receiver1.log
   &')
h12.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h2.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.12 -c 200 -C 1 -t
   150000 -l sender2.log -x receiver2.log
   &')
h13.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h3.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.13 -c 200 -C 10 -t
   150000 -l sender3.log -x receiver3.log
   &')
h14.cmd(
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &' )
h4.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.14 -c 200 -C 1 -t
```

```
    150000 -l sender4.log -x receiver4.log
    &’)
h15.cmd(
    ’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &’ )
h5.cmd(’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.15 -c 200 -C 1 -t
    150000 -l sender5.log -x receiver5.log
    &’)
h16.cmd(
    ’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &’ )
h6.cmd(’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.16 -c 200 -C 10 -t
    150000 -l sender6.log -x receiver6.log
    &’)
h17.cmd(
    ’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &’ )
h7.cmd(’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
    -T TCP -a 10.0.0.17 -c 200 -C 1 -t
    150000 -l sender7.log -x receiver7.log
    &’)
h18.cmd(
    ’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
    &’ )
h8.cmd(’/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
```

```
        -T TCP -a 10.0.0.18 -c 200 -C 1 -t
        150000 -l sender8.log -x receiver8.log
        &')
    h19.cmd(
        '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
        &' )
    h9.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
        -T TCP -a 10.0.0.19 -c 200 -C 10 -t
        150000 -l sender9.log -x receiver9.log
        &')
    h20.cmd(
        '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
        &' )
    h10.cmd('/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
        -T TCP -a 10.0.0.20 -c 200 -C 1 -t
        150000 -l sender10.log -x
        receiver10.log')

CLI.do_dostrafcmd = dostrafcmd

if __name__ == '__main__':
dosSim()
\small
```

A.6 Spread heavy flow traffic generation

```
from mininet.net import Mininet
```

```
from mininet.node import RemoteController
from mininet.cli import CLI
import time

def dosSim():
    net = Mininet( controller=RemoteController )

    net.addController ('c0' ,
       controller=RemoteController)

    ##build network
    # Add hosts and switches
    leftoneHost = net.addHost( 'h1' )
    lefttwoHost = net.addHost( 'h2' )
    leftthreeHost = net.addHost( 'h3' )
    leftfourHost = net.addHost( 'h4' )
    leftfiveHost = net.addHost( 'h5' )
    leftsixHost = net.addHost( 'h6' )
    leftsevenHost = net.addHost( 'h7' )
    lefteightHost = net.addHost( 'h8' )
    leftnineHost = net.addHost( 'h9' )
    lefttenHost = net.addHost( 'h10' )
    rightoneHost = net.addHost( 'h11' )
    righttwoHost = net.addHost( 'h12' )
    rightthreeHost = net.addHost( 'h13' )
    rightfourHost = net.addHost( 'h14' )
    rightfiveHost = net.addHost( 'h15' )
```

```
rightsixHost = net.addHost( 'h16' )
rightsevenHost = net.addHost( 'h17' )
righteightHost = net.addHost( 'h18' )
rightnineHost = net.addHost( 'h19' )
righttenHost = net.addHost( 'h20' )
leftSwitch = net.addSwitch( 's1' )
rightSwitch = net.addSwitch( 's2' )
centerSwitch = net.addSwitch( 's3' )

# Add links
net.addLink( leftoneHost, leftSwitch )
net.addLink( lefttwoHost, leftSwitch )
net.addLink( leftthreeHost, leftSwitch )
net.addLink( leftfourHost, leftSwitch )
net.addLink( leftfiveHost, leftSwitch )
net.addLink( leftsixHost, leftSwitch )
net.addLink( leftsevenHost, leftSwitch )
net.addLink( lefteightHost, leftSwitch )
net.addLink( leftnineHost, leftSwitch )
net.addLink( lefttenHost, leftSwitch )
net.addLink( rightoneHost, rightSwitch )
net.addLink( righttwoHost, rightSwitch )
net.addLink( rightthreeHost, rightSwitch )
net.addLink( rightfourHost, rightSwitch )
net.addLink( rightfiveHost, rightSwitch )
net.addLink( rightsixHost, rightSwitch )
net.addLink( rightsevenHost, rightSwitch )
```

