PROJECT REPORT

Distributed Network Intrusion Detection

FOR ADVANCED METERING INFRASTRUCTURE

Submitted by-

Abhiraj Bishnoi	**PRN: 12070121602**
Indrayudh Roy	**PRN: 12070121335**
P Santy Blesson	**PRN: 12070121654**
Parth Gilitwala	**PRN: 12070121131**

Under the Guidance of-

Dr. Himanshu Agrawal
Associate Professor
CS and IT Department

Department of Computer Science Engineering
SYMBIOSIS INSTITUTE OF TECHNOLOGY, PUNE

January 2016- June 2016

DECLARATION

We hereby declare that the project work entitled **— Distributed Network Intrusion Detection for Advanced Metering Infrastructure** is an authentic record of our own work carried out as a requirement of six months project semester for the award of B.Tech degree in Computer Science Engineering at Symbiosis Institute of Technology Pune, affiliated to Symbiosis International University Pune, under the guidance of Dr. Himanshu Agrawal, during January 2016 to June 2016.

Abhiraj Bishnoi	**Indrayudh Roy**	**P Santy Blesson**	**Parth Gilitwala**
12070121602	12070121335	12070121654	12070121131

Date: ___________________

Certified that the above statement made by the students is correct to the best of our knowledge and belief.

Dr. Himanshu Agrawal	Dr. Shraddha Phansalkar	Dr. T. P. Singh
Associate Professor	Head of Department	Director
CS and IT Department	CS and IT Department	SIT, Pune
SIT, Pune	SIT, Pune	

TABLE OF CONTENTS

Acknowledgement

Our project titled “Distributed Network Intrusion Detection System for Advanced Metering Infrastructure” would not have been possible to complete without the guidance of many wonderful people surrounding us. Our first mention would undoubtedly go to our Project Guide Dr. Himanshu Agarwal for his constant guidance and support throughout the entire semester. The numerous discussions we had with him on a weekly basis helped to bring the much needed clarity at the beginning of the project phase when we struggled with zeroing in on the specific problem. He helped bring out the best out in us and was a superior source of inspiration and encouragement.

Our thanks extends to the Open Source Software communities with whom we’ve had endless discussions on IRC. The passion that software developers have towards writing code and making it available for free inspires us and directs us in making our own tool open source. We thank the head of our department Dr. Shraddha Phansalkar for making it easier for us to deploy our project in the college. We would like to extend our gratitude towards the lab assistants and server room administrators for complying with our requests for our testbed.

Abstract

The use of secure protocols and the enforcement of strong security properties have the potential to prevent vulnerabilities from being exploited and from having costly consequences. Prevention is one aspect of a comprehensive approach that must also include the development of a complete monitoring solution. Our project develops a system to monitor the flow of packets in the network and detect any anomalies based on certain preset rules and algorithms. In its most primitive way the algorithm would work in the following manner:

$$\begin{tikzcd}
\text{Extract a packet} \arrow[d] \\
\text{Search for any anomalies} \arrow[d] \\
\text{Report anomalies}
\end{tikzcd}$$

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Overview

The problem of designing a network-based intrusion detection system is as much in the availability of good data to validate the system, as much as in its implementation. In 1999, the KDD Cup Challenge was held, and the eponymous KDD '99 data set became academia's standard for conducting research on intrusion detection. In the intervening years, multiple technological advances – both helpful and harmful to intrusion detection – have taken place; most notably, the rise of distributed networks of all forms – from pure software-service clouds to Advanced Metering Infrastructure systems – and the rise of streaming/real-time data. These necessitate another look at two of the fundamental issues in such a network IDS – the data and the implementation method. We base our intrusion detection system on these two premises – a distributed model of computation, and a modern data set which provides more realistic data and results as compared to prior approaches.

