Software Defined Network Defense

Sumanth M. Sathyanarayana

A THESIS

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Master of Science in Engineering

2011-12

Jonathan M. Smith, Supervisor of Thesis

Jianbo Shi, Graduate Group Chairman

To my parents!

Contents

List of Figures

Acknowledgments

I would like to thank my Master's Thesis Advisor Prof. Jonathan M. Smith, who gave me an opportunity to work and research in this newly emerging field of Software Defined Networking and without whose advices and ideas, this Thesis would not have been possible. I would like to thank Prof. Nate Foster of Cornell University who gave constant support on our initial understanding of Frenetic and during our implementation of Pushback; his code on DDoS has been used by us as a base for our implementation. Heartfelt thanks to Prof. Boon Thau Loo for his inspiring advices on the work and for advising to implement Pushback using Frenetic. I also would like to thank John Sonchack, Second Year PhD student of the University of Pennsylvania for all his advices, ideas and brainstorming sessions which we had together, throughout my work. Finally and most importantly I would like to thank my parents for all their support in grooming me up and having played a vital role in my growth and development, for me to be at my present position!

Abstract

The increasingly sophisticated attacks on networks have been threatening their importance to the society. Distributed Denial of Service Attack(DDoS) is a good example in this regard which disrupts connectivity and prevents access to essential services by legitimate users. Through our work, we show how counter measures to such attacks can be deployed using Software Defined Networking(SDN) and maintain Networking Services for legitimate users. We primarily demonstrate that a Software Defined Defense mechanism can be deployed quickly for such malicious attacks and the future of such defense mechanism looks feasible and promising.

The languages used to program Computer Networks have traditionally had a lower-level abstraction and has been pretty cumbersome on the Network Researchers. With the new OpenFlow Architecture, where-in the Controlling Plane is separated from the Data Plane, we demonstrate an implementation of Software Defined Networking Defense against DDoS using Frenetic network programming Language which gives a declarative and higher level abstract to the Programmer. Our prototype uses Pushback Scheme of DDoS Defense Mechanism proposed by Ioannidis/Bellovin, to evaluate Frenetic's efficiency. We demonstrate success in our implementation, which we have tested using Mininet, with differences in performances based on variable parameters such as polling rate of the Network States.

1 Introduction

Today's closed and proprietary networks have made innovation and formation of policies difficult and complicated. With OpenFlow, a new network platform, it has led to the opening up of the network software and provided new innovative ways for researchers to run experimental protocols in the networks. The key idea in OpenFlow Networks is to segregate the control plane and the forwarding plane. This is achieved by having a separate general purpose machine running programs which form the control and management planes. It monitors network traffic, decides on routes and installs the forwarding tables on the switches, which forms the data/forwarding plane. This also results in simple, cheaper, flexible routers termed as switches in the context of OpenFlow.

Though OpenFlow has changed the network architecture and has in some way prevented the ossification of Network Research, the software used to control the networks(mainly NOX) has a lower layer abstraction and the Network Programmers have to constantly grapple with several challenges like maintaining complex book-keeping in a two-tiered system architecture and avoiding race conditions, which distributed systems are susceptible to. This led to the birth of Frenetic-a new Network Programming Language, which is easier, reliable, modular, secure and also provides a high level abstraction to programmers through a declarative paradigm. Frenetic consists of two integrated sub-languages: (1) Network Query Language which provides an abstraction for reading network state and (2) Functional and Reactive Network Pol-

icy Management Library which provides abstraction for specification of forwarding policy.

One of the important goals of this new architecture and Software Defined Networking (SDN) has been to make the highly vulnerable internet of today more secure and robust. Distributed Denial of Service attacks are hard to defend against because the attacks do not target specific vulnerabilities in a system, but the very fact that the victim is in the system and is connected to the network. Thus the vulnerability in case of DDoS is the entire system itself and the chief culprit would be the design of the internet. Our Report explains how Frenetic, using the new OpenFlow network Platform, can be utilized to prevent DDoS attacks. It gives us an idea on how Frenetic can be used in Software Defined Networking(SDN) to program complex security measures like DDoS in a simple and efficient way with a minimal code length.

Overall our contributions are as follows: (1) Implementation of a naive Pushback Defense Mechanism for the prevention of DDoS Attacks using Frenetic; (2) Evaluation of Frenetic 1.0 for Software Controlled Network Defense; (3) Providing a road-map on how newer versions of Frenetic could be used to provide better network programming in general and better DDoS prevention mechanisms and thus better Network Security in particular.

RouteMap The rest of the Report is organized as follows. In Section 2 a brief background of Openflow and NOX are provided and then a concise explanation on Frenetic is provided in Section3. Section 4 explains DDoS and Pushback. In Section 5, the

implementation of Pushback using Frenetic is explained. Section 6 evaluates Frenetic based on the current implementation. Section 7 tells about the future phase of Frenetic implementations and finally concludes in Section 8.

2 Background

2.1 Openflow

The traditional networks have usually been built using special purpose devices which run complicated distributed algorithms. Though this design makes the connectivity robust and scalable, it has always been an hindrance with regard to security. It has also resulted in extensive use of proprietary hardware and software and has ossified Network Research to a certain extent. Openflow architecture was developed mainly to allow researchers to evaluate their ideas in real world traffic settings as it allows them to experiment their protocols at line rate without having to change the existing router settings to a great deal; also the vendors need not expose the internal working of their products. The main principle in Openflow architecture, which is based on Software Defined Networking Paradigm, is to segregate the Control Plane from the Data Plane. A Central controller which can be a machine outside the network, can run simple programs on it and control ethernet switches, installing and updating rules on them to create the necessary forwarding table for data propagation. Software Defined Networking allows developers to write programs which tailor the network behavior and also suit certain applications. [21]

Figure 1 shows the traditional Network architecture where-in the control logic is intertwined with packet handling in the individual routers/switches distributed throughout an Autonomous System[15] Greenberg et. al. first considered of separating this im-

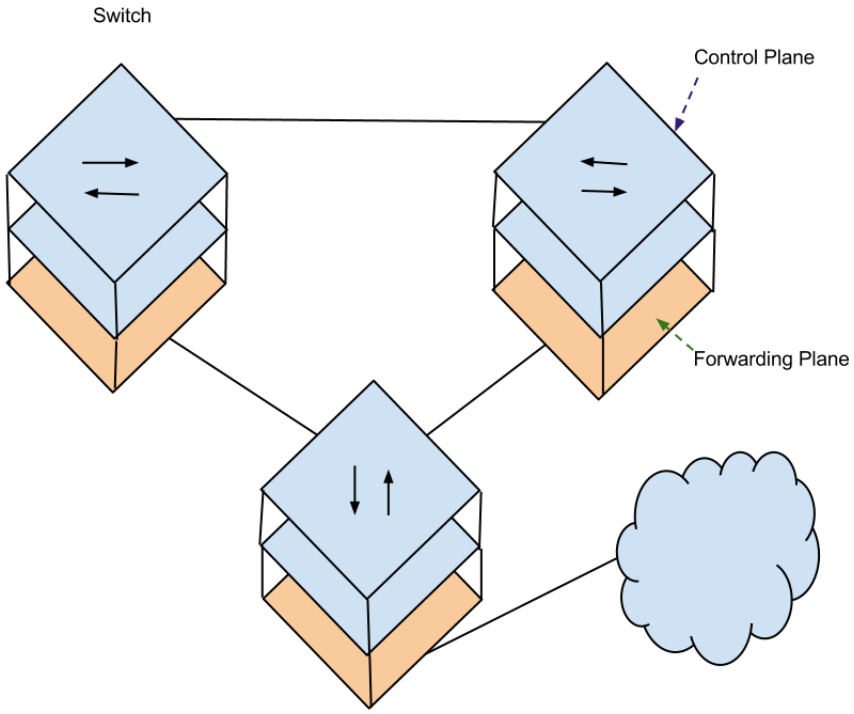

Figure 1: Typical Network Architecture

plicit embedding of the different planes in today's IP architecture and proposed a clean slate approach of having four different planes - Data, Discovery, Dissemination and Decision Plane which later underwent several metamorphisms and became one of the first influences for the Openflow architecture.

