**Machine Learning Approach for an Anomaly Intrusion Detection System using ONOS**

*University of Texas at Dallas*

*CS 6301-503: Software Defined Networks (Spring 2018)*

*Professor Timothy Culver*

**Project Team**

Stephen Blystone

Taniya Riar

Juhi Bhandari

Ishwank Singh

Table of Contents

[Introduction 3](#_Toc512462795)

[Problem Statement 3](#_Toc512462796)

[Architecture 3](#_Toc512462797)

[Tools Used 4](#_Toc512462798)

[Implementation 6](#_Toc512462799)

[Traffic and Network Generation 6](#_Toc512462800)

[Big Data Infrastructure 8](#_Toc512462801)

[Big Data Pipeline Architecture 9](#_Toc512462802)

[Elasticsearch Index Creation 10](#_Toc512462803)

[Machine Learning 11](#_Toc512462804)

[Feature Engineering 11](#_Toc512462805)

[Capturing Data for Training 12](#_Toc512462806)

[Supervised vs. Unsupervised Machine Learning 12](#_Toc512462807)

[K-Means Machine Learning Algorithm 12](#_Toc512462808)

[Standardizing the Data 13](#_Toc512462809)

[Threshold Implementation 13](#_Toc512462810)

[Generating Anomalies 15](#_Toc512462811)

[ONOS REST API Call 16](#_Toc512462812)

[Side Effect of Anomaly Detection 18](#_Toc512462813)

[Data Visualization 19](#_Toc512462814)

[Future Scope 19](#_Toc512462815)

[Lessons Learned 19](#_Toc512462816)

[Conclusion 20](#_Toc512462817)

[References 21](#_Toc512462818)

[Appendix A – Guide Descriptions 23](#_Toc512462819)

[Appendix B – Code File Descriptions 25](#_Toc512462820)

# Introduction

Our project implements an Anomalous Intrusion Detection System Architecture using Big Data, Machine Learning, and the ONOS controller. It identifies and blocks anomalous traffic in real-time. Our project is divided into the following parts:

* Creation of a miniature network (Mininet) to simulate traffic which is controlled by the ONOS Controller.
* NetFlow messages sent from Open vSwitches in Mininet to the Big Data Infrastructure.
* The Big Data Infrastructure performs:
  + Feature Engineering on the NetFlow data and stores features in Elasticsearch.
  + Machine Learning (K-Means) to detect 3 clusters using 36 hours of “Normal” traffic data.
  + Anomaly detection of new data compared with our trained Machine Learning model.
  + Visualization of NetFlow data in Kibana.
* REST API call sent from Big Data Infrastructure to ONOS when an anomaly is detected.
* ONOS updates the flow rules in the Open vSwitch in Mininet to block the source of the anomaly.

# Problem Statement

There are two categories of Intrusion Detection Systems (IDS): signature-based IDS and anomaly-based IDS. Signature-based IDS look for known patterns (or signatures) in network traffic to identify intrusions. One popular open-source product for signature-based IDS is Snort. Anomaly-based IDS look for anomalous network traffic. This is used to identify new or previously unknown intrusion methods.

The goal of our project is to develop an architecture that incorporates SDN, Big Data, and Machine Learning to detect anomalies in network traffic and send a REST API call to the ONOS controller to block the source of the anomalous traffic in real-time.

Our data consists of NetFlow messages generated from Open vSwitches located in Mininet. Due to the volume of NetFlow messages being sent we require a Big Data Infrastructure to buffer, process, and store the data. We incorporate our Machine Learning algorithm into the Big Data Infrastructure.

# Architecture

Our project architecture is comprised of three sections (each in a separate Virtual Machine):

1. ONOS Controller – SDN Controller for the Mininet network. It communicates with Mininet via OpenFlow. It also receives REST API calls from the Big Data Infrastructure.
2. Miniature Network (Mininet) – Contains a realistic network architecture comprised of 9 “users with laptops”, 3 “servers”, and 10 Open vSwitches. 5 of the Open vSwitches are configured with NetFlow and forward those messages to the Big Data Infrastructure.
3. Big Data Infrastructure – Receives and processes NetFlow messages from Mininet. Performs Feature Engineering and stores data in Elasticsearch. Performs Machine Learning (K-Means) in PySpark to identify normal traffic patterns in stored data. Determine if new NetFlow messages are anomalous compared to normal network traffic patterns. If an anomaly is detected, it sends the source IP address of the anomaly to ONOS via REST API. It also uses Logstash and Kibana to visualize NetFlow data.