ISCX Dataset and Apache Storm

We used the ISCX data set; its summary states –

> *"In network intrusion detection (IDS), anomaly-based approaches in particular suffer from accurate evaluation, comparison, and deployment which originates from the scarcity of adequate datasets. Many such datasets are internal and cannot be shared due to privacy issues, others are heavily anonymized and do not reflect current trends, or they lack certain statistical characteristics."*

After selecting an anomaly-detection based approach – aided by machine learning – to be our methodology, we realized the "scarcity of adequate datasets", as outlined above. Further –

> *"These deficiencies are primarily the reasons why a perfect dataset is yet to exist. Thus, researchers must resort to datasets which they can obtain that are often suboptimal."*

With the use of the ISCX data set, we also hope to employ the state-of-the-art in terms of data quality that also remains relevant in the future.

With an eye towards improving performance, we wished to make the deployed system a distributed cluster. For this, Apache Storm proved to be an ideal solution, providing a way to check for anomalies in a stream of incoming data (i.e. a network) packets. With a multi-node Storm cluster we can distribute the computation – in the form of decision-making classifiers – throughout the various nodes.

1.2 Motivation

As the Internet continues to grow in terms of scale, complexity, openness and accessibility, Network Security assumes a pivotal role in the Society of the future. With the proliferation of devices connected to the Internet, and the advent of the 'Internet of Things', the use cases for Internet connected devices continues to grow and increasingly complex mechanisms are being developed to efficiently use this technology to improve existing services. One good example of this is the Advanced Metering Infrastructure (AMI), which uses Internet connected sensors and predictive analytics to optimize power generation and distribution, reducing costs for consumers and producers alike. It does so by enabling the Power industry to function using Just-In-Time production principles that drastically reduce the cost of generation of power. It also lets consumers monitor their usage patterns and has quite a few interesting use cases in the domain of smart homes. With any opportunity, comes associated threats and the AMI has its fair share when it comes to securing the infrastructure against network based attacks that may even result in financial losses and power theft.

It is here, that we wish to study the current state of the art, research on existing Intrusion Detection mechanisms and apply our learning towards the design and implementation of a Distributed Statistical Anomaly based Intrusion Detection System for Advanced Metering Infrastructure (AMI) that works on the foundational principles of Machine Learning. The primary motivation towards advocating a distributed approach is the overall traffic of the Internet is bound to increase which translates into a direct increase in the amount of traffic that the individual sub-networks of the Internet are expected to handle. This also results in an increase in the amount of traffic that is to be analyzed by the Intrusion Detection System and if not designed well, can lead to a Bottleneck. A cluster based solution offers the best return on investment, provides scalability and fault-tolerance in addition to providing scope for parallelization of Computations that result in a direct increase in system throughput, responsiveness and performance.

1.3 Problem Statement

Our problem is composed of a number of interdependent issues at various granularities:

- Protect a home or workplace intranet from the external internet
- Design it as a layer of protection complementing the firewall
- Classify normal network usage
- Detect deviations from such “normal” usage, i.e., intruders
- Refine the classifier models for best results
- Deploy a multi-node Storm cluster with working topology
- Add automation and bookkeeping tasks

The main task for any NIDS is to provide an efficient handling of the network’s security. This also involves optimizing the usage of network resources so as not to impede the network through unnecessarily large consumption due to itself. This is even more critical in AMI networks where attacks exist specifically to rapidly leech or slowly drain resources such as battery power and network bandwidth from the network nodes. Our IDS is thus assumed to have access to an accumulator node, such as a switch on an intranet or a head-end node in AMI.

We designed our IDS to also be easily extended to an Intrusion Prevention System (IPS) in the future, if required, by co-coordinating with the firewall and making necessary configuration changes. An example of this would be the auto-assignment of open and closed ports in response to “everyday” application usage as compared to an insider probing ports.

Out of the various options available to us for detecting intruders, we have chosen statistical analysis for anomaly detection. Various mathematical classifiers provide an opportunity for our machines to undergo supervised learning. The partitioned data will be used both for training our cluster on various attack types as well as testing it thereafter.

Different classifiers provide different trade-offs, with respect to feature availability, computation time, and simplicity/fit of implementation. Multiple classifiers such as decision trees, support vector machines, etc. can be tried out for optimal choices.