Figure 2 shows the Openflow architecture, whose chief components consist of the central controller and the switches. An Openflow switch is usually a dumb datapath element which carries out the actions specified by the controller between ports and consists of three main parts[21]:

1. Flow Table: It consists of an action field along with the flow which tells the

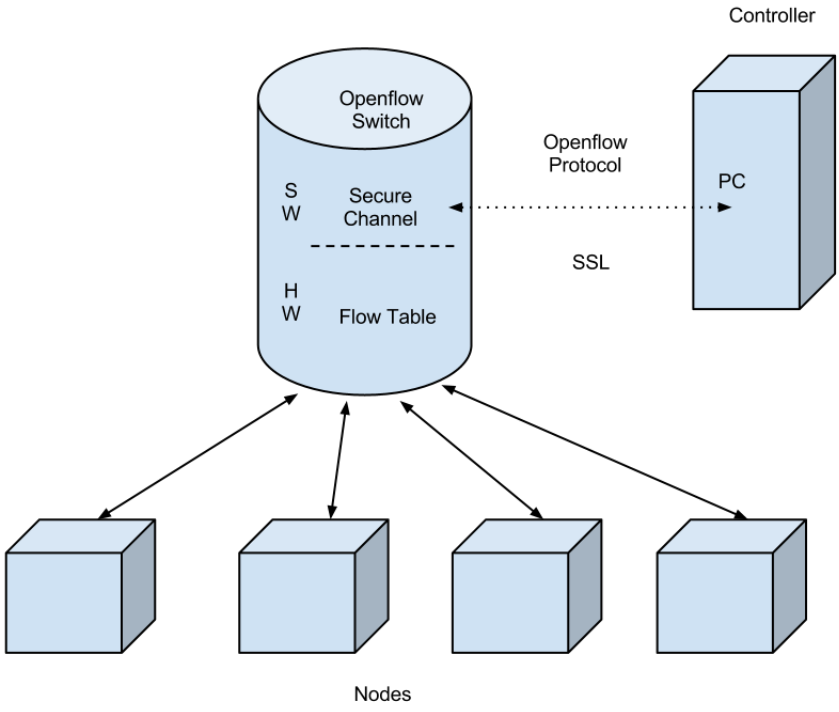

Figure 2: Openflow Architecture

switch on how a flow should be processed.

2. Secure Channel: A secure channel is present between the Controller and the switches which is very essential as Openflow architecture leads to single point of failure and an attacker overtaking the controller can lead to severe problems.

3. Openflow Protocol: It is the standard interface through which entries in the flow table can be defined, avoiding the necessity to program the switches.

The Central Controller computes the flow-entries based on the program or the protocol running on it. It also takes care of adding or deleting flow-entries from the flow

table present in the switches. A Controller can either be a simple PC or a dedicated sophisticated machine for handling complex tasks and having multiple accounts for different researchers to experiment their researches. [21]

The natural questions which arise from using a Central Controller is regarding the scalability, reliability and performance of using a single controller. These questions have been answered to some extent in one of the previous prototypes of such a model as in Ethane where-in a low-cost desktop PC could process around ten thousand flows per second which was good enough for a large college campus. Even scalability and reliability could be accounted for by having multiple separate controllers and each of them working on a stateless principle.

2.2 NOX

NOX is a Network Operating System which provides a central and uniform programmatic interface to the whole of the Openflow network. [16] Like an Operating System which provides read and write access to various resources, NOX provides the ability to view and control the network. Applications written on top of NOX would perform the actual Network management tasks, while NOX provides the following two major functionalities:

1. Centralized Programming Model, where-in programs would be written as if the entire network is present on a single machine. This gives an omniscient view of the network to the Programmer.

2. Programs are written in terms of high level abstractions.

NOX can be viewed as a software which has several control processes and one single *'network view'* which is kept in a database running on one of the servers. Applications running on NOX will make use of this Network view to manage the network. The granularity at which NOX provides observation is the Network view while the granularity for control is called a flow. A flow granularity is apt for a centralized network architecture like Openflow to scale since once control is exerted on some packet, the remaining packets with the same header need not go to the controller for this decision and would take the same route as the earlier packet via the switches itself i.e, when an incoming packet matches a certain flow, the switch updates the appropriate counters and applies the relevant actions. Only if the packet does not belong to a previously seen flow, it is sent to the controller. As the 'Network view' is a global data, NOX should take care of using it consistently across different process instances of the controller, whereas packet state and flow state which are not part of the network view can be stored locally in switches and controller instances respectively. The capacity of the system can be increased by increasing the number of servers which serve as the controller processes and parallelism can be introduced between them. Whenever an event happens like packet_receive or a switch_join, NOX applications execute a set of registered handlers for that particular event. The handling happens in an order or priority of their registration.[16]

3 Frenetic

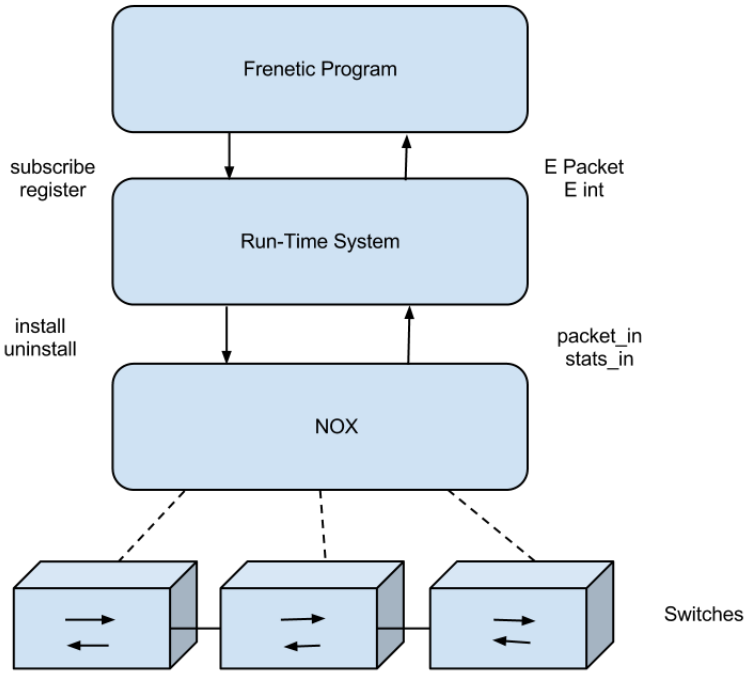

Figure 3: Frenetic Architecture

Frenetic is a network programming language embedded in Python. OpenFlow Network which provides a centralized controlling platform, runs NOX - Network Operating System on the controller. Though NOX provides a general programming interface, it still has a lower layer abstraction and has disadvantages like the programs do not compose which makes the interaction between two modules difficult. Programming in NOX often leads to Network race conditions and is a little cumbersome on the programmer due to its two tier system architecture where-in the programmer has to maintain the complex book keeping between the controller and the switches through

his program. Frenetic on the other hand mitigates the problems of NOX and provides the following functionalities:

1. *Declarative Design*Intuitive, high-level primitives are provided by Frenetic keeping in mind the programmer. Frenetic is primarily a Functional, Reactive Programming (FRP) Language in which programs manipulate streams of values. Frenetic programs function based on the abstraction of "see every packet." Event driven programs need not be written in Frenetic, unlike NOX, which leads to a unified architecture. Frenetic's unified architecture allows it to be expressed in simple, declarative query form while NOX used to specify rules that had to be laid on the switch along with the communication needed to retrieve counters from the switch.

2. *Modular Design* Frenetic's primitives have limited network-wide effects and semantics can be stated independently of the context. Frenetic's high level abstraction, frees programmers from worrying about the lower level details and allows programs to be written in a modular manner with re-useable parts. Frenetic's rich set of algebra and mathematical set theory like operators is what gives it a high level abstraction, without which as in NOX the programmer would have to bother about all the lower level switch details to implement his program. Also, one of the most important and interesting feature about Frenetic has been its support for composition of programs. Programmers can easily combine their programs without any concerns about undesirable interactions or

timing related issues as in NOX i.e., two queries can be merged using a generic combinator - Merge() without having to bother about the details of the individual query routines as Frenetic supports the abstraction that queries merely read the network state but do not modify it.