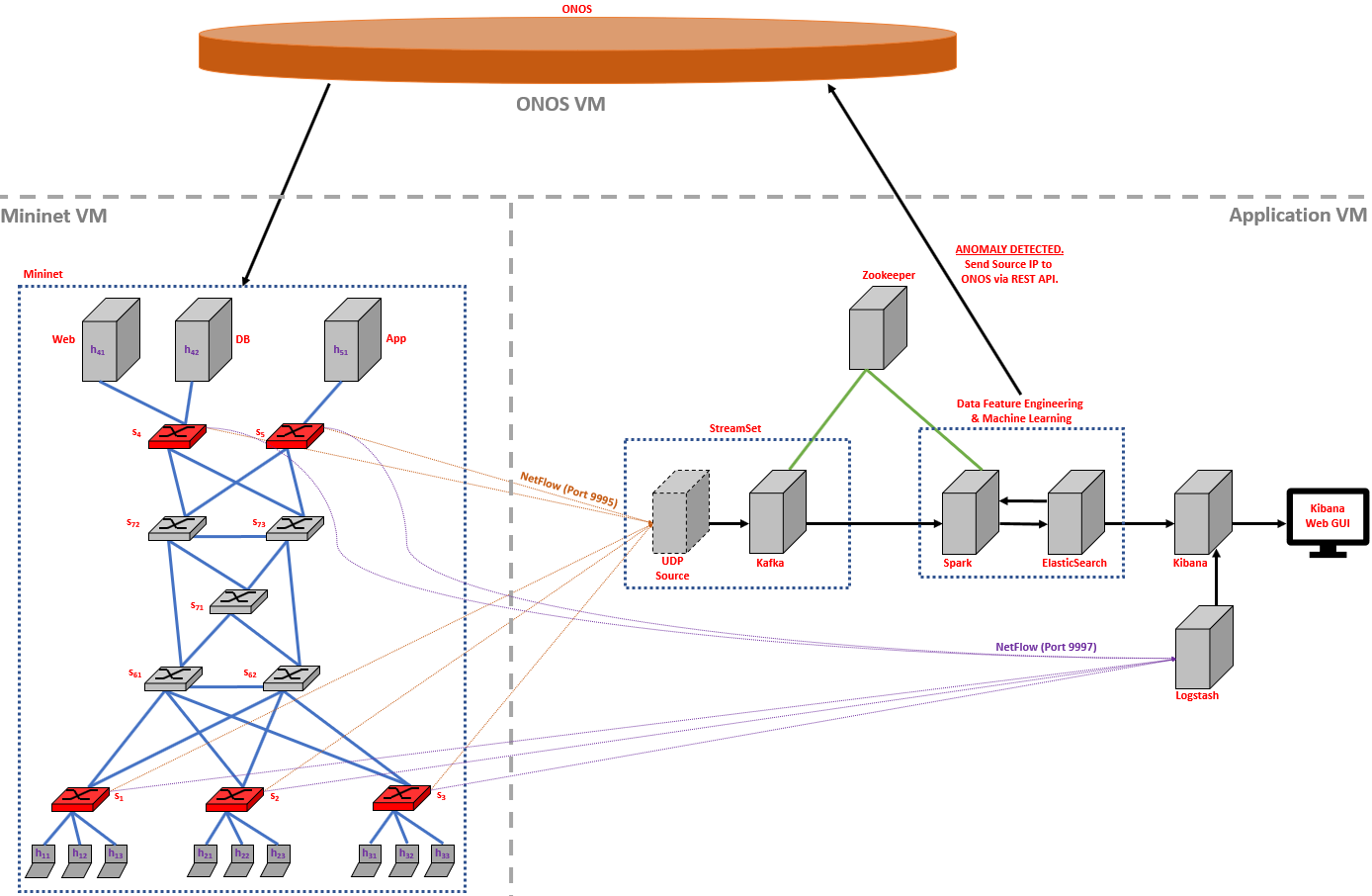


Figure 1 - The Project Architecture

# Tools Used

* Oracle VM VirtualBox – VirtualBox is used to host our 3 Virtual Machines.
* ONOS Controller – ONOS stands for Open Network Operating System and is a project of the Open Networking Foundation. ONOS provides the control plane for our software-defined network (SDN), communicating with the Open vSwitches in our network via OpenFlow.
* Mininet – Mininet is a software emulator for prototyping a large network on a single machine. It can be used to quickly create a realistic virtual network running actual kernel, switch and software application code on a personal computer. It allows the user to quickly create, interact with, customize and share a software-defined network (SDN) prototype to simulate a network topology that uses OpenFlow switches.
* NetFlow – NetFlow is a network protocol developed by Cisco for collecting IP traffic statistics and monitoring network traffic. Routers and switches that support NetFlow can collect IP traffic statistics on all interfaces where NetFlow is enabled, and later export those statistics as NetFlow records toward at least one NetFlow collector - typically a server that does the actual traffic analysis. We used NetFlow version 5 in our project. Version 5 is the only version currently available in Mininet. The newest release is NetFlow version 10, aka IPFIX. Version 5 statistics include source and destination IP, source and destination port, total number of bytes in the packets of the flow, etc.
* StreamSet – StreamSet is a cloud native collection of products to control data drift; the problem of changes in data, data sources, data infrastructure and data processing. The company calls its applications a data operations platform. We used StreamSet to create a UDP NetFlow collector that translated NetFlow messages into a form that Apache Kafka recognized.
* Apache Zookeeper – Apache Zookeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services between some Apache products. We use Apache Zookeeper to manage Apache Kafka and Apache Spark.
* Apache Kafka – Apache Kafka is a distributed streaming platform. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies. Kafka is generally used for building real-time streaming data pipelines that reliably get data between systems or applications and for building real-time streaming applications that transform or react to the streams of data. We used it to receive the data from the StreamSet UDP NetFlow collector and buffer the data until Apache Spark is ready to use it.
* Apache Spark – Apache Spark is an open-source cluster-computing framework. Apache Spark is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing. It is a fast, in-memory data processing engine with development APIs to allow data workers to execute streaming, machine learning or SQL. We used Apache Spark to perform feature engineering and Machine Learning. After feature engineering, Apache Spark sends the data to Elasticsearch for storage.
* PySpark – PySpark is Apache Spark’s Python library.
* Elasticsearch – Elasticsearch is a search engine based on Lucene. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is developed in Java and is released as open source under the terms of the Apache License. Elasticsearch is developed alongside a data-collection and log-parsing engine called Logstash, and an analytics and visualization platform called Kibana. The three products are designed for use as an integrated solution, referred to as the "Elastic Stack" (formerly the "ELK stack").
* Kibana – Kibana is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Below is an example of the visualization we can see for the NetFlow data:

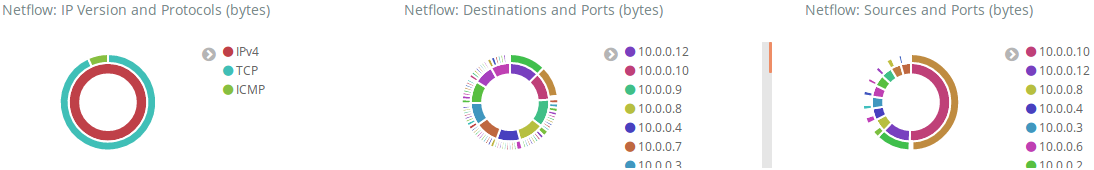


Figure 2 - Kibana NetFlow Data Visualization

* Logstash – The combination of Elasticsearch, Logstash, and Kibana, referred to as the "Elastic Stack" (formerly the "ELK stack"), is available as a product or service. Logstash provides an input stream to Elasticsearch for storage and search, and Kibana accesses the data for visualizations such as dashboards.

# Implementation

## Traffic and Network Generation

The Mininet topology used is like a real-world topology. We used Python scripts to configure the network topology, configure the Open vSwitches with NetFlow, and generate traffic. The traffic generated includes randomness to emulate “Normal” network traffic.

There are 9 hosts that are “users with laptops” labeled h11 – h33. These users can send HTTP requests to the Web Server, send ICMP Pings to any host or server, can send data via TCP to the App Server, and can send data via TCP to another user laptop.

The Web Server contains a webpage (index.html) and is labeled h41. It can respond to HTTP requests from hosts and can send data via TCP to the DB Server.

The DB Server is labeled h42. It responds to messages sent from the Web Server via TCP. The Web Server is the only host that communicates with the DB Server.

The App Server is labeled h51. It responds to messages sent from the users via TCP.

