Big Data Analytics for Real-time DDoS Detection and Mitigation in SDN

Rahul Prabhu
CISE Dept.
University of Florida
rprabhu@ufl.edu

Sanil Sinai Borkar
CISE Dept.
University of Florida
sanilborkar@ufl.edu

Sile Hu
CISE Dept.
University of Florida
husile@ufl.edu

Umar Majeed
CISE Dept.
University of Florida
umarmajeed@ufl.edu

***Abstract*—Distributed Denial of Service (DDoS) is a major security threat on the Internet and the lack of real-time DDoS detection and mitigation tools can render servers vulnerable. In this paper, we propose a system to detect and counter DDoS for Software Defined Networks (SDN) that learns from historical DDoS attack data and deploys counter measures in real-time. The decoupling of the control and the data plane in SDN provides a centralized control, which allows controllers to alter the routing decisions on the go. The accuracy of the proposed system exceeded 97% on the CAIDA DDoS attack dataset, and the latency added by the extra computation was found to be negligible. As SDN gains popularity, the system that proposed here can be used to detect and counter DDoS attacks targetting real world networks.**

***Index Terms*—Software Defined Networks; Big Data Analytics; DDoS; Classification**

I. Introduction

Distributed denial of service (DDoS) is an attack to consume victims' resources such as bandwidth, memory and TCP connections to make the their services unavailable [1]. Attackers control malicious botnets or utilize defects in communication protocols to launch attacks. Utilizing valid services over the Internet and leveraging amplifying effects of certain protocols, attackers can quickly exhaust victims' bandwidth. Attacking techniques include TCP SYN flooding, ICMP flooding, reflective DNS/NTP/SNMP attacking etc. [2]. Attacks are typically short bursts of heavy traffic which makes it particularly difficult to detect and defend against. The lack of a centralized view in a network adds to the complexity of the problem.

Software Defined Networking (SDN) is an approach to networking where the control plane and the data plane are decoupled [3]. The ability of SDN to filter network packets with the help of the controller, when they enter a network, gives it the ability to implement DDoS detection and countering mechanisms. SDN switches can act as a firewall at the edge of a network, providing security to hosts residing in the network. They also have the ability to dynamically respond to application requirements, optimize the utilization of the network without compromising the QoS which allow it to implement extra defense mechanisms without any significant impact to network performance.

In this paper, we describe a solution to the problem of real-time detection and mitigation of DDoS attacks in SDNs. The fact that the SDN controller has a centralized view of the network and can reprogram network switches on the go makes such a solution possible. The following are some of the features of our solution:

1) SDN-enabled DDoS detection.
2) Use of big data analytics to enable time detection of DDoS attacks.
3) Use of SDNs ability to reprogram networks on the go for real-time mitigation of DDoS attacks.

In traditional networks, every switch decides where an incoming packet has to be routed. In a large network, this translates to lot of entities acting independently which make collective decision making very hard. In SDNs, the routing decisions are made by the controller for each switch. This gives the controller a centralized view of the network and gives it the ability to make collective decisions for the network as a whole [4]. We make use of this unique feature of SDNs to build our system.

When a packet arrives at a switch in an SDN, the switch communicates to the controller about the incoming packet and requests the controller for the action to be performed. The controller then makes the decision on where the packet must be sent and relays this information back to the switch. Here, the SDN controller can do some computation to detect if a packet is a malicious attack packet or a benign packet, and based on this output, instruct the switch to either drop the packet or forward it.

In large networks, the SDN controller may have to deal with thousands of switches and hundreds of thousands of packets at any given point of time. This makes the computation to determine whether a packet is malicious or not infeasible to do in real-time. To address this, we move the computation to a DDoS Detection Engine (DDE) which runs on a cluster of commodity computers. The controller extracts certain features from incoming packets and sends it in batches to the DDE which does the necessary computation for determining if any packet is malicious or not and sends back the decision to the controller. Now, the controller is only left with the task of making the decision of dropping or forwarding the packet, and sending these instructions to the appropriate switch.

The DDE takes in historical DDoS attack data and builds a classification model which classifies an incoming packet as malicious or benign. The process of building a classification model is mostly done offline as it takes in large amounts of data and is difficult to build in real time. However, since this is done very infrequently compared to the classification task itself, it does not affect the performance of the system. The classification task is carried out in real time and is what enables our solution to respond to threats on the fly.

In our experiments, we use an Apache Spark cluster for enabling large scale data processing in the DDE as it gives near real-time performance and provides support for streaming data. We use the Machine Learning library (MLlib) provided with Spark to classify packets as malicious or benign.

Since DDoS is a vast area, building a system that works with all types of DDoS attacks and under all circumstances is not possible. So, we make some simplifying assumptions here. We assume that the controller is secure and not under attack. Though this is a strong assumption to make when dealing with DDoS, this assumption aids in the process of building our prototype without taking the focus away from the main idea which is building a DDE that can run in real time. The DDE learns from historical data and predicts the behaviour of packets and hence the dataset used determines the quality of the prediction. We have used the CAIDA dataset which is a SYN flooding DDoS attack data from a real world network. So, at present our system only detects and defends against TCP SYN flooding. This can be enhanced to detect other type of attacks by using other attack datasets in the training.

The rest of the paper is organized as follows. Section III discusses the System Architecture. Section IV discusses implementation details. Section V discusses the experiments carried and presents the results. Sections VII and VI present our conclusions and some directions for future work in this field, respectively.

II. Related Work

Traditionally, various techniques have been used to defend against DDoS attacks from an independent routers' point of view. These techniques include access control lists, protocol hardening, packet filtering based on IP addresses, TCP and UDP port numbers, and packet lengths and content matching [5]. These mechanisms are effective once the attacking mechanisms are well understood, but suffer from the lack of real-time response and the to be applied on a per-router basis.