3. *Race-free Semantics* Superfluous packets arriving to the controller can be suppressed by the programmers as the Frenetic queries supply the information that programmer's want, to the run-time system. It creates an abstraction that a read of the network state does not make the controller see every packet and thus prevent race conditions. The syntax which majorly helps in this regard is the Limit clause: *Limit(n)* It specifies the number of packets which can be sent to the controller via the switches.

4. *Two Sub-Languages* Frenetic has a Query Language which is similar to any Database Query Language, but here the purpose of the query is to get the network information by the central controller with the controller having the omniscient view on the network(rather it can be thought of as the omniscient view is on the switches which are controlled by it, as a single controller cannot scale the entire internet). The other sub-language of Frenetic is its Run-Time Network Policy Management Library which provides an abstraction for the formulation of network policies. These network policies associate rules, generated using the rule constructor, with the switches.

3.1 Frenetic Data Types

Frenetic, based on the past FRP languages, uses three primary data types to represent, transform and consume streams of values.[18]

1. Frenetic mostly supports the discrete streams rather than continuous streams. These discrete, time varying streams of values are represented as Events in Frenetic. All Events of type α is written as αE which is an infinite list of pairs (t, v) where 't' is the timestamp and 'v' is the value of type $'\alpha'$. Some examples of 'events' are 'Packets' which contains all the packets flowing through the network, 'SwitchJoin' and 'SwitchLeave', which contains the identifiers of switch joining and leaving the network respectively.

2. "Event Functions" are the second data type that transform one event types into another. $\alpha\beta EF$ is the representation given to an Event Function which transforms an event of type α to an event of type β. An example for an Event Function, as Robert J. Harrison specifies in his Master's Thesis, if an EF g has type packet$\rightarrow$ bool, then $Group(g)$ splits the stream of packets into two streams, one in which g returns true and another in which g returns false.

3. The third data type is the Listener which consumes a stream of event and produces a side effect on the controller. Listeners of events αE is written as αL .An example of a Listener is 'Print' which takes its input and prints it on the console.

Some of the important Frenetic Operators are as below:

- Events

  $Seconds \in intE$

  $Packets \in packetE$

  $SwitchJoin \in switchE$

  $SwitchLeave \in switchE$

- Event Functions

  $>>\in \alpha E \to \alpha\beta EF \to \beta E$

  $Lift \in (\alpha \to \beta) \to \alpha\beta EF$

  $Merge \in (\alpha E \times \beta E) \to (\alpha option \times \beta option)\, E$

  $LoopPre \in (\gamma \times ((\alpha \times \gamma)\, (\beta \times \gamma)\, EF)) \to \alpha\beta EF$

  $Accum \in \gamma \times (\alpha \times \gamma \to \gamma) \to \alpha\gamma EF$

- Listeners

  $>>\in \alpha E \to \alpha L \to unit$

  $Print \in \alpha L$

  $Register \in policyL$

  $Send \in (switch \times packet \times action)\, L$

Some of the Frenetic query syntax is shown below:

- Queries $q ::= Select\,(a)*$

  $Where\,(fp)*$

  $GroupBy\,([qh_1, qh_2, ..., qh_k])*$

  $SplitWhen\,([qh_1, qh_2, ..., qh_k])*$

  $Every\,(n)*$

  $Limit\,(n)$

- Aggregates $a ::= packets|sizes|counts$

- Headers $qh ::= inport|srcmac|dstmac|ethtype|$

  $vlan|srcip|dstip|protocol|srcport|dstport|switch$

- Patterns $qp ::= true\_fp\,()\,|qh\_fp\,(n)\,|$

  $and\_fp\,([fp_1, \ldots, fp_n])\,|$

  $or\_fp\,([fp_1, \ldots, fp_n])\,|$

  $diff\_fp\,(fp_1, fp_2)\,|not\_fp\,(fp)$

3.2 Frenetic Run Time System

Though Frenetic provides high level abstractions and prevents the programmers from bothering about the lower level details, the need to deal with the lower level details just doesn't disappear because programs operate at a higher level. The lower level details

involving the underlying switch hardware is dealt by the Frenetic Run Time System. It is the core piece of the Frenetic Implementation which sits between the high-level program and NOX. Figure 4 depicts the architecture of Frenetic Run-time system. It does all the book-keeping of installing and uninstalling rules on switches and generates the necessary communication pattern between the switches and the controller. These functionalities make OpenFlow Network Programming Single tiered and unlike NOX, the programmers need not worry that installing packet-handling rules may prevent the controller from analyzing other traffic. At the same time, it is unacceptable that every packet is sent to the controller for analysis.

The subscribe queries define a set of 'packet subscribers' and 'statistic subscribers' for each program. The subscriber queries convey to the run-time system of Frenetic, the set of packets and statistics required for a particular program. The run-time system now knows that which packet is required to be seen by the controller, and hence a rule preventing such packets from being sent to the controller is not installed on the switches. While for statistic subscribers, the controller enforces the rules on the switches and the built in counters at OpenFlow switches are probed to obtain the statistic information. But, hereto the controller should install rules on the switches only when the rules do not interfere with other packet subscribers. If the traffic does not belong to either of the two streams of packet or statistic subscribers, then the controller can safely install rules on the switches and push its work away.

Frenetic's Run-time system can mainly be divided into three parts.

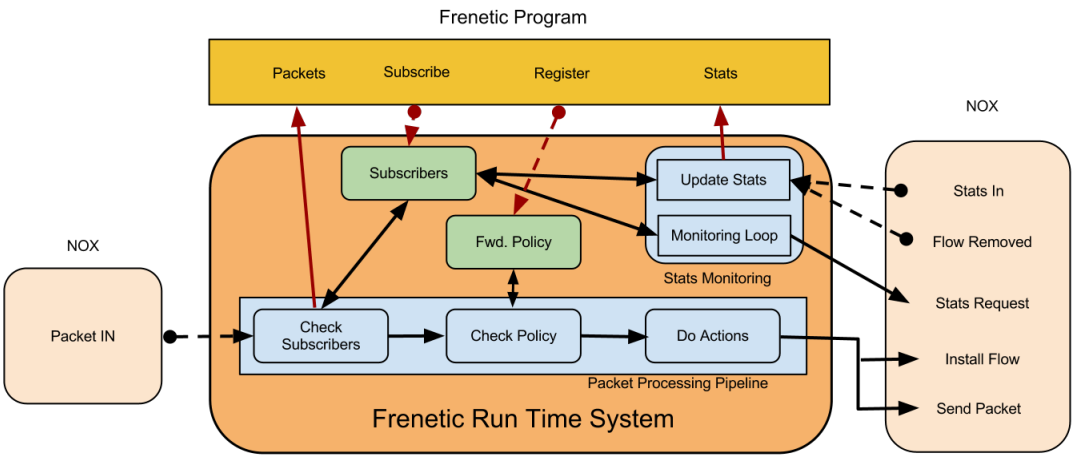

Figure 4: Frenetic Run Time System

1. A Packet Processing Pipeline

2. Statistics Monitoring Facility

3. Global Data Structures which are mainly of three types.

   rules They specify the current packet-forwarding policies

   flows They are the low-level rules that get installed on the switches.

   subscribers They are generally represented as a set of tuples of the form (q,e,cs,rs) where 'q' is the query that defines the subscriber, 'e' is the event for the subscriber, 'cs' is the counter for tracking bytes and packets and 'rs' is a set of identifiers for outstanding requests for statistics[18]

When a program starts its execution, the flow table in each switch would be empty and hence all packets are sent to the controller to the packet_in handler. When a

packet is received by the run-time system, it propagates the packet to each subscriber whose query required such a packet to be seen at the controller. Then the run-time collects the list of actions to be performed from all the rules that matched the packet. Only if the packet did not match a registered subscriber, microflow rules are installed on the switches so as to traverse the future packets with the same header fields via the switches itself, while if the packet did match a registered packet subscriber then just the action is taken but the rules are not installed on the switches as the controller expects to see such future packets. For statistic subscribers, the run time executes a loop that waits until the interval for a statistic subscriber elapses. At that point, it goes through the 'flows' set and requests for the counters maintained for bytes and packets from each switch level rule whose pattern matches the statistics query. The 'stats_in' handler receives the asynchronous replies which then adds the packet and byte counters received to the particular counters maintained for that subscriber. When the set of outstanding requests become empty, the run-time pushes these counters into the subscriber's event streams, which will then contain the correct statistic information.