The culmination lies in the planned test bed – a five-node Apache Storm cluster. The topology contains a master and slave nodes that communicate via a middleman; it also remains extensible in terms of current and future laboratory infrastructure – it will be very easy to scale up the cluster by adding any machine on the network.

Storm provides a first layer of distributed automation; carrying that forward to improve the system is one such open-ended problem task. Additionally, network security inherently being a task suited to administration, a reporting mechanism for intrusions, policy violations, etc. is also needed.

1.4 Contribution:

Our primary contributions can be outlines as follows:

1. **Extensible Test-Bed**

We have set up a first-of-its-kind laboratory cluster in the college, for further study into the security of distributed networks as well as provided a working runtime system for any test demonstrations. Our design easily incorporates any number of new nodes – making it a very easy to maintain and extensible test-bed.

2. **Distributed IDS using State-of-the-Art Technology**

We have built and deployed a working IDS using a five-node Apache Storm cluster. We feel that Apache Storm – being heretofore underutilized, yet very powerful as a real-time, distributed platform – along with its open source nature provides a promising path towards future work, as well as providing a non-proprietary, commodity-hardware based implementation approach – something that is vital to consider during scaling up with the network.

3. **Learning Environment**

We have provided a fully functional environment on a live network for the study of network traffic security and its related issues. The IDS can be used for displaying and observing network traffic, patterns and analysis, real-time behavior on the entry of an intruder, etc. Various tests can be performed by changing network policy and node activity. This will provide an easy learning tool for studying intrusion detection.

4. **ISCX Data Set – A Modern Data Source**

The ISCX data set is a new and important benchmark entry into the pool of realistically-generated data sets which have been generated by simulated and non-simulated traffic. The resultant analysis of the data is also an exploration into the quality of the data itself as well as a validation of our methodology along with the efficacy of the algorithm used. It is important to remain relevant with recent attacks and network trends in a field like security and likewise promote the use of accurate, high-quality data.

In addition to these, our IDS contains these features, which were the result of critique and selection of various design trade-offs:

- Statistical, anomaly-based:

This subclass of system works by identifying 'patterns' in the packets flowing through a Network and classifying a packet as a potential threat based on its position relative to a baseline that is calculated using sophisticated statistical techniques.

In a nutshell, these systems observe packets to come up with an underlying mathematical representation of well-formed packets that don't pose a threat to the network and anomalous packets that are potentially dangerous using sophisticated statistical models and artificial intelligence.

The intersection of Statistics and Artificial Intelligence culminates into an interesting field of Computer Science known as Machine Learning, which essentially deals with teaching computers to learn something without explicitly programming them to do so. Machine Learning techniques provide an interesting solution to the problem at hand and may be used to ‘learn’ patterns in network traffic by analyzing empirical data and classify them as potential threats or well-formed packets.

These systems compute a numerical value for each incoming packet using a mathematical abstraction (Feature Extraction and reduction), and compare this value against the pre-calculated Baseline (which is the result of a mathematical abstraction of well-formed packets and anomalous packets) to test for potential threats. If the value is found to be above the threshold (baseline), this indicates anomalous behavior, and the packet is flagged as being potentially dangerous.
Statistical Anomaly based Intrusion Detection Systems do a much better job at protecting the network against unknown attacks that haven’t been previously observed. However, these systems tend to pay a performance penalty and are comparatively slower than Signature-based Intrusion Detection Systems and tend to have high false positive rates.

- Supervised machine learning-based:

The system was designed to have pre-trained machines with the requisite knowledge to detect anomalies on new data. The data on which it learns from the ISCX data set is designed to “generalize” well so as to predict successfully for various classes of unseen data. Additionally, prediction should not fit the data to its knowledge to tightly, but remain flexible for security threats within new packets. The parameters of supervised learning were various function constraints.

- Focus on performance via distribution:

The Storm topology along with quick processing of packet tuples results in an approach that doesn’t sacrifice accuracy for quick results. Logical nodes are dedicated to detecting one type of attack each and the system assures high availability and robustness in addition to processing speed.

- Well-tagged data:

The pre-processing engine that was required to be developed is an independent module that readies the test data for the learning system. We were fortunate to receive well-tagged XML data of the relevant features from the ISCX data set, and we have propagated the use of this data – this being a modern data source, especially relevant for IoT and AMI systems in the future.