```
    net.addLink( righteightHost, rightSwitch )
    net.addLink( rightnineHost, rightSwitch )
    net.addLink( righttenHost, rightSwitch )
    net.addLink( leftSwitch, centerSwitch )
    net.addLink( rightSwitch, centerSwitch )

    ##start network
    net.start()
    CLI( net )

    ##en network
    net.stop()

##traffic command
def dostrafcmd( self, line ):
    "this command executes the scripts for
       traffic generation"
    net = self.mn
    h1 = net.get('h1')
    h2 = net.get('h2')
    h3 = net.get('h3')
    h4 = net.get('h4')
    h5 = net.get('h5')
    h6 = net.get('h6')
    h7 = net.get('h7')
    h8 = net.get('h8')
    h9 = net.get('h9')
```

```
h10 = net.get('h10')
h11 = net.get('h11')
h12 = net.get('h12')
h13 = net.get('h13')
h14 = net.get('h14')
h15 = net.get('h15')
h16 = net.get('h16')
h17 = net.get('h17')
h18 = net.get('h18')
h19 = net.get('h19')
h20 = net.get('h20')
# send normal traffic through system
recom =
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGRecv/ITGRecv
   &'
secom =
   '/home/ubuntu/D-ITG-2.8.1-r1023/src/ITGSend/ITGSend
   -T TCP -a 10.0.0.'
secom2 = ' -c 200 -C 1 -t 150000 -l sender'
secom2b = ' -c 200 -C 10 -t 150000 -l
   sender'
secom3 = '.log -x receiver'
secom4 = '.log &'
h11.cmd(recom)
h12.cmd(recom)
h13.cmd(recom)
h14.cmd(recom)
```

```
h15.cmd(recom)
h16.cmd(recom)
h17.cmd(recom)
h18.cmd(recom)
h19.cmd(recom)
h20.cmd(recom)
for i in range(1,10):
    h1.cmd(secom + str(i + 10) + secom2 +
       str(i) + '1' + secom3 + str(i) + '1'
       + secom4)
    h2.cmd(secom + str(i + 10) + secom2 +
       str(i) + '2' + secom3 + str(i) + '2'
       + secom4)
    h3.cmd(secom + str(i + 10) + secom2 +
       str(i) + '3' + secom3 + str(i) + '3'
       + secom4)
    h4.cmd(secom + str(i + 10) + secom2 +
       str(i) + '4' + secom3 + str(i) + '4'
       + secom4)
    h5.cmd(secom + str(i + 10) + secom2 +
       str(i) + '5' + secom3 + str(i) + '5'
       + secom4)
    h6.cmd(secom + str(i + 10) + secom2 +
       str(i) + '6' + secom3 + str(i) + '6'
       + secom4)
    h7.cmd(secom + str(i + 10) + secom2 +
       str(i) + '7' + secom3 + str(i) + '7'
```

```
            + secom4)
        h8.cmd(secom + str(i + 10) + secom2 +
            str(i) + '8' + secom3 + str(i) + '8'
            + secom4)
        h9.cmd(secom + str(i + 10) + secom2 +
            str(i) + '9' + secom3 + str(i) + '9'
            + secom4)
        h10.cmd(secom + str(i + 10) + secom2 +
            str(i) + '10' + secom3 + str(i) +
            '10' + secom4)
    h1.cmd(secom + str(20) + secom2 + '101' +
        secom3 + '101' + secom4)
    h2.cmd(secom + str(20) + secom2 + '102' +
        secom3 + '102' + secom4)
    h3.cmd(secom + str(20) + secom2 + '103' +
        secom3 + '103' + secom4)
    h4.cmd(secom + str(20) + secom2 + '104' +
        secom3 + '104' + secom4)
    h5.cmd(secom + str(20) + secom2 + '105' +
        secom3 + '105' + secom4)
    h6.cmd(secom + str(20) + secom2 + '106' +
        secom3 + '106' + secom4)
    h7.cmd(secom + str(20) + secom2 + '107' +
        secom3 + '107' + secom4)
    h8.cmd(secom + str(20) + secom2 + '108' +
        secom3 + '108' + secom4)
    h9.cmd(secom + str(20) + secom2 + '109' +
```

```
   secom3 + '109' + secom4)
h10.cmd(secom + str(20) + secom2 + '1010' +
   secom3 + '1010' + '.log')