3.3 Cost of a Frenetic Query

One of the problems in using declarative languages is that the higher level abstraction provided by them does not accurately give the users the cost involved in terms of time-complexity in using such higher level abstraction. But Frenetic avoids this by giving

guidance to the programmers on cost of the programming constructs. The cost of executing a Frenetic query is measured in terms of *Microflows*. A *Microflow* is a set of related packets that have identical header fields and arrive at the same switch. In case of a statistic query like that of getting 'size' or 'count' of packets at most one packet will be diverted to the controller. Hence the cost on the controller is to process that one single packet per flow. But in case of packet queries every packet might have to go to the controller and the limit on the number of packets that would be processed by the controller is the 'Limit' clause which would be mentioned in the query and determines the number of packets to be sent to the controller.

These functionalities of Frenetic has made it robust, compact and a user-friendly programming language compared to NOX.

4 DDoS and its Prevention Mechanisms

4.1 Distributed Denial of Service Attacks

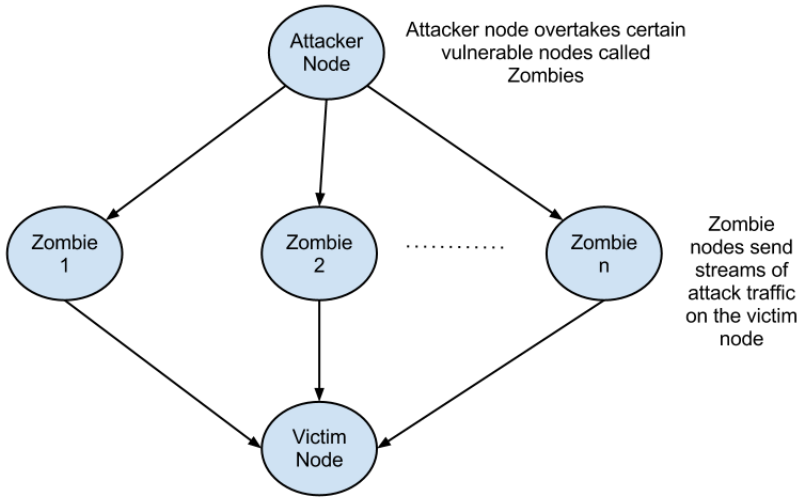

Figure 5: Typical DDoS Attack

The internet was originally designed with the intention of providing openness and scalability; and very less concern was given on security which has led to some vulnerabilities with respect to its security[23]. There is no support in the IP layer to verify a source's authorization to access a service. Packets are delivered to the destination at any cost regardless of who the packet is coming from and it is only up to the server at the destination to decide whether to accept and service these packets. This has led

to a class of security problem called Denial of Service attacks where the vulnerability or the weak point of the system is the system itself. A Denial of Service (DoS) attack aims to deny access by legitimate users to shared services or resources.[14]

An example of a DoS attack is 'ping-of-death' in which the attacker sends large ICMP ping packets that is fragmented into multiple data-grams to a target system, which can cause certain Operating Systems to crash, freeze or reboot due to buffer overflow. A Denial of Service attack can be launched in two forms. In the first form, the attacker sends carefully crafted packets which exploits a software vulnerability of the targeted system and crashes it as in the case of the ICMP ping packets. In the second form, large volumes of useless traffic occupy all the resources of a targeted system and prevents legitimate traffic from getting serviced. [23]

When the DoS attack is being targeted from multiple sources where-in usually a single attacker overtakes certain vulnerable systems and without their approval starts using them for his malicious intent, it is called a Distributed Denial of Service Attack(DDoS). In case of DDoS attacks, the attacker need not have to exploit a security hole at the target to cause problems and there is almost nothing that the victim can do to protect itself which is what makes these class of attacks dangerous. There has been no protection mechanism or a protection model which has proven to be completely efficient against DDoS attacks. [14] They are weapons of mass destruction and usually can incapacitate the targeted victims causing huge productivity and revenue loss especially when the victims turn out to be large businesses.

A DDoS attacks generally has a two-fold impact. One is the consumption of all the resources of the targeted system due to which the victim node starts dropping the packets from both the legitimate and the malicious nodes to inform them to reduce their rate of sending. The legitimate nodes reduce their sending rate while the malicious nodes would not, thus exhausting the CPU and memory resources of the victim node. The second impact is exhausting the network bandwidth which is detrimental from the entire network point of view. In this case not only is the victim node impacted, but the entire network could go down as legitimate traffics which might share the same congested links start getting dropped if there is no clear method to distinguish malicious nodes from the legitimate ones. A DDoS attack is generally measured by its attack power which consists of two parameters. The first parameter is the traffic volume which is nothing but the number of packets in a given interval of time flowing through particular link. The second parameter is the resources consumed per packet which can be denoted by the CPU time consumed or the memory resources used to process a single packet. [23]

Though the traditional security technologies against DDoS such as firewalls and intrusion detection systems prove to be important components of the DDoS defense scheme, in order to completely eliminate the possibility of attack, the network architecture is vital too. Some of the important design principles of today's internet architecture which have resulted in loop holes and thus have been the causes of DDoS attacks are:

1. Simple Core Network and Complex Edge of today's internet has resulted in the core to be too naive, which only needs to deliver IP Packets without needing to understand about the network layer services.[23]

2. Fast Core Networks and Slow Edge networks which has created a drawback that traffic from high capacity core links can overwhelm the low capacity edge links when many sources want to talk to the same edge node which is the case in DDoS attacks.[23]

3. Decentralized Internet Management has resulted in lack of central control of the internet and the DDoS Defense schemes have to be deployed at various locations in a complicated distributed manner to be effective.[23]

With Openflow architecture, the above drawbacks of today's internet are mitigated as it provides central management; the core network is no longer simple as the entire brain of the network is at the core now and the edge of the networks are no longer slow as the switches only need to forward the packets and need not do complex computations. Through our approach we try to convey that using the Openflow architecture, detection, mitigation and prevention of DDoS can be handled better as there is a central controller which can manage the switches which are in its range and using Frenetic as the Network Programming Language makes it much more efficient and convenient in defeating DDoS attacks.

4.2 Methods of DDoS Attacks

1. Protocol Based Bandwidth attacks: These attacks try to exploit specific weaknesses of the Internet Protocols. SYN flood and ICMP flood attacks are good examples of such kind of attacks. For example, in SYN flood attacks the attacker makes use of the vulnerability of the three way handshake of TCP by sending non-existing source ip-addresses in the SYN packets, so that these requests fill up the memory stack of the server and disables the services of the system.

2. Application based attacks: In these types of attacks, the attacker amplifies his attack by forcing the target to execute expensive CPU and memory operations, as in the case of HTTP flood and SIP flood attacks. For example, an attacker can exploit the application of search engines by sending huge number of queries which forces the website to perform intensive resource consuming operations and leaves few resources to serve legitimate users.

3. Distributed Reflector attacks: Here, the attacker tries to obscure himself by relaying his attacks through third party nodes, called reflectors. A Distributed Reflector attack usually comprises of three stages. The first stage is similar to any DDoS attack where-in the attacker gains control of certain vulnerable nodes called zombies. In the second stage, the zombies are ordered to send spoofed traffic with victim's ip address onto the third parties which are the reflector

nodes. In the third stage, the reflectors are controlled to send the reply traffic to the victim nodes, which constitute the actual DDoS attack. The relay of the attacks through innocent third party reflectors makes the source trace back of such attacks extremely difficult.

4. Infrastructure attacks: These attacks aim to disable the services of critical components of the Internet and potentially affect the whole of internet . An example of such attacks is the attack on DNS Root Servers [9]. DDoS attacks on such root servers can exhaust both their host and networking resources which can disrupt all the internet services which depend on such root servers

4.3 DDoS Defense Scheme

1. Attack prevention: It aims to stop the attack before it reaches the target. Though not easy to specify a filtering rule which can differentiate spoofed traffic from legitimate traffic, IP spoofing can be solved by deploying various packet filtering schemes.[23].