SDN provides the ability to have a centralized control and real-time response over a set of routers, and there are a vendors working on integrating DDoS mitigating features into the SDN controllers, such as Brocade [6] and Radware [7]. These applications provide attack detection and control over a whole SDN, but the detection and access rules are still largely packet filtering based.

sFlow-RT by InMon implements the sFlow protocol which acts as a link between the network devices and the SDN flow controller [8]. They use the sFlow analytics engine for DDoS mitigation. The SDN flow controller communicates with the DDoS engine which instructs the SDN controller to drop packets that are malicious.

Lee and Lee worked on detecting DDoS attacks at scale by using a Hadoop cluster for computation [9]. However, they were not able to tackle this in real-time as Hadoop is a batch processing framework. Gavrilis et al. [10] designed a DDoS attack detection system using Radial-basis-function Neural Netowrks (RBF-NN).

Mousavi et al. propose a DDoS attack detection scheme in SDNs using the notion of entropy of window size and thresholds [11]. Their study, however, was able to only detect an attack and there were no mitigation procedures proposed.

For training the data and classification purposes, a good feature set is required which can be derived using various methods. The most promising method was specified by Ganapathy et al. [12] wherein they used an intelligent rule-based attribute selection algorithm to determine the feature set. Seo et al. [13] propose another way of determining the feature set using a clustering-based method.

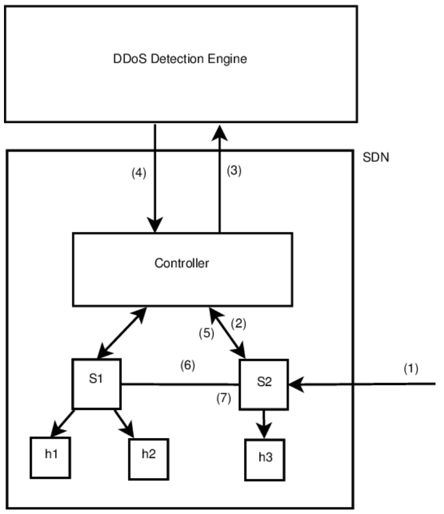


Fig. 1: Architecture

III. Architecture

The architecture of our system shown in Figure 1 comprises of the SDN and the DDoS Detection Engine (DDE). An incoming packet reaches one of the switches of the SDN (1). This switch is called a boundary switch and serves as the entry point to the SDN. Most of the malicious packet filtering happen at such boundary switches thus preventing attacks from impacting other nodes in the network. This switch requests the controller for directions on what to do with the packet (2). The controller extracts packet information from all the packets it gets and streams it to the DDE (3). The DDE receives this stream of packet information and evaluates each packet against the classification model built and gives a decision on whether a particular packet is malicious or not (4). It relays this information back to the controller. Based on the decision of the DDE, the controller sends appropriate flow commands to the switch (5). Depending on the flow command received, the switch either forwards the packet en route to its destination (6) or drops it (7).

The components of our system are explained here:

DDoS Detection Engine (DDE) - The DDE is the main component of the system as it performs the critical task of DDoS attack detection. It is a cluster of commodity machines running Apache Spark that communicates with the OpenFlow SDN controller. The DDE accepts the network packets' information sent by the OpenFlow controller, feeds this information to the already trained classification model and relays the result back to the controller. The DDE performs this computation for every packet information that the controller sends it. Since the DDE has to do high magnitudes of computation, we have deployed the DDE on a cluster of machines in Amazon EC2 [14]. For the computation of such seemingly large magnitudes of data, we have used Apache Spark as it does in-memory computing and is nearly 10 times faster than its batch processing counterpart Hadoop [15]. In addition to this, Apache Spark has support for streaming data which is helps us stream packet information from the controller to the DDE. Another reason for choosing Apache Spark was the availability of Machine Learning libraries (MLlib) bundled with it. MLlib provides implementations of various machine learning algorithms that we use in our experiments [16].

Software Defined Networks (SDN) - Our system requires the underlying network to be an SDN as the SDN controller plays a crucial role in the DDoS attack detection and mitigation process. It acts as a centralized authority that links the DDE and rest of the network. Switches send requests to the controller on encountering new incoming network packets which the controller evaluates with the help of the DDE. Depending on the decision of the DDE, it instructs switches to either drop or forward the packet. The controller does this by sending add and/or modify flow rules to the switches in order to create or update the switch's flow rules. The controller's ability to reprogram the switches in this fashion allows us to implement countering mechanisms in real-time.

Communication Modules - Individual modules are interfaced in a way to reduce latency and enhance performance. A daemon running on the controller that reads the requests sent by the switch, extracts the required packet data from it and write this information to a socket. This streams the data to the DDE which reads this incoming stream using the Apache Spark streaming interface [17]. After the computation, the DDE sends the results back to the controller by writing to a socket. We implement a daemon which listens to a port on the controller for the DDE to send the packet decisions back. The communication between the controller and the switches take place using the open-flow protocol.

IV. System Design and Implementation

The experimental setup consisting of an SDN network simulated using Mininet [18] is as shown in Figure 2 and consists of the following components:

- The local host machine hosting the controller (c0), two hosts (h3 and h4) and one OpenFlow-enabled switch (s2) of a SDN.
- A guest Mininet VM running on the host machine contains two hosts (h1 and h2) and an OpenFlow-enabled switch (s1). This switch *s1* is connected to the remote controller *c0* hosted by the local machine.
- The DDE is setup on an AWS EC2 cluster (Experiment 1) and local machine(Experiment 2).