There are 5 Access Open vSwitches (labeled in red in the diagram below) that have NetFlow enabled. There are 5 Core Open vSwitches that do not have NetFlow.

Client.py and Server.py are run on every host (both “users with laptops” and “Servers”). Client.py generates the random “Normal” traffic. Server.py is a server program that listens for TCP messages from other hosts. Each host has a different port it listens on for network traffic. The Web Server port is 8000, the DB Server port is 2000, and the App Server port is 3000. For the Web Server to respond to HTTP requests, it must have a webpage for clients to request and download. If the file does not already exist, I had the Web Server generate a fake index.html page and store it in the root web directory containing fake “Lorem Ipsum” Latin text.

Each “user with laptop” must have a unique listening port since all hosts reside on the same Virtual Machine. The IP addresses of the “users with laptops” are in the range 10.0.0.1 to 10.0.0.9. Similarly, the listening ports are in the range 5001 to 5009, where the last octet of the IP address corresponds to the last digit in the port.

To make the generated network traffic appear “normal” I had to insert multiple sources of randomness into Client.py. The “users with laptops” sleep for a random duration between 1-10 seconds. They then choose one of four tasks at random:

1. Send data (random amount between 1 and 1024 bytes) to another host (“user with laptop”).
   1. Choose which host to send to at random (other than self).
2. Ping another IP.
   1. Choose host or server to send to at random (other than self).
3. Send HTTP request to Web Server.
4. Send data (random amount between 1 and 1024 bytes) to App Server.

The Web Server generated random traffic by sleeping for a random duration between 1-10 seconds. It then sent a random amount of data (between 1 and 1024 bytes) to the DB Server.

The DB and App Servers did not generate traffic. They only responded to each message.

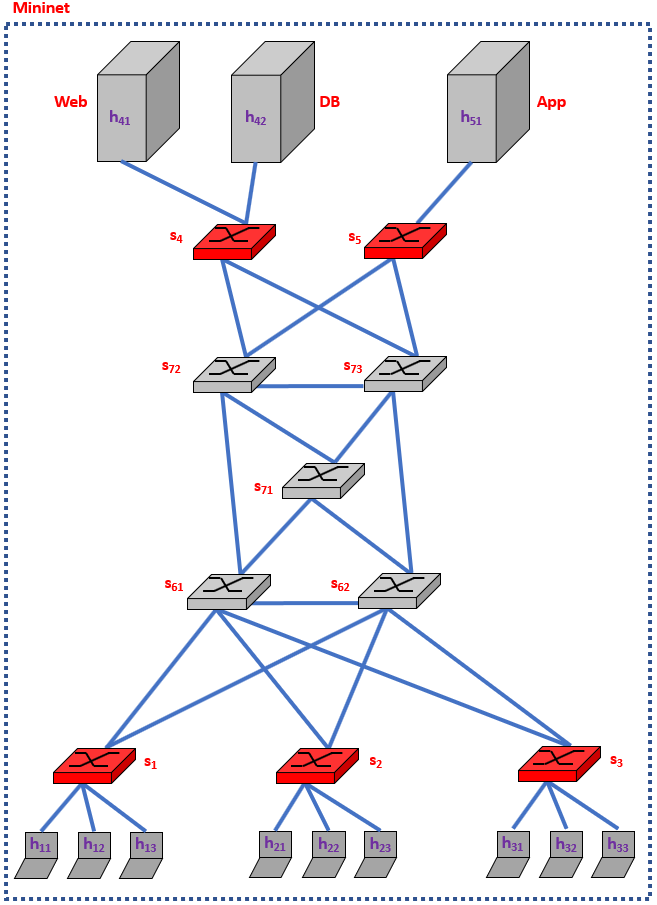


Figure 3 - Mininet Architecture

## Big Data Infrastructure

Due to the volume of NetFlow messages coming from the Open vSwitches in our Mininet network, we had to implement a Big Data Infrastructure. The NetFlow messages are received by the UDP Source NetFlow Collector in StreamSet, converted to JSON, and passed to Apache Kafka for buffering. The messages then flow to Apache Spark for feature engineering and Machine Learning. Apache Spark stores the features in Elasticsearch in index form for fast retrieval. Elasticsearch passes the data to Kibana for visualization. The raw NetFlow data is also sent to Logstash and visualized in Kibana.

### Big Data Pipeline Architecture

Below is a basic Big Data architecture for real-time processing from the Microsoft website (<https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/real-time-processing>). We are taking inspiration from there when designing our architecture.