The outline of our report continues as follows: first – the literature review and our base paper, second – we define the basic principles of intrusion detection, third – we evaluate, discuss, and select the platforms and technologies we shall use, fourth – the design phase of our IDS which involves classifier selection, machine learning models, etc. and fifth – the implementation phase which culminates with deployment. Lastly, we summarize and discuss the future scope of our work.

CHAPTER 2: LITERATURE REVIEW

The deliverable of our project is a proof of concept. The basis of our idea evolved from gleaning information about the past and the current research being carried out on Intrusion Detection Systems and Security for Advanced Metering Infrastructure. We devoted a significant amount of our time in reading research papers, communicating with authors of various papers, holding discussions with our Project Guide.

Several areas which we explored pertaining to our research were:

1. Security for Internet of Things
2. Types of Intrusion Detection
3. Datasets for IDS
4. Distributed and Real-time processing
5. Machine learning for IDS

Following are the list of research papers that helped us in the formation of our proof of concept:

P1: *"Intrusion detection and Big Heterogeneous Data: A Survey"* – Richard Zuech, Taghi M Khoshgoftaar and Randall Wald

Intrusion Detection has been heavily studied in both industry and academia, but cybersecurity analysts still desire much more alert accuracy and overall threat analysis in order to secure their systems within cyberspace. Improvements to Intrusion Detection could be achieved by embracing a more comprehensive approach in monitoring security events from many different heterogeneous sources. Correlating security events from heterogeneous sources can grant a more holistic view and greater situational awareness of cyber threats. One problem with this approach is that currently, even a single event source (e.g., network traffic) can experience Big Data challenges when considered alone. Attempts to use more heterogeneous data sources pose an even greater Big Data challenge. Big Data technologies for Intrusion Detection can help solve these Big Heterogeneous Data challenges. In this paper, we review the scope of works considering the problem of heterogeneous data and in particular Big Heterogeneous Data. We discuss the specific issues of Data Fusion, Heterogeneous Intrusion Detection Architectures, and Security Information and Event Management (SIEM) systems, as well as presenting areas where more research opportunities exist. Overall, both cyber threat analysis and cyber intelligence could be enhanced by correlating security events across many diverse heterogeneous sources.

P2: "*Network-Level Security and Privacy Control for Smart-Home IoT Devices*" - Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli and Olivier Mehani

Abstract

The increasing uptake of smart home appliances, such as lights, smoke-alarms, power switches, baby monitors, and weighing scales, raises privacy and security concerns at unprecedented scale, allowing legitimate and illegitimate entities to snoop and intrude into the family's activities. In this paper we first illustrate these threats using real devices currently available in the market. We then argue that as more such devices emerge, the attack vectors increase, and ensuring privacy/security of the house becomes more challenging. We therefore advocate that device-level protections be augmented with network-level security solutions that can monitor network activity to detect suspicious behavior. We further propose that software defined networking technology be used to dynamically block/quarantine devices, based on their network activity and on the context within the house such as time-of-day or occupancy-level. We believe our network-centric approach can augment device-centric security for the emerging smart-home.

P3: "*An Artificial Neural Network based Intrusion Detection System and Classification of Attacks*" - Devikrishna K S and Ramakrishna B B

Abstract

Network security is becoming an issue of paramount importance in the information technology era. Nowadays with the dramatic growth of communication and computer networks, security has become a critical subject for computer system. Intrusion detection is the art of detecting computer abuse and any attempt to break the networks. Intrusion detection system is an effective security tool that helps to prevent unauthorized access to network resources by analyzing the network traffic. Different algorithms, methods and applications are created and implemented to solve the problem of detecting the attacks in intrusion detection systems. Most methods detect attacks and categorize in two groups, normal or threat. One of the most promising areas of research in the area of Intrusion Detection deals with the applications of the Artificial Intelligence (AI) techniques. This proposed system presents a new approach of intrusion detection system based on artificial neural network. Multi-Layer Perceptron (MLP) architecture is used for Intrusion Detection System. The performance and evaluations are performed by using the set of benchmark data from a KDD (Knowledge discovery in Database) dataset. The proposed system detects the attacks and classifies them in six groups.