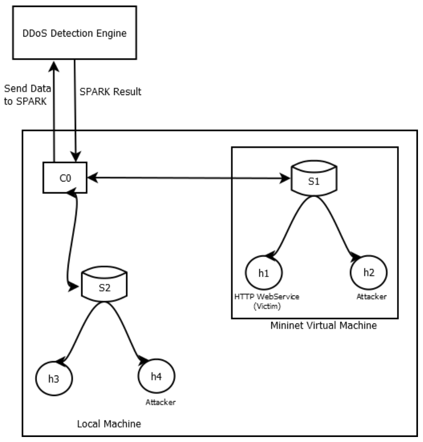


Fig. 2: Experimental Setup

This setup was implemented on a local (host) machine and a Mininet VM that was hosted on this local machine. The local machine contained a SDN network using Mininet with one switch, two hosts and one controller. The Mininet VM hosted another SDN that consisted of two hosts and one switch. This switch was connected to the remote controller present at the default port of 6633 on the local machine.

In this setup, host *h1* ran a python HTTP web server which acted as our victim. The host *h2* acted as the attacker which carried out TCP SYN flooding attack on *h1*. The switch *s1* communicated with the remote controller *c0* to get the flow-rules at the switch. The data received by the controller from the switch was then sent to the DDE running on the AWS EC2 cluster. The DDE is the heart of our project makes a decision from the input it receives from the controller making use of Apache Spark and the MLlib present within it.

Once the controller received this decision back from EC2, it relayed this information to the switch in the form of add/modify flow rules. The switch then took the necessary action(s) based on the instructions given by the controller.

The first time when the host *h2* requested a webpage from the web server *h1*, this request arrived at the switch *s1*. Since the switch *s1* did not have a flow rule corresponding to this flow in its flow tables, it sent this data to the (remote) controller *c0* seeking for an answer. The controller got this request data from the switch and relayed it to the DDE.

The main computation took place at the DDE. Using the pre computed model, the DDE predicted if the newly received packet from the controller was a malicious packet or a benign packet. Once this decision was taken, it notified the controller about it and sent this information to the controller. The controller then accepted this decision and instructed the switch by adding and/or modifying flow-rule(s) in its switch table. The switch now aware of what has to be done, either forwards/drops the packet as per the flow-rules set by the controller.

While designing our system, we selected Apache Spark which is a faster big data processing framework than Hadoop's MapReduce paradigm, and performs up to hundred times faster than Hadoop. Spark is an open source data analytics, cluster computing framework [19]. Spark supports in-memory clustering as opposed to involving disk operations within Map/Reduce processes [20]. For training a model and future predictions, we made use of the machine learning algorithms that are provided by Spark.

In addition to the already mentioned advantages, Spark has good support for AWS EC2 implementations. Spark comes with EC2 scripts that can be used to directly set up clusters along with certain command-line arguments [21]. It gives us the flexibility to even initiate a cluster on EC2 with a specific version of Spark as per what is input on the command-line.

To build the DDE, we trained it with datasets consisting of a mixture of normal and attack data. We trained the model using various classification algorithms and the classification accuracies are compared here. Research shows that these can be effectively for DDoS filtering [22]. We made use of the ML-lib, which is an in-built machine learning library framework provided by Apache Spark, to train the classification model.

In case of a malicious packet being encountered, the DDE sent a decision value of 1 (malicious). In cases of benign packets, it sent a decision value of 0 (benign). Depending on whether a 1 or 0 was received, the controller then instructed the switch to drop the packet (in case of malicious packets) or forward the packet (in case of benign packets).

The accuracy of the classification procedure depends on the set of features that are extracted from the network packets and fed to the DDE. If these features are selected carefully, we can get a very good classification output with minimal features. For the DDE, we selected a feature set consisting of 43 distinct features which are listed in Table I.

TABLE I: Feature Set For Training the Model

Feature	Description
frame.time_relative	Time since reference or first frame
ip.id	IP Identification
ip.proto	Protocol
frame.interface_id	Interface ID
frame.encap_type	Encapsulation Type
frame.offset_shift	Time shift for this packet
frame.time_epoch	Time epoch
frame.time_delta	Time delta from previous captured frame
frame.len	Time delta from previous captured frame
frame.cap_len	Frame length stored into the capture file
frame.marked	Frame is marked
frame.ignored	Frame is ignored
ip.hdr_len	Header Length
ip.dsfield	Differentiated Services Field
ip.dsfield.dscp	Differentiated Services Codepoint
ip.dsfield.ecn	Explicit Congestion Notification
ip.len	Total Length
ip.flags.rb	Reserved bit
ip.flags.df	Donť fragment
ip.flags.mf	More fragments
ip.frag_offset	Fragment offset
ip.ttl	Time to live
tcp.stream	Stream index
tcp.hdr_len	Header Length
tcp.flags.res	Reversed Flag
tcp.flags.ns	Nonce
tcp.flags.cwr	Congestion Window Reduced (CWR) Flag
tcp.flags.ecn	ECN-Echo
tcp.flags.urg	Urgent Flag
tcp.flags.ack	Acknowledgement Flag
tcp.flags.push	PUSH Flag
tcp.flags.reset	Reset Flag
tcp.flags.syn	SYN Flag
tcp.flags.fin	FIN Flag
tcp.window_size_value	Window size value
tcp.window_size	Calculated Window size
tcp.window_size_scalefactor	Window size scaling factor
tcp.option_len	Option Length
tcp.options.timestamp.tsval	Timestamp value
tcp.options.timestamp.tsecr	Timestamp echo reply
tcp.analysis.bytes_in_flight	Bytes in flight
eth.lg	LG bit
eth.ig	IG bit

V. Experiments and Results

We carried out a number of experiments on the experimental setup described in Figure 2. Due to network fair usage policies, we were not able to carry out an actual (massive) DDoS attack. Instead we have used the attack dataset provided by the Center for Applied Internet Data Analysis (CAIDA) that contains TCP SYN flooding attack data [23]. In our experiments we simulated SYN flooding using the hping3 tool available in linux.