2. Attack detection: It aims to detect attacks as they occur and are important with regard to further action that needs to be taken. Attack detection is required so that if a target can detect an attack before it happens, the actual damage and wastage of network bandwidth can be avoided and more so the legitimate users in the network can be protected. [23]

3. Attack source identification: It aims to locate the attack sources regardless of whether the source address field in each packet contains the correct or erroneous(spoofed) information. Once the attack detection phase is over, the IP attack traffic should be traced back to its source, which is taken care in this phase. [23]

4. Attack reaction: It tries to eliminate the effects of an attack and filter the attack traffic without disturbing the legitimate traffic. Attack reaction must minimize the loss caused due to an attack by deploying a reaction scheme when the attack is underway. [23]

4.4 Pushback Scheme of DDoS Defense

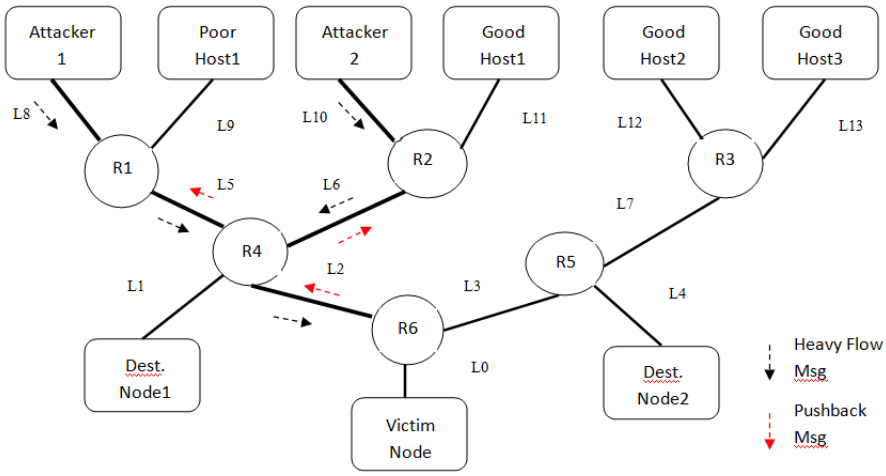

Figure 6: Illustration of Pushback[Ioannidis and Bellovin]

Pushback is a network based solution to prevent DDoS attacks. It utilizes 'Aggregate

based Congestion Control' at each router. The routers identify aggregates responsible for congestion and drop them preferentially. Then the router sends the pushback messages to the upstream routers indicating them to preferentially drop the packets too from the aggregates which are causing congestion. Let us consider Figure6 to illustrate what would be happening in a DDoS attack . Here, each circle depicts a router and the black dotted arrows indicate the links through which the DDoS attack is occurring (and the red dotted arrows indicate the pushback messages which we will see later). As per the figure, heavy traffic(which might be due to the DDoS attack) is flowing through Attacker1-R1-R4-R6-VictimNode and Attacker2-R2-R4-R6-VictimNode. The rest of the links have either good or poor packets flowing through them. A good packet is a packet which is non-anomalous but shares the same congested links in which the attack is taking place. A poor packet is also a non-anomalous packet in transit to the victim node(i.e. shares the same signature as that of the attack) and there are chances that such packets get dropped when DDoS defense is deployed along with the maicious packets. Some of the links can be unaffected like link L4. While some other links like GoodHost1-R2(link L11) is though not affected, the traffic through this link might be flowing to the victim node. High chances are there that such traffic too will get affected though they are not malicious. There might be some other links like L6 which is congested and carrying packets intended for Destination Node1 but as it is a congested link, even such good packets get adversely affected.

In the detection phase of the pushback scheme, R6 detects the aggregates that are

causing a DDoS attack and informs its upstream router R4 with a pushback message, telling it to preferentially drop or rate-limit traffic destined to the victim node. When packets arrive at R6 they are going to be dropped anyway, so they might as well get dropped at R4 itself. R4 in-turn sends pushback messages to its upstream routers R1 and R2 respectively asking them to rate-limit the bad traffic and allowing some of the poor and more of the good traffics.[19]

The design decision in pushback is to separate the rate-limiting and packet dropping functionality. [19] When congestion of links are detected, the router checks for anomalous aggregates. If an anomaly is detected based on congestion signature(a congestion signature is something like a victim node's ip-address), then rate-limiting is done on such aggregates. The packets from those aggregates which are not rate-limited are sent to the output queue of the router as usual. The packets which are dropped both by the rate limiter and the output queue are sent to the pushback daemon for an analysis on the attack estimated to be happening. The pushback daemon checks the dropped packets and updates the congestion signature and adjusts the rate limit based on how much congestion is still detected in the links. For e.g., even after setting a rate limit, if congestion is detected in the links, then either the rate limit has to be increased or rate limits have to be applied on other aggregates which might be anomalous. The router then sends 'Pushback messages' to the upstream routers in order to let them know that due to a particular signature, congestion is happening at the links and hence they in turn have to adapt accordingly by limiting their own

rates and dropping certain packets.

5 Implementation of Pushback using Frenetic

We have tried to implement a naive version of the Pushback Scheme of DDoS Defense mechanism and evaluate Frenetic based on this implementation. We use mininet which is a virtual network simulator to test our implementation.

5.1 Frenetic System Bring Up

The entire Frenetic system along with the mininet virtual network simulator is on a virtual machine[2]. The image 'frenetic.9.27.2010.ovf' can be started using a virtual machine such as VMware or Oracle VM Virtual Box. The script 'setup-vm-ssh.sh' is run with a port number greater than 1024 (say 9992) to enable port forwarding from a user-defined localhost port(9992) to the test bed Virtual Machine (VM) port 22.It also creates another script 'frenetic-ssh.sh', which on running opens up an ssh session with the Frenetic System i.e. the VM. When, first logged into the VM, the Frenetic code can be checked out and installed, using a htpassword file by getting access permission to the svn repository from concerned people. Usually while testing, two ssh sessions are created to the VM, one in which the Frenetic Program or the application would be running and the other in which the mininet network simulator runs the network topology.

5.2 Mininet

This is the Network Simulator which we have used for our testing purposes. It creates scalable and customizable Software Defined Networks(SDN) on a single PC by using Linux Processes in network namespaces. [4] The advantage of using mininet as the network simulator for Openflow applications is that complex topology testing can be done in a simple, efficient and inexpensive manner. Moreover the code developed and tested on the mininet network, can be moved to a real system with no or minute changes, for real-world testing, performance evaluation and deployment.

5.2.1 Working of Mininet

Mininet uses process-based virtualization to run the entire network consisting of hosts, Openflow switches and controller on a single OS kernel. Network namespace is a lightweight virtualization feature that provides individual processes with separate network interfaces, routing and ARP tables. It uses virtual ethernet to connect switches and hosts. An example which creates a network in a single command of mininet is shown below:

mn –switch ovsk –controller nox –topo tree, depth=2, fanout=8 –test pingAll[4] The above command starts a network topology in tree shape which has a depth of 2 and fanout of 8 i.e. 9 switches connected to 64 hosts. It uses Openflow vSwitch switches and the NOX controller and does a pingAll to test the connectivity between every pair of nodes.

5.3 Implementation

In our case, the Frenetic code running on the OpenFlow Controller acts as both the pushback daemon and the rate limiter. Pushback messages are not sent by the router as the switches do not communicate and all the communication is done by the Central Controller in case of OpenFlow networks. Thus, the pushback messages in this case are the rules that would be installed by the controller on the switches i.e. the communication between the controller and the switches.