P4: "*Real-time Hybrid Intrusion Detection System using Apache Storm*" - Goutam Mylavarapu, Dr. Johnson Thomas and Ashwin Kumar TK

Abstract

Networks are prone to intrusions and detecting intruders on the Internet, is a major problem. Many Intrusion Detection Systems have been proposed to detect these intrusions. However, as the Internet grows day by day, there is a huge amount of data (big data) that needs to be processed to detect intruders. For this reason, intrusion detection has to be done in real-time before intruders can inflict damage, and previous detection systems do not satisfy this need for big data. Using Apache Storm, a Real time Hybrid Intrusion Detection System has been developed in our work. Apache Storm serves as a distributed, fault tolerant, real time big data stream processor. The hybrid detection system consists of two neural networks. The CC4 instantaneous neural network acts as an anomaly-based detection for unknown attacks and the Multi-Layer Perceptron neural network acts as a misuse-based detection for known attacks. Based on the outputs from these two neural networks, the incoming data will be classified as an ”attack” or ”normal”. We found the average accuracy of hybrid detection system is 89% with a 4.32% false positive rate. This model is appropriate for real time detection since Apache Storm acts as a real time streaming processor, which can also handle big data.

P5: "*Toward developing a systematic approach to generate benchmark datasets for intrusion detection*" - A. Shiravi, H. Shiravi, M. Tavallaee and A. A. Ghorbani

Abstract

In network intrusion detection, anomaly-based approaches in particular suffer from accurate evaluation, comparison, and deployment which originates from the scarcity of adequate datasets. Many such datasets are internal and cannot be shared due to privacy issues, others are heavily anonymized and do not reflect current trends, or they lack certain statistical characteristics. These deficiencies are primarily the reasons why a perfect dataset is yet to exist. Thus, researchers must resort to datasets that are often suboptimal. As network behaviors and patterns change and intrusions evolve, it has very much become necessary to move away from static and one-time datasets toward more dynamically generated datasets which not only reflect the traffic compositions and intrusions of that time, but are also modifiable, extensible, and reproducible. In this paper, a systematic approach to generate the required datasets is introduced to address this need. The underlying notion is based on the concept of profiles which contain detailed descriptions of intrusions and abstract distribution models for applications, protocols, or lower level network entities. Real traces are analyzed to create profiles for agents that generate real traffic for HTTP, SMTP, SSH, IMAP, POP3, and FTP. In this regard, a set of guidelines is established to outline valid datasets, which set the basis for generating profiles. These guidelines are vital for the effectiveness of the dataset in terms of realism, evaluation capabilities, total capture, completeness, and malicious activity. The profiles are then employed in an experiment to generate the desirable dataset in a testbed environment. Various multi-stage attacks scenarios were subsequently carried out to supply the anomalous portion of the dataset. The intent for this dataset is to assist various researchers in acquiring datasets of this kind for testing, evaluation, and comparison purposes, through sharing the generated datasets and profiles.

P6: "*Real-time network-based anomaly intrusion detection*" - R. Balupari, B. Tjaden, S. Ostermann, M. Bykova, and A. Mitchell

Abstract

The global internet has made computer systems world-wide vulnerable to an ever-changing array of attacks. A new approach to perform real-time network-based anomaly intrusion detection is presented in this paper. Real-time Tcptrace generates data streams which are analysed to detect network-based attacks. Real-time Tcptrace periodically reports statistics on all the open TCP/IP connections in the network. Then, using the Abnormality Factor method, statistical profiles are built for the normal behavior of the network services. Abnormal activity is then flagged as an intrusion. This approach has the advantage of being able to monitor any service without the prior knowledge of modelling its behavior. The paper presents interesting results and evaluation of the approach by conducting experiments using the MIT Lincoln lab evaluation data.