Apache Spark and MLlib provide us with a number of machine learning algorithms viz. Decision Tree, Naive Bayes, Support Vector Machine (SVM) and . To come up with a good learning algorithm, we tested our datasets against some of the candidate machine learning algorithms provided by MLlib, the results of which are tabulated in Table II.

As can be seen from the Table II, SVM provides an error rate of more than 10% as compared to Decision Tree which exhibits an error of less than 3%. The time for training the model seems to be more than the time taken for prediction because the latter is for predicting about 36K records. The time for prediction per input record is thus obtained by dividing the given prediction times by 36K to yield time in the order of microseconds. The prediction time for Decision Tree for the given features set and dataset is 0.114 microseconds.

One more thing to note is that Decision Tree exhibits a very low error rate with the same feature set as compared to the others. On the other hand, Logistic Regression takes an even lesser time for prediction at the cost of a high error rate. There is a trade-off here and as per the requirements of our system, we would want a higher accuracy (lower error rate) as opposed to the time required for prediction.

The accuracy results are plotted in Figure 3. As can be seen from this figure and the argument put forth, the Decision Tree machine provides a high accuracy of more than 97% as compared to others which are lower than this value. Although these algorithms were run on datasets of size in the range of 36K, it works with almost the same accuracy with a Root Mean Square (RMS) error of less than 3%.

After taking accuracy into consideration, we tested the

Algorithm	Feature Set Size	Dataset Records	Time for Training (s)	Time for Prediction (s)	Error
Decision Tree	43	36,646	0.0002520084	0.0040941238	2.46%
SVM	43	36,646	0.0002040863	0.0044519901	10.75%
Naive Bayes	43	36,646	0.0005002022	0.0016739368	8.84%
Logistic Regression	43	36,646	0.0003650188	0.0011081696	9.04%

TABLE II: Performance of Various MLlib Machine Learning Algorithms

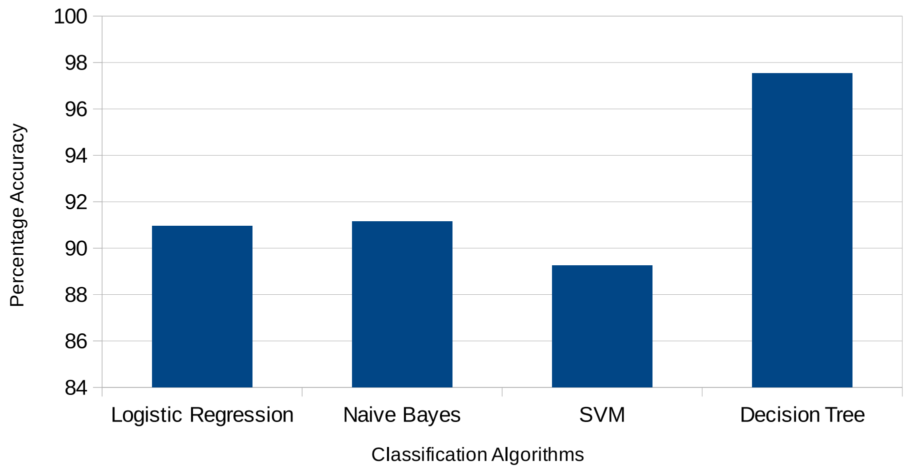


Fig. 3: Classification Accuracy of MLlib Machine Learning Algorithms

dataset for the time required for predictions, the results of which are shown in Figure 4 for a dataset of 36,646 records. We tested this dataset on an EC2 cluster (one master node and three slave nodes) and on the local machine that was supposed to host the SDN controller. As can be seen, the time taken by all the algorithms is significantly lesser when run on an EC2 cluster as compared to that on the local machine. Decision Tree, however, takes more time than a couple of its counterparts but this comes with an added advantage of improved accuracy. Since our system requires that the incoming packets are correctly specified given an attack is in force, accuracy become a major high-priority factor than the time taken for predictions. In particular, Decision Tree takes a 4.094124 ms to predict/classify a dataset of 36,646 records on the local machine as compared to 0.160933 ms on an EC2 cluster.

From the results, we selected Decision Tree as the machine learning algorithm for our system due to its high accuracy and very low error rates even on huge datasets. Its ability to work with categorical attributes added to its advantages over others.

We monitored and analysed the decisions that were taken by the DDE as against the actual interpretation of data. Out of all the experiments that we conducted, we achieved a high degree of accuracy with minimal error rate (more than 97% accuracy with an error rate of less than 3%) in accordance to the already discussed performance of the Decision Tree model.

We applied the following defense mechanism. After the packet was classified, we either forwarded the packet to its destination or blocked the malicious host from sending any further (attack) packets based on the decision from the DDE. OVS-OFCTL commands were used to achieve this. The following commands were sent from the controller to the switch based on the classification results:

- To enable forwarding in the switch:
  ovs-ofctl add-flow s1 priority=10,action=normal
- To block traffic from a malicious host:
  ovs-ofctl add-flow s1 priority=11,dl_type=0x0800,nw_src=10.0.0.1,action=drop
- To restore the traffic back:
  ovs-ofctl –strict del-flows s1 priority=11,dl_type=0x0800,nw_src=10.0.0.1

Adding a rule with a higher priority trumps the previous rule and the packets from the host with the specific IP are dropped. To restore the traffic, we simply deleted the latest flow rule (to drop the packets) added at the switch.