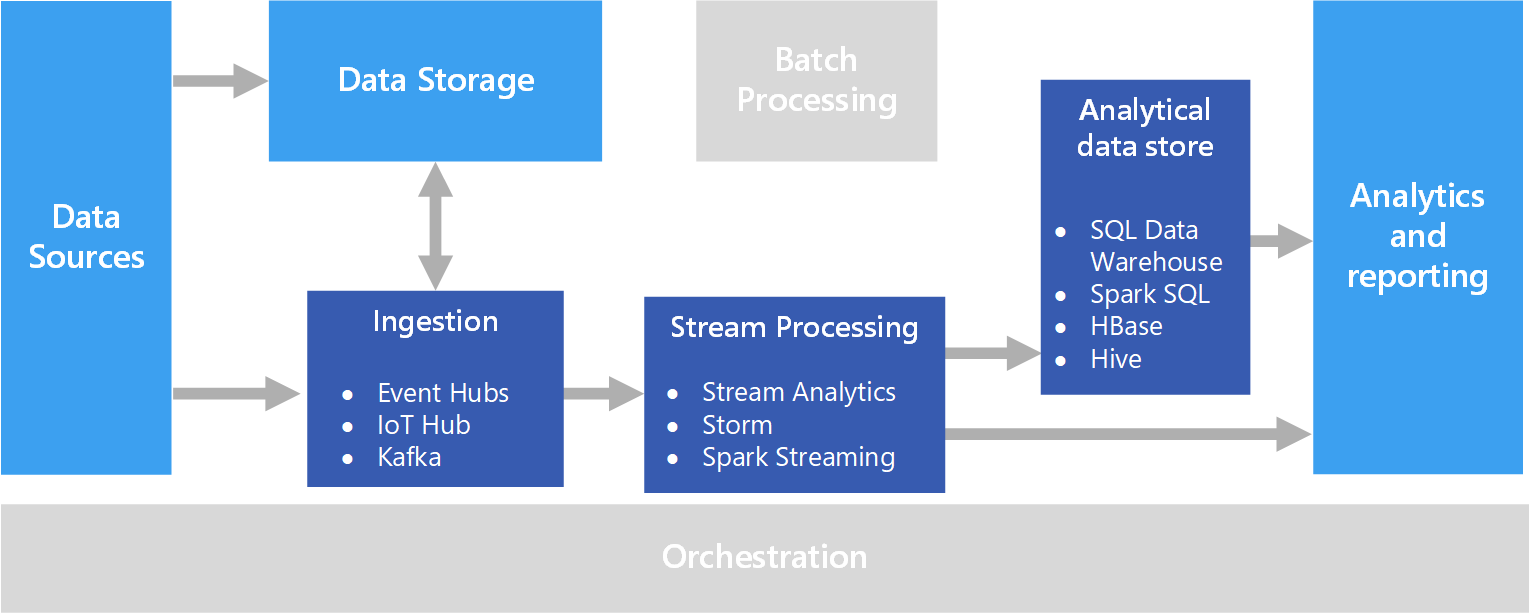


Figure 4 - Big Data Pipeline Architecture from Microsoft

1. Data Sources
   * The Open vSwitch nodes sending NetFlow data.
2. Data Storage
   * We are not using this component in our design.
3. Ingestion
   * This component essentially performs Stream Buffering, providing the “Stream Processing” block enough time to process the stream data without being overrun with data.
   * We use Apache Kafka.
4. Stream Processing
   * This component filters, aggregates, and prepares the data for analysis.
   * We use Apache Spark Streaming.
5. Analytical Data Store
   * This is where the data is stored after being processed.
   * We use Elasticsearch.
   * The Machine learning part is done using PySpark.
6. Analytics and Reporting
   * We use Logstash and Kibana for this.
7. Distributed Synchronization (Not Listed in Microsoft diagram)
   * Maintain configuration information, naming, providing distributed synchronization, and providing group services. This is mainly used for distributed systems.
   * We use Apache Zookeeper to manage synchronization between Apache Kafka and Apache Spark.

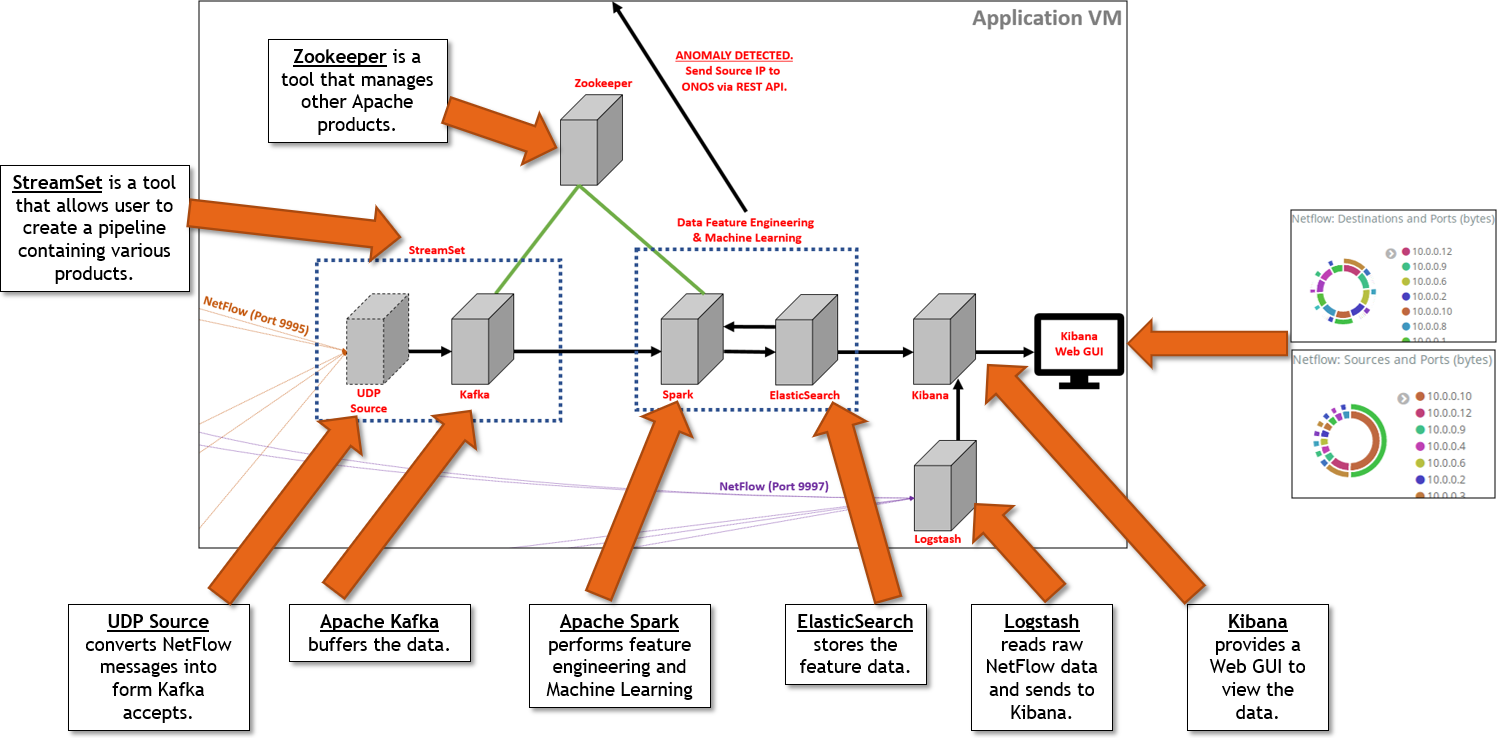


Figure 5 – Big Data Infrastructure High-Level Explanations

### Elasticsearch Index Creation

Elasticsearch is a distributed document store. It can store and retrieve complex data structures—serialized as JSON documents—in real time. In other words, as soon as a document has been stored in Elasticsearch, it can be retrieved from any node in the cluster.

The following is the Index created for storage structure of the data in the Elasticsearch.

Note that we created the index and captured data for the feature “sumOfFlows” but after testing we did not incorporate it into our Machine Learning training.

# the index created in Elasticsearch using Kibana Dev Tool

PUT /netflowrepo/entry/\_mapping

{

"entry" : {

"properties" : {

"sumOfFlows" : { "type" : "long" },

"sumOfBytes" : { "type" : "long" },

"uniqDstIPs" : { "type" : "integer" },

"uniqDstPorts" : { "type" : "integer" }

}

}

}

## Machine Learning

### Feature Engineering

Feature Engineering is defined as the process of using domain knowledge of the data to create features that make the Machine Learning algorithm work. Apache Spark captures data in intervals lasting a fixed number of seconds. We chose that number to be 60 seconds. An example of how this works is:

* From 0 to 60 seconds Apache Spark collects the NetFlow data received during that interval.
* After that interval is finished:
  + It begins the next interval (60 to 120 seconds)
  + And it passes the data collected to our PySpark application for feature engineering and processing.