##begin ddos traffic
h11.cmd(recom)
h12.cmd(recom)
h13.cmd(recom)
h14.cmd(recom)
h15.cmd(recom)
h16.cmd(recom)
h17.cmd(recom)
h18.cmd(recom)
h19.cmd(recom)
h20.cmd(recom)
for i in range(1,11):
    h1.cmd(secom + str(i + 10) + secom2 +
       str(i) + '1' + secom3 + str(i) + '1'
       + secom4)
    h2.cmd(secom + str(i + 10) + secom2 +
       str(i) + '2' + secom3 + str(i) + '2'
       + secom4)
    h3.cmd(secom + str(i + 10) + secom2b +
       str(i) + '3' + secom3 + str(i) + '3'
       + secom4)
    h4.cmd(secom + str(i + 10) + secom2 +
       str(i) + '4' + secom3 + str(i) + '4'
```

```
            + secom4)
        h5.cmd(secom + str(i + 10) + secom2 +
            str(i) + '5' + secom3 + str(i) + '5'
            + secom4)
        h6.cmd(secom + str(i + 10) + secom2b +
            str(i) + '6' + secom3 + str(i) + '6'
            + secom4)
        h7.cmd(secom + str(i + 10) + secom2 +
            str(i) + '7' + secom3 + str(i) + '7'
            + secom4)
        h8.cmd(secom + str(i + 10) + secom2 +
            str(i) + '8' + secom3 + str(i) + '8'
            + secom4)
        h9.cmd(secom + str(i + 10) + secom2b +
            str(i) + '9' + secom3 + str(i) + '9'
            + secom4)
        h10.cmd(secom + str(i + 10) + secom2 +
            str(i) + '10' + secom3 + str(i) +
            '10' + secom4)


CLI.do_dostrafcmd = dostrafcmd

if __name__ == '__main__':
dosSim()
```

REFERENCES

[1] Afek, Y.; Bremler-Barr, A.; Landau Feibish, S.; and Schiff, L. 2017. Detecting Heavy Flows in the SDN Match and Action Model. *ArXiv e-prints*.

[2] Botta, A.; Dainotti, A.; and Pescapè, A. 2012. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks* 56(15):3531–3547.

[3] Gamer, T., and Mayer, C. P. 2011. Simulative evaluation of distributed attack detection in large-scale realistic environments. *Simulation* 87(7):630–647.

[4] Giotis, K.; Argyropoulos, C.; Androulidakis, G.; Kalogeras, D.; and Maglaris, V. 2014. Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments. *Comput. Netw.* 62:122–136.

[5] Hofstede, R.; eleda, P.; Trammell, B.; Drago, I.; Sadre, R.; Sperotto, A.; and Pras, A. 2014. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys Tutorials* 16(4):2037–2064.

[6] Jose, L.; Yu, M.; and Rexford, J. 2011. Online measurement of large traffic aggregates on commodity switches. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, 13–13. Berkeley, CA, USA: USENIX Association.

[7] Kalliola, A.; Lee, K.; Lee, H.; and Aura, T. 2015. Flooding DDoS mitigation and traffic management with software defined networking. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, 248–254.

[8] Lantz, B.; Heller, B.; and McKeown, N. 2010. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, 19:1–19:6. New York, NY, USA: ACM.

[9] Mirkovic, J., and Reiher, P. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.* 34(2):39–53.

[10] Ranjan, S.; Shah, S.; Nucci, A.; Munafo, M.; Cruz, R.; and Muthukrishnan, S. 2007. Dowitcher: Effective worm detection and containment in the internet core. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, 2541–2545.

[11] Shirali-Shahreza, S., and Ganjali, Y. 2013. FleXam: Flexible sampling extension for monitoring and security applications in OpenFlow. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, 167–168. New York, NY, USA: ACM.

[12] Yan, Q.; Yu, F. R.; Gong, Q.; and Li, J. 2016. 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* 18(1):602–622.

[13] Yang, L.; Ng, B.; and Seah, W. K. G. 2016. Heavy hitter detection and identification in software defined networking. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, 1–10.

[14] Zhang, Y.; Singh, S.; Sen, S.; Duffield, N.; and Lund, C. 2004. Online identification of hierarchical heavy hitters: Algorithms, evaluation, and applications. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, 101–114. New York, NY, USA: ACM.