DDE Deployement on AWS EC2

Since the controller (and also the network) and the DDE have to be ideally separated physically and logically, we deployed our DDE code on an Amazon Web Services Elastic Cloud 2 (AWS EC2) cluster. This EC2 cluster contained only the code of the DDE which performed classification of incoming packets. The system architecture and setup remained the same, as mentioned in Figure 2.

In this case, the controller after receiving the incoming packets from the concerned switch, sent the captured network packet information to the DDE which was running on an EC2

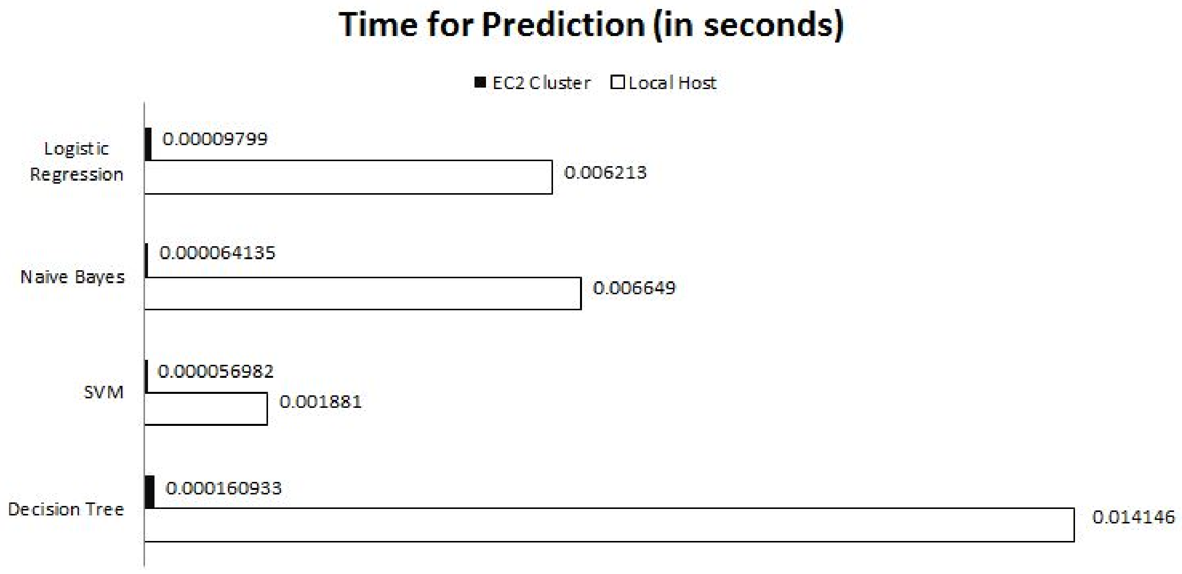


Fig. 4: Time Taken for Prediction by MLlib Machine Learning Algorithms

cluster. After the DDE performed its tasks, the controller then received the data from the DDE (EC2 cluster) and continued with its job.

The EC2 cluster, on the other hand, ran a daemon which continuously listened to connections from the local machine (controller). Once some bits came over the socket, it redirected it to our DDE code which accepted all the data and classified them against an already trained model. The results of this classification were then written to the socket to be sent to the requesting controller. Once this transfer was done, the job of the DDE was done and it went back to process the next incoming network packet information batch from the controller.

We created a cluster of 4 machines in EC2 - one master VM and three slave VMs. Each of these EC2 instances that we used hosted Ubuntu Amazon Machine Image (AMI), running 64-bit Ubuntu 12.04 with 1 GB of RAM.

As the EC2 cluster is an external system that needs to be accessed over an Internet network, the major issue with this experimental setup is the network latency that tags along with it. After the data was sent by the controller from the local machine, the time delay to receive the same data at the EC2 instance end was noticeably significant. There was nothing that could be done in this area as such a network latency is beyond our control.

Moreover, once the data received by the EC2 instance was processed, it needed to be sent back to the controller. This caused a major issue for us as the local machine was within the University network and was assigned an internal IP address. The IP address translation seemed difficult to counter within the time span allotted for this research. Due to these two major issues that we faced after deploying our DDE code on EC2, made us to install the DDE locally in the local machine itself.

DDE Deployment on Local Machine

Due to the issues with deployment on EC2, we deployed our DDE on a local machine which ran 64-bit Ubuntu 14.04 with 8 GB of RAM. However, the controller and DDE were still logically separated as the former was within a Mininet network and the latter was (physically) on the local machine. Instead of the local machine, the DDE could have been deployed anywhere, particularly on any other potential VM.

The entire procedure of DDoS attack detection and mitigation can be seen in Figure 5. As can be seen in the Figure 5, starting from the top left screen and moving in the clockwise direction, we have the classification performed by the DDE (running Apache Spark), the data being sent by the controller, the data being received by the controller, the mininet VM hosting the mininet network and the attack web requests being sent using hping3 [25].