We researched all the data that NetFlow version 5 provided and selected the following pieces of data:

* srcaddr\_s: This is the source IPv4 address of the flow.
  + We mapped this value to “srcAddr”
* dstaddr\_s: This is the destination IPv4 address of the flow.
  + We mapped this value to “dstAddr”
* dstport: This is the destination port of the flow.
  + We mapped this value to “dstPort”
* dOctets: This is the number of bytes sent in the flow.
  + We mapped this value to “numBytes”
* count: This is the number of flows aggregated together.
  + We mapped this value to “numFlows”

From this raw NetFlow data, we performed feature engineering to determine the following information for each srcAddr:

* sumOfFlows: Sum of all flows (numFlows) for a given srcAddr.
  + NOTE: We stored this information in Elasticsearch, but we did NOT end up using it as a feature in our Machine Learning training.
  + This feature was causing too many False Positives (detecting normal traffic as Anomalies). We removed this feature before we began standardizing the data, so it is possible standardizing would have fixed this issue.
* sumOfBytes: Sum of all bytes (numBytes) sent by a given srcAddr.
* uniqDstIPs: Count of the number of unique destination IP addresses a given srcAddr sent to.
* uniqDstPorts: Count of the number of unique destination Ports a given srcAddr sent to.

The above 4 features were stored in Elasticsearch.

### Capturing Data for Training

We captured and stored 36 hours’ worth of “Normal” network traffic data in Elasticsearch.

### Supervised vs. Unsupervised Machine Learning

Supervised Machine Learning is used when you have labeled data. You provide the Machine Learning algorithm examples of “Normal” traffic and examples of “Attack” traffic, and it learns how to recognize them. After being trained, when shown a previously unseen example the algorithm should be able to correctly classify if the new traffic is “Normal” or if it is an “Attack”. This would work for Signature-based IDS because it can recognize the patterns of specific attacks.

Unsupervised Machine Learning is used when you do not have labeled data. We chose Unsupervised Machine Learning since we only had examples of “Normal” network traffic data to train from. Unsupervised Machine Learning is typically used when looking for anomalies.

### K-Means Machine Learning Algorithm

We used the data stored in Elasticsearch to train our Machine Learning clustering algorithm. Clustering-based algorithms group similar data into clusters. K-Means algorithm splits the training data into ‘K’ clusters where each observation belongs to the cluster with the closest center. We used an elbow graph to determine the optimal number of clusters to be K = 3. We implemented the K-Means algorithm using PySpark.

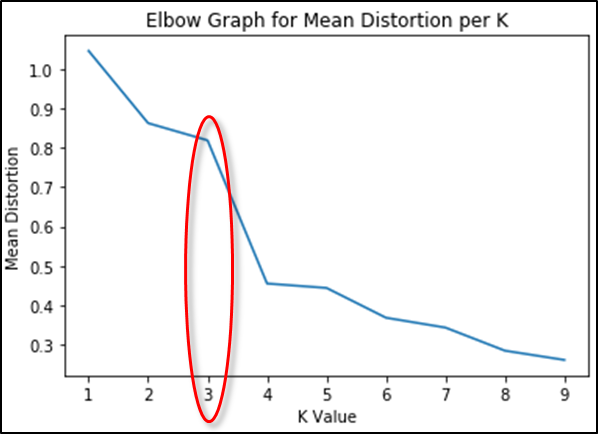


Figure 6 - Elbow Graph for 'K' determination for K-Means

### Standardizing the Data

We standardized each feature by subtracting the mean and dividing by the standard deviation. We did this when training the K-Means algorithm and when testing new data to determine if it was “normal” or an “anomaly”.

Before standardizing, different features can have different scales. For example, sum of bytes could have a max value of 10,000 and a min value of 50. Number of unique destination ports could have a max of 100 and a min of 3. The feature with the larger values is treated as “more important” by the algorithm. Standardizing the data places all the features on a similar scale so the algorithm does not treat one feature as more important than the other.

### Threshold Implementation

Each piece of data in Elasticsearch that is used to train the K-Means algorithm forms a single data point in 3-dimensions. We are using 3-dimensions due to having 3 features we are training with. The number 3 is unrelated to the K=3 clusters that we assigned our algorithm. The data point can be thought of as in geometry where (3, 5, 7) is the point where x=3, y=5, and z=7. In our case, we have “sumOfBytes”, “uniqDstIPs”, and “uniqDstPorts” instead of x, y, and z.

The K-Means algorithm assigned each of the “Normal” data points stored in Elasticsearch to 1 of the K=3 clusters. When trying to assign a new data point to one of the 3 clusters, we will assign it to the closest cluster. However, we need to have a threshold in place so that if the new data point is farther than the threshold for the assigned cluster, then the new data is considered an anomaly. We chose the threshold for each cluster to be the (maximum value assigned to that cluster) + 2 \* (standard deviation). We chose 2 as a starting point. This could be further refined with additional tests. Any new data points that are within the threshold for their assigned cluster would be considered “Normal” traffic.

An important concept with network traffic is that the traffic is still “normal” if there is less than the usual amount. For example, let’s say a user normally sends 3500 bytes to 80 ports. They go on vacation and for one interval only send 1000 bytes to 10 ports. This value is outside any known cluster, so should it be considered an anomaly? No, it is still “normal” even though it is less than their usual amount.

The below diagrams show example data that indicate this thresholding concept. Anything in a square or circle is still considered “Normal”. Only point E in Figure 8 is an “Anomaly”.