P7: "*A New Intrusion Detection Benchmarking System"* - Richard Zuech, Taghi M. Khoshgoftaar, Naeem Seliya, Maryam M. Najafabadi, and Clifford Kemp

Abstract

This paper presents a new quality network-based dataset for the purpose of intrusion detection system (IDS) evaluation, and is referred to as the IRSC (Indian River State College) dataset. Network flows and full packet capture (FPC) data are collected creating two types of datasets. The IRSC dataset represents a real-world network that gives us the advantage of collecting actual normal and attack traffic data reflecting a real-world environment. The attack portion of the traffic contains both controlled attacks (which are intentional attacks generated by our team) and uncontrolled attacks (which are real attacks on the IRSC network not created by our team). One main goal is to produce a reliable dataset with normal and attack traffic that is realistic and meets real world criteria. Another major goal is to produce a systematic process which would allow others to generate high quality IDS evaluation datasets. Our work's main contributions are that we have both accurate labeling through the inclusion of controlled attacks, and also realistic data by including real-world attacks.

P8: "*Exploring discrepancies in findings obtained with the KDD Cup '99 data set"* - Vegard Engen, Jonathan Vincent and Keith Phalp

Abstract

The KDD Cup '99 data set has been widely used to evaluate intrusion detection prototypes, most based on machine learning techniques, for nearly a decade. The data set served well in the KDD Cup '99 competition to demonstrate that machine learning can be useful in intrusion detection systems. However, there are discrepancies in the findings reported in the literature. Further, some researchers have published criticisms of the data (and the DARPA data from which the KDD Cup '99 data has been derived), questioning the validity of results obtained with this data. Despite the criticisms, researchers continue to use the data due to a lack of better publicly available alternatives. Hence, it is important to identify the value of the data set and the findings from the extensive body of research based on it, which has largely been ignored by the existing critiques. This paper reports on an empirical investigation, demonstrating the impact of several methodological differences in the publicly available subsets, which uncovers several underlying causes of the discrepancy in the results reported in the literature. These findings allow us to better interpret the current body of research, and inform recommendations for future use of the data set.

P9: “*An introduction to the WEKA data mining system”* - Z. Markov and I. Russell

Abstract
This is a proposal for a half day tutorial on Weka, an open source Data Mining software package written in Java and available from www.cs.waikato.ac.nz/~ml/weka/index.html. The goal of the tutorial is to introduce faculty to the package and to the pedagogical possibilities for its use in the undergraduate computer science and engineering curricula. The Weka system provides a rich set of powerful Machine Learning algorithms for Data Mining tasks, some not found in commercial data mining systems. These include basic statistics and visualization tools, as well as tools for pre-processing, classification, and clustering, all available through an easy to use graphical user interface. Data Mining studies algorithms and computational paradigms that allow computers to discover structure in databases, perform prediction and forecasting, and generally improve their performance through interaction with data. Machine learning is concerned with building computer systems that have the ability to improve their performance in a given domain through experience. Machine learning and Data Mining are becoming increasingly important areas of engineering and computer science and have been successfully applied to a wide range of problems in science and engineering. Recently, acknowledging the importance of these areas in computer science and engineering, more work is being done to incorporate these areas into the undergraduate curriculum. Weka is a widely used package that is particularly popular for educational purposes. It is the companion software package of the book “Data Mining: Practical Machine Learning Tools and Techniques” by Ian H. Witten and Eibe Frank. The Weka team has been recently awarded with

the 2005 ACM SIGKDD Service Award for their development of the Weka system, including the accompanying book. As Gregory Piatetsky-Shapiro writes in the news item about this event (KDnuggets news, June 28, 2005), "Weka is a landmark system in the history of the data mining and machine learning research communities, because it is the only toolkit that has gained such widespread adoption and survived for an extended period of time (the first version of Weka was released 11 years ago)".

P10: "*Security in the Internet of Things: A Review"* – Hui Suo, Jiafu Wan, Caifeng Zou, Jianqi Liu

Abstract

In the past decade, internet of things (IoT) has been a focus of research. Security and privacy are the key issues for IoT applications, and still face some enormous challenges. In order to facilitate this emerging domain, we in brief review the research progress of IoT, and pay attention to the security. By means of deeply analyzing the security architecture and features, the security requirements are given. On the basis of these, we discuss the research status of key technologies including encryption mechanism, communication security, protecting sensor data and cryptographic algorithms, and briefly outline the challenges.