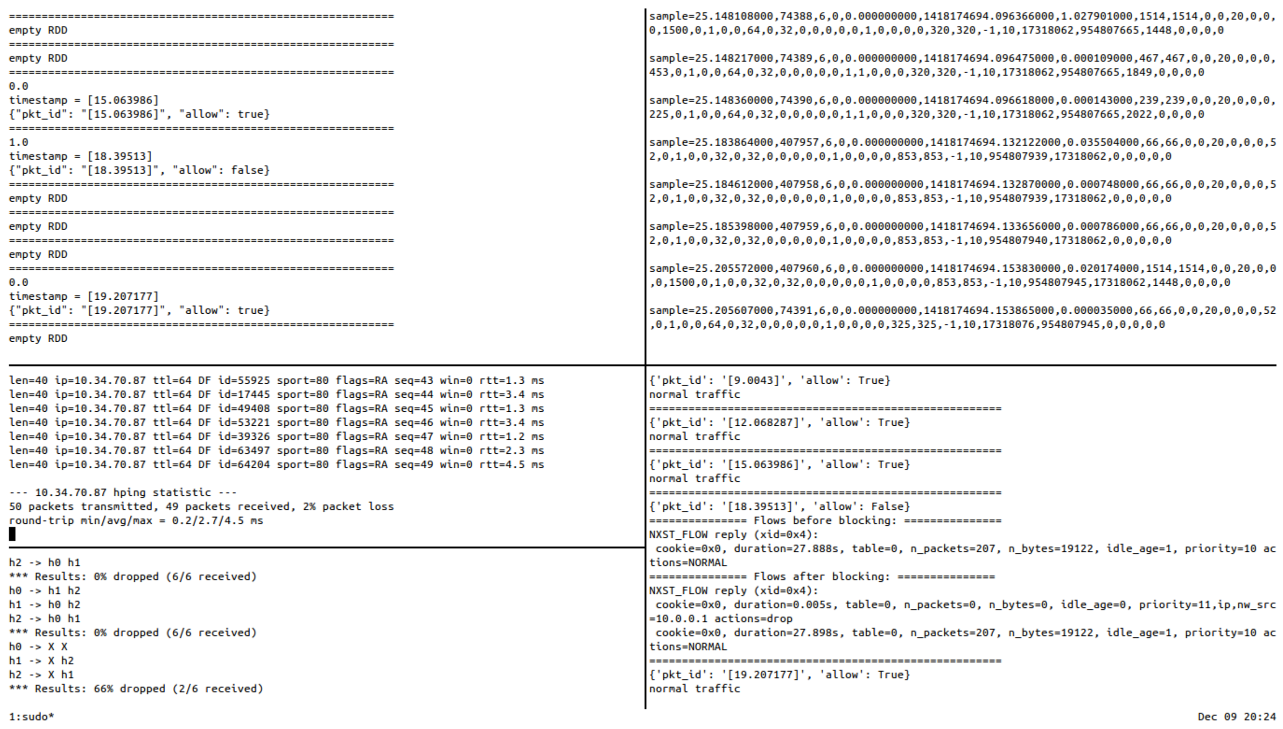
From the data that was received by the DDE, it can be seen that a timestamp is used to uniquely identify an incoming packet. This timestamp refers to the time of each incoming frame with reference to the first frame, and will always be unique for any host that is sending these packets and cannot be spoofed easily. The value displayed above the timestamp in Figure 5 is the decision of the DDE - 0.0 (benign packet) or 1.0 (malicious packet). This same decision is also mentioned in the form of a pre-defined structure as

“pkt_id”: “[*timestamp_value*]”, “allow”:true, or

“pkt_id”: “[*timestamp_value*]”, “allow”:true

The bottom-right corner shows the screen containing the controller terminal. It shows the data that was received from the DDE. The DDE decision (results) are in the same format as mentioned earlier