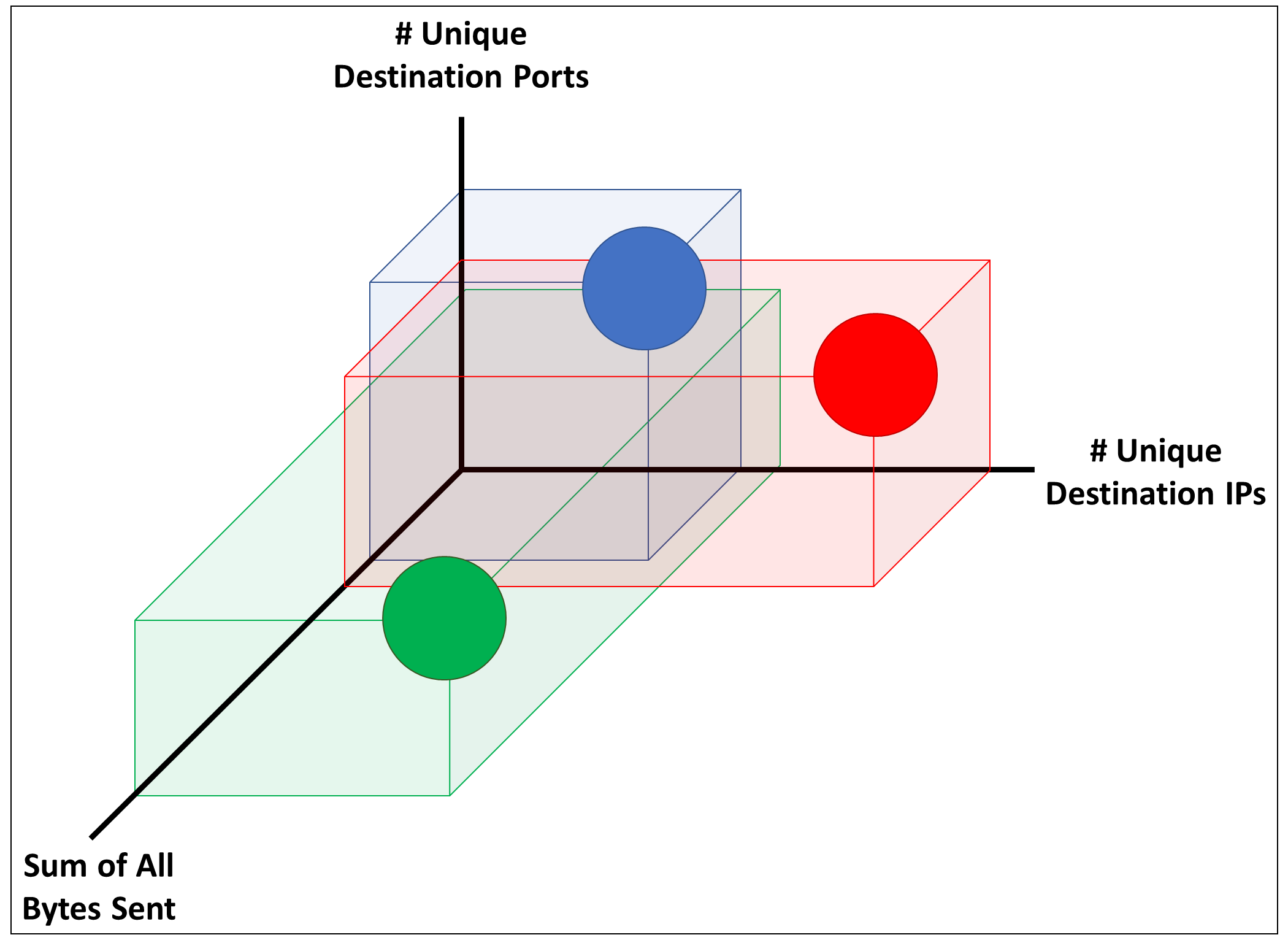


Figure 7 - Example Diagram showing Thresholding in 3-dimensions

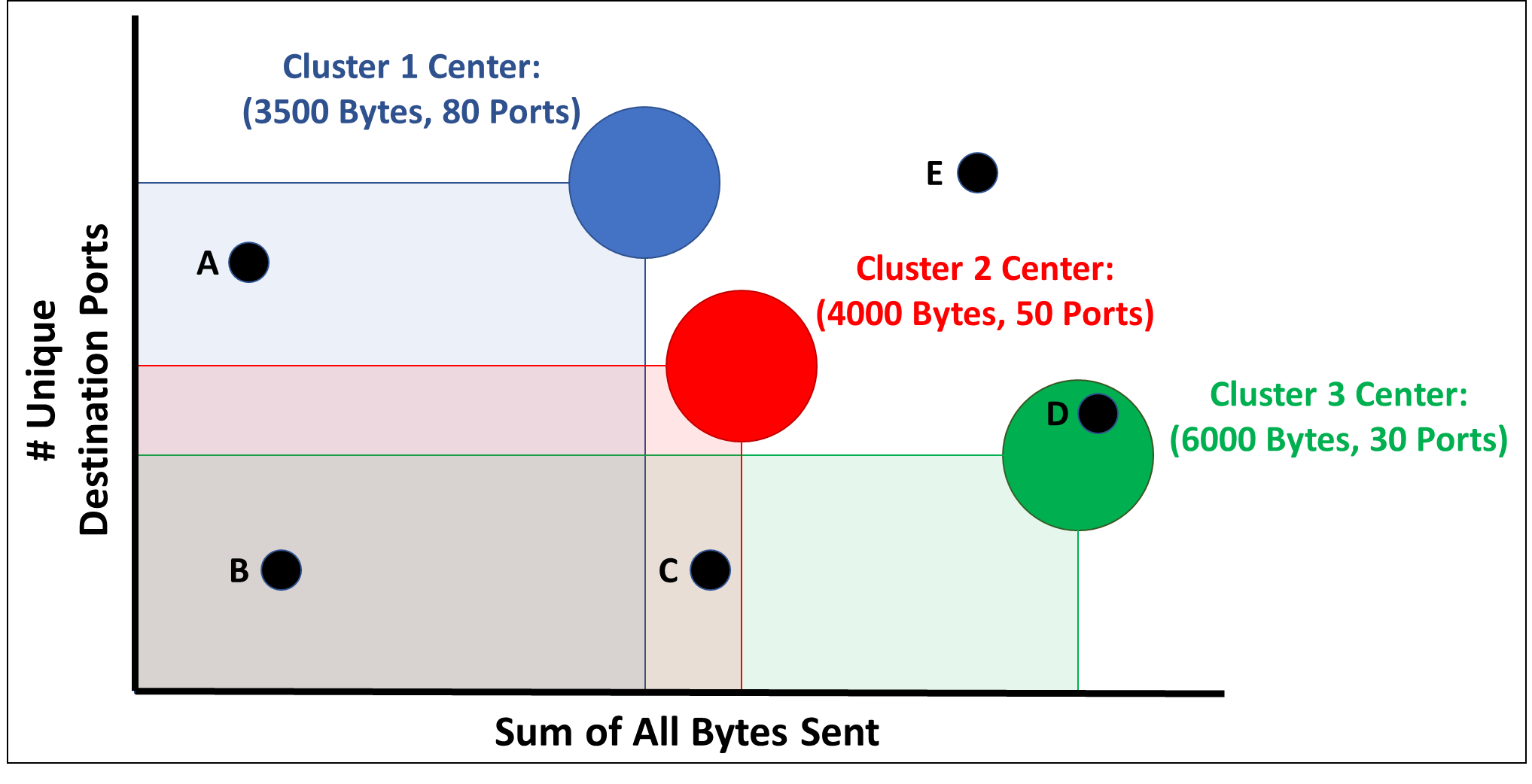


Figure 8 - Example Diagram showing an Anomaly (E) using 2 Features

### Generating Anomalies

We generated 2 types of attacks: a ICMP Ping flood and a UDP Port Scan. An ICMP Ping Flood generates a large volume of ICMP Pings and sends them all to a single destination. A UDP Port scan attempts to initiate a conversation from the attacking device to the victim device on every port from port 1 through port 65535.

We used hping3 to send 100,000 pings from h11 (10.0.0.1) to h12 (10.0.0.2).

sudo hping3 -V -c 100000 -d 9000 -S -w 64 --flood 10.0.0.5

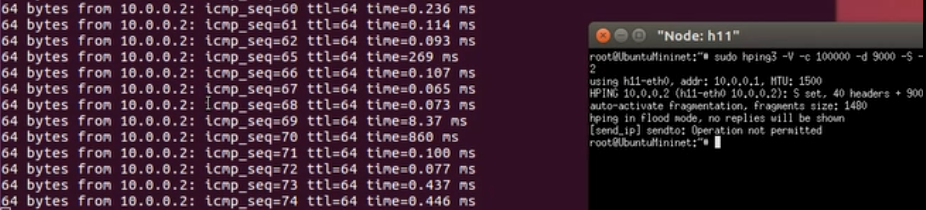


Figure 9 - Output from hping3. Blocked at icmp\_seq=74

When PySpark received the data after the interval, it determined an anomaly was present and sent the REST API call to ONOS to block. The below screenshot shows the full output from our PySpark application.

The new data comes in from 10.0.0.1 with 5 unique destination ports, 7 unique destination IPs, and 39,165,380 bytes. It was assigned to cluster 2. The distance from the center of that cluster is 1790.82 and the threshold distance is 69.66. Since 1790.82 > 69.66, PySpark sent the REST API call to ONOS.

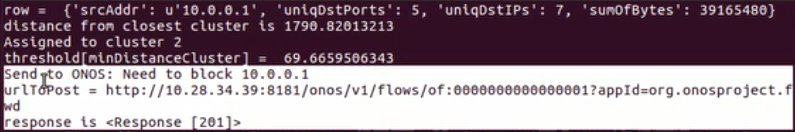


Figure 10 - PySpark Application showing REST API Call to ONOS

We used Python’s Scapy library to perform the UDP Port Scan from h21 (10.0.0.4) to h22 (10.0.0.5)

for dst\_port in range(1, 65536):

sr(IP(dst=dst\_ip)/UDP(dport=dst\_port),retry=0,timeout=0)

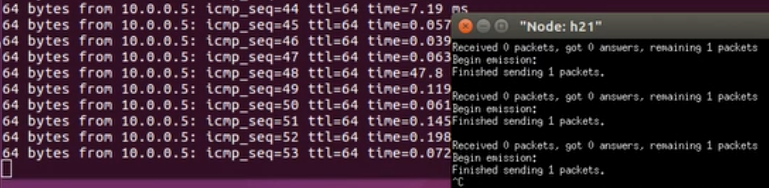


Figure 11 - Output from Scapy Port Scan. Blocked at icmp\_seq=53

### ONOS REST API Call

When an anomaly is detected, our PySpark application:

* Queries ONOS via REST API to determine which Open vSwitch the attacking device is connected to.