The code functionality is as described below: A node receiving data greater than the threshold amount (defined to be 5 MB of data) is identified as part of the congestion signature, that is such a node is considered to be under threat. Any node is allowed to receive the normal amount of 0.5 MB of data. When nodes start receiving larger than 0.5 MB of data, the amount exceeded is kept track of. When this exceeding amount surpasses the threshold of 5MB, it is assumed that there has been a congestion which might be due to a DDoS attack. After identifying the congestion in the network, the switches between the malicious node/s and the victim node is obtained by utilizing a simple network query as below, which returns the size of the data being sent from the malicious source node/s to the destination node and the switches through which the data is getting traversed :

$sizes = (Select('sizes')*$

$GroupBy(['srcmac','switch','dstmac'])*$

$Every(polling\_interval) >> Identity())$

Once the attack is detected, there are two phases. One is the detection of anomalous nodes which are the cause of attack and hence are to be removed, taking care that the nodes sending good/poor packets aren't removed. Though, it is hard to come up with such a scenario, a best case effort is currently used. The second phase involves the application of negation rule of such anomalous nodes on switches in a pushback manner from the switch connected to the victim node to the switch next to the malicious node/s. The negation rule prevents any packet identified to be arriving from a malicious node to proceed further from that switch and is dropped. The first phase is carried out using a greedy approach i.e. the node sending the highest data is removed first and then the second highest data sending node is removed. If two or more nodes are sending the same amount of data, then one of the nodes out of them is randomly removed.

In the second phase, the controller first installs this negation rule on the first switch in the switch list which would be the switch connected to the victim node. Then in the subsequent instance of iteration (or polling interval of 10 and 20 second is taken in our implementation), the negation rule is installed on the second switch in order of the list and so on until all the switches between the victim node and the malicious node/s are applied with proper rules in a pushback manner so that congestion is prevented in the network and hence DDoS attacks are avoided. This is the preferential dropping phase of the Pushback. The negation rule is applied through a simple Frenetic statement shown below where in a ' minus badsrcmacs_fp' is done and 'badsrcmacs_fp' is

the filter pattern which stores all the malicious nodes calculated as explained earlier:

$nr[switch] =$

$[Rule(rule.pattern - badsrcmacs\_fp, rule.actions)$

$for\ rule\ in\ switch\_rules]$

Below is a description of each function used in our implementation:

1. main()

   The main method first imports the learning_switch() module and forms the appropriate flow tables at the switches and to form this flow table the classic learning switch policy is employed where in the switches dynamically learns the association between hosts and ports. It then calls the ddos_filter() method which would modify the rules formed at the switches if a DDoS is detected to be happening such that all the packets from the anomalous nodes are negated from the rules and hence gets dropped at the switches.

2. ddos_filter()

   It takes the rules formed by the learning switch module as an argument and modifies it by negating the anomalous nodes from the rules installed at the switches. To do this, it calls two functions namely ddos_attackers() and drop_packets_using_pushback()

3. ddos_attackers()

   This function executes the network query and collects the statistic - 'size' or

the amount of data getting traversed between a source node and a destination node via the switches at every polling interval. This polling interval variant can be altered to get network information from the database at regular intervals. This query is just like a sql statement which obtains the size statistic information. After obtaining the 'size' of the data getting traversed between every pair of nodes in the network, it passes this information to the function detect_anomalies()

4. detect_anomalies()

   In detect_anomalies() function, the statistic information of the size of data is made use of to determine if there is any DDoS happening in the network. For our implementation purposes we have made use of a threshold amount of data = 5MB, which every node can handle. When a destination node starts receiving data of size greater than this threshold amount, it is assumed that a DDoS might be happening or it might be due to a temporary spike in the traffic. Once, it is detected that the threshold value of the data limit is exceeded on a particular node, we list out all the source nodes which are sending data to such a node and then get the highest data sending node as the first anomalous node. If two or more nodes are sending the same amount of data and both are the highest traffic sending nodes, then one of the nodes is randomly chosen to be an anomalous node in a naive way! Even after removing the anomalous node(i.e., the highest traffic sending node), if the destination node is still receiving data which exceeds

the threshold value, the process is repeated on all other source nodes in a greedy approach(i.e., from the highest to the lowest data sending node) until the data received by the destination node is within the threshold. This list of anomalous node/s formed in a greedy manner is returned to ddos_attackers() function. Once the ddos_attackers() gets the information about the anomalous node/s, it sends this to ddos_filter() method.

5. drop_packets_using_pushback()

   The ddos_filter() function on receiving the anomalous node list, sends it to drop_packets_using_pushback() function so that it can negate the anomalous node/s obtained from the rules installed on the switches in a pushback manner. This is the 'Preferential Dropping phase of the Pushback' mentioned by Ioannidis et. al. where-in first the anomalous node/s are removed from the flow table of the switch present next to the victim and then successively on all the switches one after the other from the victim node to the malicious node/s.

6. srcmac_in_fp()

   This is a convenient function called by the drop_packets_using_pushback() function, which when provided with a list of anomalous nodes, returns a filter pattern which matches all those anomalous nodes.

6 Evaluation of Frenetic1.0

We try to evaluate Frenetic and our Pushback implementation based on two factors. One is by taking the Input/Output Graph using wireshark, which tells us how quickly Frenetic can detect and mitigate DDoS attacks. The second factor, though still a work in progress is more difficult to evaluate and would be covered in the future, which is to what extent only the malicious packets get dropped and the normal flow of good and bad packets is continued. We have conducted our testing of the first factor using 'mininet' network simulator; the network topology constructed for our evaluation is the same as in Figure6, whose mininet code can be found in Section 'A' of the Appendix.

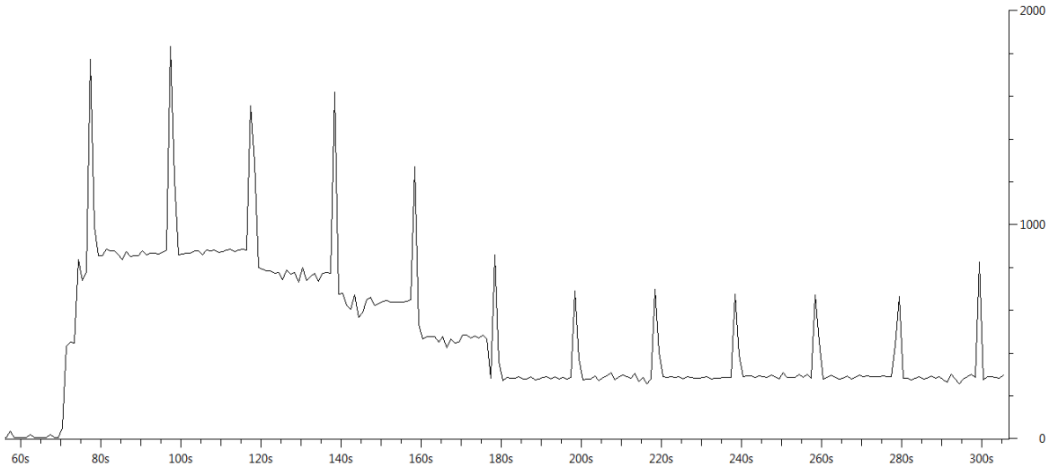

Figure 7: Input/Output Graph obtained for polling interval=20s

We have obtained two Input/output Graphs as part of our evaluation for two different polling intervals of the network state, 20 and 10 seconds respectively. Let us consider

the graph in Figure7,when obtaining which, the polling interval to obtain the network states and statistics was kept as 20 second. Here, the Y-axis represents the number of packets per second and the X-axis represents time in seconds. In this graph, at around 70 second, all the nodes (as shown in fig. 6) are sending data to the victim node, which means the two anomalous nodes have started their attack. At about 120 second, DDoS detection happens and the node(out of the two) which is sending higher data (this is the greedy approach) is negated in the forwarding rule installed on the switch which is connected to the victim node i.e. R6 in figure6. This is the pushback beginning phase which is propagated upstream next to the switch R4 and so on. At about 140 second, the second anomalous node is detected and it is likewise removed. Finally the load stabilizes throughout the network at around 160-180 second. Thus, our implementation takes around 110 seconds for the total detection and mitigation of the attack in a Pushback manner throughout the network when the polling time is kept as 20 seconds.