‘pkt_id’: ‘[*timestamp_value*]’, ‘allow’:true, or



```
============================================================
empty RDD
============================================================
empty RDD
============================================================
0.0
timestamp = [15.063986]
{"pkt_id": "[15.063986]", "allow": true}
============================================================
1.0
timestamp = [18.39513]
{"pkt_id": "[18.39513]", "allow": false}
============================================================
empty RDD
============================================================
empty RDD
============================================================
empty RDD
============================================================
0.0
timestamp = [19.207177]
{"pkt_id": "[19.207177]", "allow": true}
============================================================
empty RDD
```

```
sample=25.148108000,74388,6,0,0.000000000,1418174694.096366000,1.027901000,1514,1514,0,0,20,0,0,
0,1500,0,1,0,0,64,0,32,0,0,0,0,0,1,0,0,0,0,320,320,-1,10,17318062,954807665,1448,0,0,0,0

sample=25.148217000,74389,6,0,0.000000000,1418174694.096475000,0.000109000,467,467,0,0,20,0,0,0,
453,0,1,0,0,64,0,32,0,0,0,0,0,1,1,0,0,0,320,320,-1,10,17318062,954807665,1849,0,0,0,0

sample=25.148360000,74390,6,0,0.000000000,1418174694.096618000,0.000143000,239,239,0,0,20,0,0,0,
225,0,1,0,0,64,0,32,0,0,0,0,0,1,1,0,0,0,320,320,-1,10,17318062,954807665,2022,0,0,0,0

sample=25.183864000,407957,6,0,0.000000000,1418174694.132122000,0.035504000,66,66,0,0,20,0,0,0,5
2,0,1,0,0,32,0,32,0,0,0,0,0,1,0,0,0,0,853,853,-1,10,954807939,17318062,0,0,0,0,0

sample=25.184612000,407958,6,0,0.000000000,1418174694.132870000,0.000748000,66,66,0,0,20,0,0,0,5
2,0,1,0,0,32,0,32,0,0,0,0,0,1,0,0,0,0,853,853,-1,10,954807939,17318062,0,0,0,0,0

sample=25.185398000,407959,6,0,0.000000000,1418174694.133656000,0.000786000,66,66,0,0,20,0,0,0,5
2,0,1,0,0,32,0,32,0,0,0,0,0,1,0,0,0,0,853,853,-1,10,954807940,17318062,0,0,0,0,0

sample=25.205572000,407960,6,0,0.000000000,1418174694.153830000,0.020174000,1514,1514,0,0,20,0,0
,0,1500,0,1,0,0,32,0,32,0,0,0,0,0,1,0,0,0,0,853,853,-1,10,954807945,17318062,1448,0,0,0,0

sample=25.205607000,74391,6,0,0.000000000,1418174694.153865000,0.000035000,66,66,0,0,20,0,0,0,52
,0,1,0,0,64,0,32,0,0,0,0,0,1,0,0,0,0,325,325,-1,10,17318076,954807945,0,0,0,0,0
```

```
len=40 ip=10.34.70.87 ttl=64 DF id=55925 sport=80 flags=RA seq=43 win=0 rtt=1.3 ms
len=40 ip=10.34.70.87 ttl=64 DF id=17445 sport=80 flags=RA seq=44 win=0 rtt=3.4 ms
len=40 ip=10.34.70.87 ttl=64 DF id=49408 sport=80 flags=RA seq=45 win=0 rtt=1.3 ms
len=40 ip=10.34.70.87 ttl=64 DF id=53221 sport=80 flags=RA seq=46 win=0 rtt=3.4 ms
len=40 ip=10.34.70.87 ttl=64 DF id=39326 sport=80 flags=RA seq=47 win=0 rtt=1.2 ms
len=40 ip=10.34.70.87 ttl=64 DF id=63497 sport=80 flags=RA seq=48 win=0 rtt=2.3 ms
len=40 ip=10.34.70.87 ttl=64 DF id=64204 sport=80 flags=RA seq=49 win=0 rtt=4.5 ms

--- 10.34.70.87 hping statistic ---
50 packets transmitted, 49 packets received, 2% packet loss
round-trip min/avg/max = 0.2/2.7/4.5 ms
█
```

```
h2 -> h0 h1
*** Results: 0% dropped (6/6 received)
h0 -> h1 h2
h1 -> h0 h2
h2 -> h0 h1
*** Results: 0% dropped (6/6 received)
h0 -> X X
h1 -> X h2
h2 -> X h1
*** Results: 66% dropped (2/6 received)

1:sudo*
```

```
{'pkt_id': '[9.0043]', 'allow': True}
normal traffic
======================================================
{'pkt_id': '[12.068287]', 'allow': True}
normal traffic
======================================================
{'pkt_id': '[15.063986]', 'allow': True}
normal traffic
======================================================
{'pkt_id': '[18.39513]', 'allow': False}
=============== Flows before blocking: ===============
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=27.888s, table=0, n_packets=207, n_bytes=19122, idle_age=1, priority=10 ac
tions=NORMAL
=============== Flows after blocking: ===============
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=0.005s, table=0, n_packets=0, n_bytes=0, idle_age=0, priority=11,ip,nw_src
=10.0.0.1 actions=drop
 cookie=0x0, duration=27.898s, table=0, n_packets=207, n_bytes=19122, idle_age=1, priority=10 ac
tions=NORMAL
======================================================
{'pkt_id': '[19.207177]', 'allow': True}
normal traffic

                                                                                  Dec 09 20:24
```

Fig. 5: DDoS Attack and Detection

'pkt_id': '[*timestamp_value*]', 'allow':true

However, it can be seen that once the controller receives a decision of a packet being malicious (indicated by the *'allow': False*), the controller sends a flow request to the switch asking it to block this flow that contains the concerned malicious packet. Before this packet was identified, the flow table contained only one rule (specified under *"Flows before blocking"*)) with an *action=NORMAL* parameter which the switch could use to forward data traffic. But once the "malicious" decision was received by the controller from the DDE, it sent out a *flow-add* request to block this flow. This is evident from the dump of the flow table after application of the rule (specified under *"Flows after blocking"*) which contains a flow-rule with *action=drop* as its first rule in the flow table. This rule will then be executed by the switch to block the concerned flow.

This can also be seen in the bottom-left corner which shows the mininet VM screen. Before the attack occurred, each of the hosts in the SDN viz. *h0*, *h1* and *h2* were able to ping each other. However, once the attack occurred and the flow was blocked, the attacker (host *h0*) was not able to ping the other hosts *h1* and *h2*. As a consequence, the host *h0* was not even reacheable from the other hosts *h1* and *h2*.

The middle frame in the left pane of Figure 5 shows the attack packets being generated with the minimum, average and maximum round-trip time for a batch of 50 attack packets. Since these packets were termed to be malicious, they were dropped. The "ping" statistics show this fact indicating a 2% packet loss.

The top-right corner screen displays the data that is sent from the controller to the DDE. It contains records with all the elements of the feature set that the controller extracted from an incoming network packet (sent from the switch as a request).

Once this deployment was done, the performance of the entire system as a whole improved significantly, and we were able to achieve near real-time performance.

Although the results were obtained in a reasonable amount of time, the actual performance did not match the expected performance. This was due to the fact of many underlying limitations and external factors . During our research, there were many obstacles due to which an efficient implementation of the system was not feasible viz. the University network fair usage policies and the use of virtual machines. Although EC2 provides us with a platform to take advantage of cluster infrastructure, due to underlying network bottleneck, the data transmission does not happen at high speeds leading to some latency in the data transmission. Moreover, the use of a smaller magnitude of processing power at the EC2 cluster also degraded the performance to some extent. These limitations of the underlying infrastructure and University policies made our results deviate from the expected high real-time performance. However, we did achieve a near real-time response and performance. These results will be further enhanced and the latency further reduced if this system implementation could be carried out in an actual hardware/software setup.

VI. Future Work

As part of future research, there are a few areas where improvements can be made. The major assumption regarding the security of the SDN controller can be re-assessed. The controller can, just as the other network components, be a target of an attack. Since this is where the intelligence of the SDN is concentrated, it is a very lucrative part to be an attack target. The robustness and security of the system can be further improved by making the controller resistant to any type of DoS and DDoS attacks.