* Sends a REST API call directly to ONOS providing:
  + The IP address of the attacking device.
  + The Open vSwitch the attacking device is connected to.

ONOS receives the REST API call and adds a Flow to the Open vSwitch to block all traffic with source IP of the attacking device.

The REST API call sent is of the form:

blockData = {

"priority": 40000,

"timeout": 0,

"isPermanent": "true",

"deviceId": ipToSwitchMap[anomalyIP],

"treatment": {}, # blank treatment means drop traffic.

"selector": {

"criteria": [

{

"type": "ETH\_TYPE",

"ethType": "0x0800" # IPv4 Traffic.

},

{

"type": "IPV4\_SRC",

"ip": "{0}/32".format(anomalyIP) # Must include subnet mask.

}

]

}

}

That blockData is sent in JSON format to:

http://10.28.34.39:8181/onos/v1/flows/{0}?appId=org.onosproject.fwd

Where 10.28.34.39 is my ONOS controller, and {0} is the Open vSwitch connected to the source device generating the anomaly.

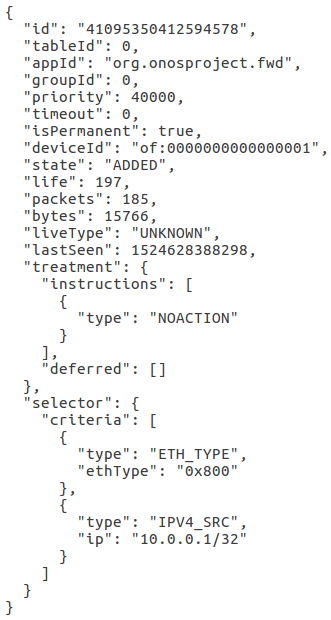


Figure 12 - Flow Entry sent by ONOS to drop all traffic at Open vSwitch of:0001 from Source IPv4 10.0.0.1

### Side Effect of Anomaly Detection

In some cases, both the attacker device and the victim device are both blocked. When the attacker sends messages to the victim, the victim tries to reply to each message. This can result in an “anomalous” amount of traffic coming from the victim.

A potential solution to this issue is to send the anomalous traffic to a group who can investigate and determine whether traffic needed to be blocked or if it was safe. This group could identify which device was the attacker and which was the victim.

### Data Visualization

The below image is a sample of the visualization Kibana provides for NetFlow data. The inner rings show the IP addresses and the outer rings show Port and Protocol information. Kibana provides additional visualizations for TCP Flags, VLANs, Autonomous Systems, Countries and Cities, etc.