In the second graph shown in Figure8, the polling interval was kept as 10 second and all other aspects were maintained the same as in the first scenario. In this case, at about 45-50 second all the nodes(as in figure 6) are sending data to the victim node including the malicious nodes, which indicates the beginning of the attack phase too. At about 80 second, DDoS detection happens and out of the two malicious nodes, the node which is sending the higher data is disconnected by applying a negation rule on R6. Then, at about 100-105 second, the second node is removed and by 120 second

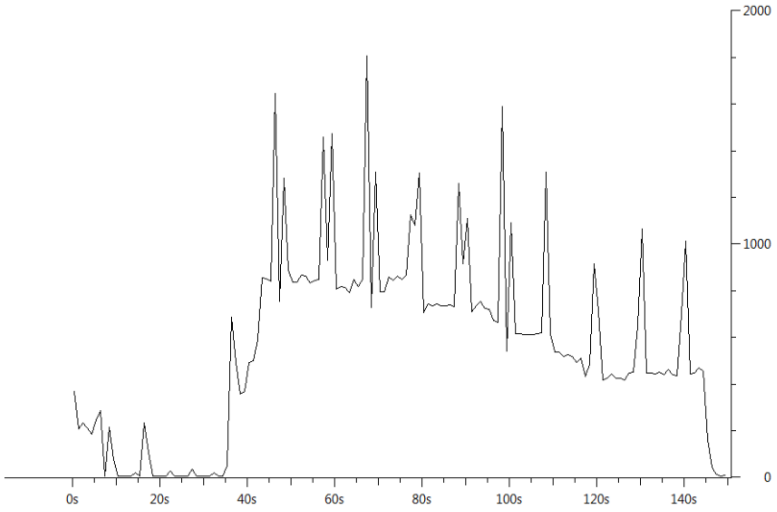

Figure 8: Input/Output Graph obtained for polling interval=10s

the load stabilizes. Hence, it has taken around 70 seconds for DDoS detection and mitigation when the network polling time is around 10 seconds. In a real scenario, with more frequent network queries, i.e. less than 20 or 10 seconds, there might be more rapid detection and mitigation of attacks.

Moreover in a real scenario, with a centralized architecture as in Open flow, the central network controller can co-ordinate with its switches and take the right decision in a minimal time provided correct DDoS detection happens, by employing strategies like blocking the malicious node at the ingress router itself rather than going for a Pushback way of handling. But our approach of Pushback takes care that even if the detection went wrong (false positive or due to spike in the network) and was not actually a DDoS attack, then during the phase of the pushback, the negation rule on

the wrongly assumed malicious node can be removed and simultaneously ensure that the entire network is safe.

We have taken a first step in our DDoS Prevention Scheme. DDoS Defense in real systems have to cope with complicated scenarios like deception by the malicious nodes through spoofing of their ip address, but still our implementation shows that defense can be erected quickly through Software Defined Networking.

Our implementation shows that with a Functional Reactive Network Programming Language like Frenetic on Openflow architecture provides an easy, flexible and efficient solution to handle DDoS attacks. The entire implementation has taken around 200 lines of Frenetic code embedded in Python. The major advantage found in the language as such is its Network queries which gives the details about the network and makes the controller omniscient without having to bother about the hassles of a distributed system. The mathematical set like operations and primitives further enrich and enhance a programmer's ability to have control over what he wants in a declarative fashion rather than bothering about the book-keeping and worrying about the lower level abstractions of the network.

7 Future Research

The next step with this implementation is to verify that the number of good/poor packets getting dropped would be substantially less and the actual anomalous packets would justly get dropped, which is the expected behavior of a good DDoS Prevention mechanism. With mininet test simulation, currently the link bandwidth cannot be configured. During our testing phase, this has resulted in the link bandwidth never getting congested and hence a DDoS attack is assumed to be happening only when a particular node starts receiving data that has exceeded its capacity i.e. the threshold value. For proper evaluation of the accuracy of filtration, congestion on the links have to be introduced and dropping of good/poor packets tested. In either case of anomalous nodes getting detected properly or not, the dropping of good/poor packets should be minimal and ideally zero.

Also, in our implementation, the criterion which decides whether packets from a good/bad node is dropped is only the negation rule applied on the switches based on the anomalous node detection. This means the detection phase is of at most importance as otherwise, it will lead to a lot of good/poor packets getting dropped. For that, we are expected to change the naive greedy approach we have used in the anomalous node detection and use more sophisticated approaches like groups of flows based on weights.[24]

The current prevention mechanism assumes a naive congestion signature as the aggregate is taken as the address of the source node itself. As the source address could

be spoofed easily, alternative congestion signatures have to be adopted. Implementations based on packet-marking seem well suited to Software Defined Networking approaches.[24]

Frenetic works on the principle of 'see every packet' abstraction, though it is costly in terms of the controller processing if every packet has to be seen. Future work also lies in this area that how efficiently packets can be looked at and DDoS detected in their earlier stages itself and thus prevent attacks from aggravating. Commercial Technologies based on Openflow architecture are emerging, where-in each packet can be handled at line speed. Hence, improvement in this direction is certainly a possibility which gives tremendous power and opportunity to a Network Programmer when he can do his programming and deploy defense by looking at the contents of every packet. This also leads to a possibility of Deep Packet Inspection(DPI) using Frenetic.

Many areas of improvement are currently underway with Frenetic 2.0 such as making the language proactive rather than reactive and also improving the run-time system. These changes would add to the present flexibility of Frenetic and give better control to a Network Programmer to deploy Software Controlled Defense Schemes on future networks. [24]

8 Conclusion

Our Thesis work is an implementation of DDoS Defense mechanism - Pushback, using Frenetic. We have tried to evaluate Frenetic based on its swiftness in detection and mitigation of DDoS attacks which gives an initial idea on how it could further be utilized for easy, efficient and robust Network Programming and efficient security deployment on future networks. It is planned to further enhance the security applications and DDoS prevention mechanisms using Frenetic as part of future work. Improvements in the language with newer versions, Frenetic as a language for Software Defined Network Defense on OpenFlow network architecture looks promising indeed!

References

[1] Frenetic network programming language. http://www.frenetic-lang.org.

[2] Frenetic testbed 9.27.2010. https://frenetic-lang.org/mediawiki/index.php/Frenetic_Testbed_9.27.2010.

[3] geni: Exploring networks of the future. http://www.geni.net.

[4] Mininet: rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[5] Open networking foundation. https://www.opennetworking.org.

[6] Openflow and software defined networks. www.openflow.org/documents/OpenFlow_2011.pps.

[7] Openflow switch consortium. http://http://www.openflow.org.

[8] Martn Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick Mckeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *In SIGCOMM Computer Comm. Rev*, 2007.

[9] S. Cheung. Denial of service against the domain name system. *Security Privacy, IEEE*, 4(1):40 – 45, jan.-feb. 2006.

[10] CiscoSystems. Defeating ddos attacks.

[11] Christos Douligeris and Aikaterini Mitrokotsa. Ddos attacks and defense mechanisms: classification and state-of-the-art. *Computer Networks*, 44(5):643 – 666, 2004.

[12] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. *SIGPLAN Not.*, 46(9):279–291, September 2011.

[13] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[14] Virgil D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Software Eng.*, 10(3):320–324, 1984.

[15] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, October 2005.

[16] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[17] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 9–9, Berkeley, CA, USA, 2001. USENIX Association.

[18] Walter Robert J. Harrison. Frenetic: A network programming language. Master's thesis.

[19] John Ioannidis and Steven M. Bellovin. Implementing pushback: Router-based defense against ddos attacks. In *In Proceedings of Network and Distributed System Security Symposium*, 2002.

[20] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Aggregate-based congestion control. *Computer Communication Review*, 32:530614, 2002.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[22] Roger M. Needham. Denial of service: an example. *Commun. ACM*, 37(11):42–46, November 1994.

[23] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM COMP. SURV*, 39(1), 2007.

[24] Sumanth M. Sathyanarayana and Jonathan M. Smith. Software defined network defense. Paper Submitted to HOTSDN Workshop 2012.

[25] J.M. Smith and S.M. Nettles. Active networking: one view of the past, present, and future. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 34(1):4 –18, feb. 2004.

[26] Jonathan M. Smith. A time capsule for sdn. Paper Submitted to HOTSDN Workshop 2012.

[27] Usman Tariq, Yasir Malik, Bessam Abdulrazak, and Manpyo Hong. Collaborative peer to peer defense mechanism for ddos attacks. *Procedia CS*, pages 157–164, 2011.