In addition to this, although the system that we have proposed is able to detect different types of DDoS and DoS attacks, we have performed only TCP SYN flooding attacks on the SDN. This system could be further tested to detect and counter DDoS attacks more types of DDoS/DoS attacks viz. ICMP flooding etc. This can be used to create a comprehensive dataset that could be used to analyse and/or evaluate such a system in greater depths.

Moreover, we have used a small cluster containing only four nodes on EC2 which negatively impacts the performance of our system due to network latency. This can be, however, improvised by increasing the size of the EC2 cluster or building an actual cluster in hardware. Though the latter option seems difficult, it would certainly reduce the latency thereby bringing the performance in par with real-time response systems.

VII. Conclusion

To address the ever-growing network security issue of DDoS, we have proposed a system to detect and counter it in Software Defined Networks (SDN) in real-time. The advantages of centralized control and decoupling of the control and the data plane in SDN makes the real-time aspect of the computation possible.

We set up a SDN network communicating with a remote controller, and also a DDoS Detection Engine (DDE) on a cluster of commodity machines on EC2. These machines were made to run Apache Spark which supports data streaming, and is hundred times faster than its counterpart Hadoop. Apache Spark along with the in-built Machine Learning library, we trained our model from historical DDoS attack data and normal traffic data. After this implementation, the DDE was able to detect and counter DDoS attacks in near real-time scenarios. Experiments show that this solution can scale well with a large size cluster running the computation.

VIII. Acknowledgement

We are highly indebted to Dr. Andy Li, the Department of Computer Science & Engineering, and the Department of Electrical & Computer Engineering at the University of Florida for providing us with the opportunity, guidance and resources without which this research project would not be possible. We would like to extend our gratitude to the Center for Applied Internet Data Analysis (CAIDA) for providing us with the DDoS attack dataset. We are also grateful to all our friends and colleagues who have assisted and motivated us throughout the duration of our research, and provided constructive feedback on our work.

References

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

[2] C. Rossow, "Amplification Hell: Revisiting Network Protocols for DDoS Abuse." *Network and Distributed System Security Symposium, NDSS 2014*, 2014.

[3] "Software-Defined Networking: The New Norm for Networks," https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[4] P. Morreale and J. Anderson, *Software Defined Networking: Design and Deployment*. Taylor & Francis, 2014. [Online]. Available: http://books.google.com/books?id=4RUeBQAAQBAJ

[5] "How to defend against DDoS attacks," http://www.computerworld.com/article/2564424/security0/how-to-defend-against-ddos-attacks.html, accessed: 2014-09-15.

[6] "Real Time SDN and NFV Analytics for DDos Mitigation," http://techfieldday.com/video/real-time-sdn-and-nfv-analytics-for-ddos-mitigation, accessed: 2014-09-15.

[7] "DefenseFlow: The SDN Application that Programs Networks for DoS Security," http://www.radware.com/Products/DefenseFlow/, accessed: 2014-09-15.

[8] "InMon: sFlow-RT," http://www.inmon.com/products/sFlow-RT.php, accessed: 2014-09-15.

[9] Y. Lee and Y. Lee, "Detecting DDoS Attacks with Hadoop," in *Proceedings of The ACM CoNEXT Student Workshop*, ser. CoNEXT '11 Student. New York, NY, USA: ACM, 2011, pp. 7:1–7:2. [Online]. Available: http://doi.acm.org/10.1145/2079327.2079334

[10] D. Gavrilis and E. Dermatas, "Real-time detection of distributed denial-of-service attacks using RBF networks and statistical features," *Computer Networks*, vol. 48, no. 2, pp. 235–245, 2005.

[11] S. M. Mousavi, "Early Detection of DDoS Attacks in Software Defined Networks Controller," Ph.D. dissertation, Carleton University, 2014.

[12] S. Ganapathy, K. Kulothungan, S. Muthurajkumar, M. Vijayalakshmi, P. Yogesh, and A. Kannan, "Intelligent feature selection and classification techniques for intrusion detection in networks: a survey," *EURASIP Journal on Wireless Communications and Networking*, vol. 2013, no. 1, 2013. [Online]. Available: http://dx.doi.org/10.1186/1687-1499-2013-271

[13] J. Seo, J. Kim, J. Moon, B. J. Kang, and E. G. Im, "Clustering-based Feature Selection for Internet Attack Defense," *International Journal of Future Generation Communication and Networking*, vol. 1, no. 1, pp. 91–98, 2008.

[14] "Aws - amazon elastic compute cloud(ec2) - scalable cloud hosting," http://aws.amazon.com/ec2/, accessed: 2014-11-25.

[15] "Apache Spark - Lightening Fast Cluster Computing," https://spark.apache.org/, accessed: 2014-09-15.

[16] "MLlib Apache Spark," https://spark.apache.org/mllib/, accessed: 2014-10-13.

[17] "Spark streaming," https://spark.apache.org/streaming/, accessed: 2014-11-25.

[18] "Mininet - An Instant Virtual Network on your Laptop (or other PC)," http://www.mininet.org, accessed: 2014-09-08.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[20] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[21] "Running Spark on EC2," http://people.apache.org/ andrewor14/spark-1.1.1-rc1-docs/ec2-scripts.html, accessed: 2014-11-08.

[22] "Applying Machine Learning for DDoS Filtering," http://stats.stackexchange.com/questions/23488/applying-machine-learning-for-ddos-filtering, accessed: 2014-10-23.

[23] "Center for Applied Internet Data Analysis," http://www.caida.org/data/, accessed: 2014-11-11.

[24] “HULK, Web Server DoS Tool,” http://www.sectorix.com/2012/05/17/hulk-web-server-dos-tool/, accessed: 2014-09-03.
[25] “hping3,” http://hping.org, accessed: 2014-11-25.