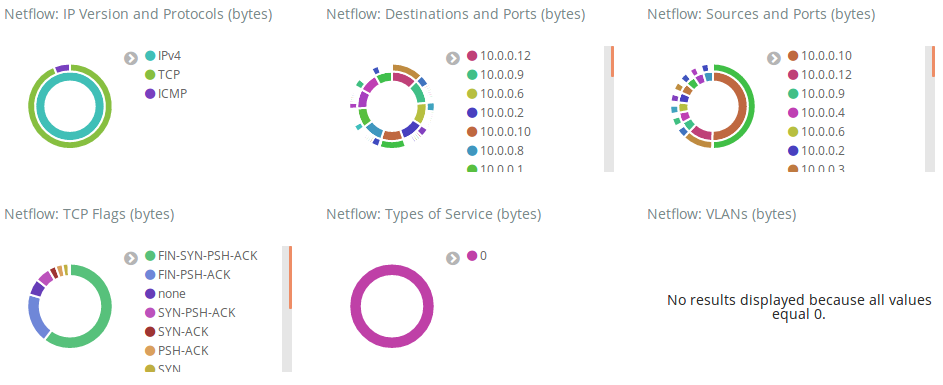


Figure 13 - Kibana NetFlow Data Visualization

# Future Scope

As mentioned earlier, our PySpark application could send an alert to a group to investigate the anomaly before immediately blocking the traffic. With our current implementation sometimes both the attacker and the victim were blocked, due to the victim trying to respond to all the attacker’s messages. When new types of “normal” traffic are added to our network, they could be detected as “anomalous”. Our “normal” network data is very limited.

We could also attempt other Machine Learning algorithms and compare them with our implementation.

Once we captured the 36 hours of initial data, we never put new data into Elasticsearch. Ideally, we would continue to store new “normal” traffic and periodically update our model.

# Lessons Learned

We had to learn how to install and configure ONOS. The documentation on the website is out-of-date and information is not all in one location. It took many hours to find the necessary resources and commands to correctly configure ONOS. We also initially created an internal application for ONOS. We had to learn how to build the application using Maven and load and enable the application into ONOS.

We also had to learn the necessary components to form a Big Data architecture. This took many hours of research to understand the roles of each component and how they worked together.

An important part of our Big Data Infrastructure was the StreamSets application. This was essential for us to get NetFlow data into Apache Kafka.

We also had to learn how to perform feature extraction, feature engineering, and Machine Learning in PySpark.

# Conclusion

We were successful in our implementation of the architecture. Our Anomaly-based IDS was able to correctly identify anomalies and correctly allow “normal” traffic.

The Software Defined Networking course was excellent and provided us with many opportunities to learn about exciting trends and technologies in the industry. We appreciate the opportunity to choose our own projects and explore what could be done with SDN.

# References

* Project source code and other files:

<https://github.com/blynotes/CS6301_SDN>

* The paper that inspired our initial project idea:

<http://shura.shu.ac.uk/16558/1/Pranggono-MachineLearningBasedIntrusionDetectionSystem(AM).pdf>

* The paper that gave us ideas on features to engineer from NetFlow data:

<https://www.hindawi.com/journals/scn/2017/6047053/>

* ONOS YouTube Channel:

<https://www.youtube.com/channel/UCTQaOpqno48GTimdwyfytFw>

* ONOS Wiki pages:

<https://wiki.onosproject.org/>

* Mininet walkthrough:

<http://mininet.org/walkthrough/>

* NetFlow version 5 reference:

<https://www.plixer.com/support/netflow-v5/>

* Enabling NetFlow on Open vSwitch with multiple targets:

<http://blog.shin.do/2014/12/netflow-on-open-vswitch-2/>

* StreamSets:

<https://streamsets.com/opensource/>

* Apache Zookeeper & Apache Kafka Download and Quickstart:

<https://kafka.apache.org/quickstart>

* Apache Spark documentation:

<https://spark.apache.org/docs/latest/>

* PySpark documentation:

<https://spark.apache.org/docs/2.3.0/api/python/pyspark.html>

* Elasticsearch

<https://www.elastic.co/products/elasticsearch>

* Kibana

<https://www.elastic.co/products/kibana>

* Logstash

<https://www.elastic.co/products/logstash>

* Scapy (Python Library) Documentation

<https://scapy.readthedocs.io/en/latest/usage.html>

# Appendix A – Guide Descriptions

**Active Guides:**

* App Installation Guide.docx

How to install and configure everything in the Application VM.

* Mininet VM Guide.docx

Install required packages into the VM to run Mininet and trigger the anomalies.

* NetFlow Guide.docx

Guide for how to configure NetFlow on Open vSwitch.

* ONOS 1.12 installation Guide.docx

3 methods of installing and configuring ONOS:

Option 1 installs an OVA file and provides a link to a Distributed ONOS tutorial.

Option 2 installs ONOS as a service (I did not get this to work).

Option 3 is the recommended option. There is also information for configuring IntelliJ if building an Internal ONOS application.

* ONOS Rest API Guide.docx

Contains information on how to view a nice webpage on localhost (after launching ONOS) to query the ONOS REST API.

* Run Applications Guide.docx

How to start and stop all applications in the Big Data pipeline and run the demo.

**Old Guides (in OldGuides subfolder). We did not use these Guides, but others may find the information useful for other projects:**

* Big Data Pipeline Installation Guide.docx

Later replaced by “App Installation Guide.docx”.

Contains information about configuring Apache Flume and IPFIXCol (both unused in our final project).

* Connecting VMs to Mininet as Hosts.docx

Useful for having external VMs be a host connected to an Open vSwitch in Mininet.

* Creating Applications for ONOS 1.12.docx

Useful for creating Internal ONOS Applications.

* IPFIX Guide.docx

How to configure IPFIX on Open vSwitch.

* sFlow Guide.docx

How to configure sFlow on Open vSwitch.

# Appendix B – Code File Descriptions

* Client.py

Generates random “Normal” traffic.

* index\_ES.txt

Information placed into Kibana Dev Tool to create our Elasticsearch index.

* scapyPortScan.py

Use Python library Scapy to perform a UDP port scan from port 1 to port 65535 on the target device.

* Server.py

Receives messages from Client.py from other hosts and responds.

* setup\_topo.py

Setup Mininet topology, configure Open vSwitches with NetFlow, call Client.py and Server.py for each Mininet host.

* sparkKafka.py

Perform feature engineering to get our features and send to Elasticsearch.

* sparkMachineLearning.py

Train K-Means algorithm on data in Elasticsearch, perform feature engineering on new data, standardize new data and check if anomaly. If anomaly detected, send REST API call to ONOS.