A Mininet Network Topology Code

```
class MyTopo( Topo ):
    def __init__( self, enable_all = True ):
        "Create custom topo."
        # Add default members to class.
        super( MyTopo, self ).__init__()
        # Set Node IDs for hosts and switches
        badHost1 = 1
        poorHost1 = 2
        badHost2 = 3
        goodHost1 = 4
        goodHost2 = 5
        goodHost3 = 6
        router1 = 7
        router2 = 8
        router3 = 9
        router4 = 10
        router5 = 11
        router6 = 12
        destHost = 13
```

```
destHost = 14

destHost = 15

# Add nodes

self.add_node( badHost1, Node( is_switch=False ) )

self.add_node( badHost2, Node( is_switch=False ) )

self.add_node( poorHost1, Node( is_switch=False ) )

self.add_node( goodHost1, Node( is_switch=False ) )

self.add_node( goodHost2, Node( is_switch=False ) )

self.add_node( goodHost3, Node( is_switch=False ) )

self.add_node( destHost, Node( is_switch=False ) )

self.add_node( destHost1, Node( is_switch=False ) )

self.add_node( destHost2, Node( is_switch=False ) )

self.add_node( router1, Node( is_switch=True ) )

self.add_node( router2, Node( is_switch=True ) )

self.add_node( router3, Node( is_switch=True ) )

self.add_node( router4, Node( is_switch=True ) )

self.add_node( router5, Node( is_switch=True ) )

self.add_node( router6, Node( is_switch=True ) )

# Add edges

self.add_edge( badHost1, router1 )

self.add_edge( poorHost1, router1 )
```

```
        self.add_edge( badHost2, router2 )
        self.add_edge( goodHost1, router2 )
        self.add_edge( goodHost2, router3 )
        self.add_edge( goodHost3, router3 )
        self.add_edge( router1, router4 )
        self.add_edge( router2, router4 )
        self.add_edge( router3, router5 )
        self.add_edge( router4, router6 )
        self.add_edge( router5, router6 )
        self.add_edge( router4, destHost1 )
        self.add_edge( router5, destHost2 )
        self.add_edge( router6, destHost )
        # Consider all switches and hosts 'on'
        self.enable_all()
topos =  'mytopo': ( lambda: MyTopo() )
```

B Frenetic Code for Pushback Defense

```
from collections import defaultdict

from nox.coreapps.examples.frenetic_lib import *

from nox.coreapps.examples.frenetic_net import *

import time

import hosts as hosts

cusum_threshold = 5242880

polling_interval = 10

normal1 = 500000

#This global counter var is introduced so as to keep track

#of the switches on which the negation rules are to be applied

counter = 0

#list which stores the switches on which negation rule has to be applied

srcmacs2 = []

#Another dictionary to trace the victim node's attackers and the

#packet size they are sending! (attackerNode,victimNode):size

vicdic={}

#Flags for convenience

done_topo = False

badmacSet = False
```

```
victimFlag=False
doneFlag=False
#List of sets of the form: [(attackerNode, victimNode)]
#It is used in comparison while doing a return in detect_anomalies fn.
vicSet=[]
vic_keys=[]
def detect_anomalies((start_time, stats), (cusum, k)):
    global victimFlag, doneFlag
    global vicSet
    global vicdic
    global vic_keys
    vic_val=[]
    for srcmac, total in stats.iteritems():
        print ("SRCMAC: ",srcmac[0], "SWITCH: ", srcmac[1] , "DSTMAC:", src-
mac[2], " TOTAL: ",total)
        if total > 0 and stats != {}:
            print("TOTAL:",total)
            cusum[srcmac] = max(0, cusum[srcmac] + total - normal1)
        elif total < 0 and stats != {}:
            cusum[srcmac] = max(0, cusum[srcmac] - total)
            print("CUSUM: ",cusum)
```

```
for each in cusum:
    print ("CUSUM AFTER: ", cusum[each])
#Form another dictionary with src and dstmac as the key
sddic={}
for each in cusum:
    sddic[(each[0],each[2])]=cusum[each]
    print("SDDIC:",sddic)
#Form a dictionary with dstmac or the node being attacked as the
#key and form a list of all the attacking nodes
destdic={}
l=[]
for each in sddic:
    destdic[(each[1])]=0
for each in sddic:
    l.append(each[0])
    destdic[(each[1])]+=sddic[each]
print("DESTDIC:",destdic)
    vic=None
for victim,tot in destdic.iteritems():
    if tot > cusum_threshold:
        vic=victim
```

```
if victimFlag == False and doneFlag == False:
    for each in sddic:
        if each[1]==vic:
            vicdic[each]=sddic[each]
            victimFlag=True
    print ("VICDIC: ",vicdic)
#A list containing the values of vicdic is formed: vic_val
vic_val=vicdic.values()
#Sort the vic_val list so that when a pop is done
#the highest value comes out first
vic_val.sort()
for each in vic_val:
    for every in vicdic:
        if vicdic[every] == each:
            vic_keys.append(every)
print("Vic_keys:",vic_keys)
total=0
if victimFlag==True:
    for each,sum_tot in sddic.iteritems():
        if each not in vicSet and each[1] == vic:
            total+=sum_tot
```

```
            print("TOTAL:", total)
            if total > cusum_threshold:
                victimFlag=True
                #Form the list vicSet which will contain the attacking nodes
                #and the victim node in a set form! This is the least num of
                #attacking nodes which have to be removed to maintain normalcy.
                key=vic_keys.pop()
                print("KEY:",key)
                vicdic.pop(key)
                vicSet.append(key)
                print("VICKEYS:", vic_keys)
            else:
                doneFlag=True
        return ([srcmac for srcmac, sum in cusum.iteritems()
            if (srcmac[0],srcmac[2]) in vicSet],
            (cusum, k+1))
def srcmac_in_fp(srcmacs):
    return or_fp([srcmac_fp(srcmac) for srcmac in srcmacs])
def drop_packets_using_pushback(srcmacs, rules):
    global counter
    print("Inside drop_packets")
```

```
print("SRCMACS: ",srcmacs)
global badmacSet
global srcmacs2
srcmacs1 = []
list_sw=[]
for each in srcmacs:
    srcmacs1.append(each[0])
    #Append the switches to this new list called list_sw[]
    if each[1] not in list_sw:
        list_sw.append(each[1])
list_sw.sort()
list_sw.reverse()
print ("The list of switches is: ",list_sw)
if srcmacs1 != []:
    badmacSet = True
badsrcmacs_fp = srcmac_in_fp(srcmacs1)
print ("BADSRCMAC", badsrcmacs_fp)
if srcmacs1 != [] and badmacSet == True:
    if counter < (len(list_sw)) and list_sw != [] :
        srcmacs2.append(list_sw[counter])
        counter+=1
```

```
    print("COUNTER: ",counter,"SWITCHES on which - to be applied:",srcmacs2)


    nr = dict()
    for switch,switch_rules in rules.iteritems():
        if switch in srcmacs2 and srcmacs != []:
            nr[switch] = [Rule(rule.pattern - badsrcmacs_fp,rule.actions) for rule in
switch_rules]
        else:
            nr[switch] = [Rule(rule.pattern, rule.actions) for rule in switch_rules]
    print ("The rules are: ", nr)
    return nr
def ddos_attackers():
    #A N/w query is done (say every 10 sec)to obtain the size
    #of the data getting sent from a particular srcnode to a destnode!
    sizes = (Select('sizes') *
    GroupBy(['srcmac','switch','dstmac']) *
    Every(polling_interval) >>
    Identity())
    sizes >> Print("SIZES: ")
    anom=[]
    if sizes != {}:
```

```
        anom=(Snapshot(Hold(0, Seconds()), sizes) >>
        LoopPre((defaultdict(int), 0), Lift(detect_anomalies)))
    return anom
def ddos_filter(rules):
    ddos_nodes = ddos_attackers()
    eventOffenders=(Smash([],{},ddos_nodes,rules))
    return (eventOffenders >>
    Lift(lambda (srcmacs, rules): drop_packets_using_pushback(srcmacs, rules)))
def main():
    import learning_switch
    ddos_filter(learning_switch.rules()) >> Register()
```