CHAPTER 3: INTRUSION DETECTION BASICS / PRINCIPLES

An intrusion detection and / or prevention system is a means of augmenting traditional lines of cyber defence. It is a device or software application that monitors network or system activities for malicious activities or policy violations and produces electronic reports to a management station.

Intrusion Detection Systems come in a variety of "flavours" and approach the goal of detecting suspicious traffic in different ways. There are network based (NIDS) and host based (HIDS) intrusion detection systems. NIDS is a network security system focusing on the attacks that come from the inside of the network (authorized users). Some systems may attempt to stop an intrusion attempt but this is neither required nor expected of a monitoring system. Intrusion detection and prevention systems (IDPS) are primarily focused on identifying possible incidents, logging information about them, and reporting attempts. In addition, organizations use these systems for other purposes, such as identifying problems with security policies, documenting existing threats and deterring individuals from violating security policies. Intrusion Detection and Prevention Systems have become a necessary addition to the security infrastructure of nearly every organization.

Intrusion Detection Systems typically record information related to observed events notify security administrators of important observed events and produce reports. Many Intrusion Detection and Prevention Systems can also respond to a detected threat by attempting to prevent it from succeeding. They use several response techniques, which involve the IDPS stopping the attack itself, changing the security environment (e.g. reconfiguring a firewall) or changing the attack's content.

Intrusion Detection and Prevention Systems are traditionally classified into two sub-classes of systems on the basis of their detection techniques, that of Signature based systems and Statistical Anomaly based systems.

3.1 Signature Based IDS:

Signature based systems essentially work on the premise that the attack has occurred before and we have a ‘signature’ of the attack, in simple terms a fairly detailed idea about the structure and content of malformed packets. This information can be encoded in the form of a ‘signature’ and stored in a database of all known attack signatures.

As packets arrive in the system, their signatures are calculated on the fly and matched against a database of packet signatures corresponding to the structure and content of all previously known attacks. If a match is found, preventive action may be taken by reconfiguring the firewall to block connections from the source IP address or the packet may be dropped. Some systems are passive in nature, in the sense that they don’t take preventive actions on their own. In these types of systems, the System administrator may be notified of a possible breach and further action is left to his/her discretion.

These types of Intrusion Detection Systems are easier to build and very efficient in terms of protecting against known threats, but naturally they are unable to prevent the Network or the Host from unknown attacks that haven’t previously been observed and this is their biggest drawback.

3.2 Statistical Anomaly Based IDS:

This subclass of system works by identifying ‘patterns’ in the packets flowing through a Network and classifying a packet as a potential threat based on its position relative to a baseline that is calculated using sophisticated statistical techniques.

In a nutshell, these systems observe packets to come up with an underlying mathematical representation of well-formed packets that don’t pose a threat to the network and anomalous packets that are potentially dangerous using sophisticated statistical models and artificial intelligence.

The intersection of Statistics and Artificial Intelligence culminates into an interesting field of Computer Science known as Machine Learning, which essentially deals with teaching computers to learn something without explicitly programming them to do so. Machine Learning techniques provide an interesting solution to the problem at hand and may be used to 'learn' patterns in network traffic by analysing empirical data and classify them as potential threats or well-formed packets.

These systems compute a numerical value for each incoming packet using a mathematical abstraction (Feature Extraction and reduction), and compare this value against the pre-calculated Baseline (which is the result of a mathematical abstraction of well-formed packets and anomalous packets) to test for potential threats. If the value is found to be above the threshold (baseline), this indicates anomalous behaviour, and the packet is flagged as being potentially dangerous.

Statistical Anomaly based Intrusion Detection Systems do a much better job at protecting the network against unknown attacks that haven't been previously observed. However, these systems tend to pay a performance penalty and are comparatively slower than Signature-based Intrusion Detection Systems and tend to have high false positive rates.

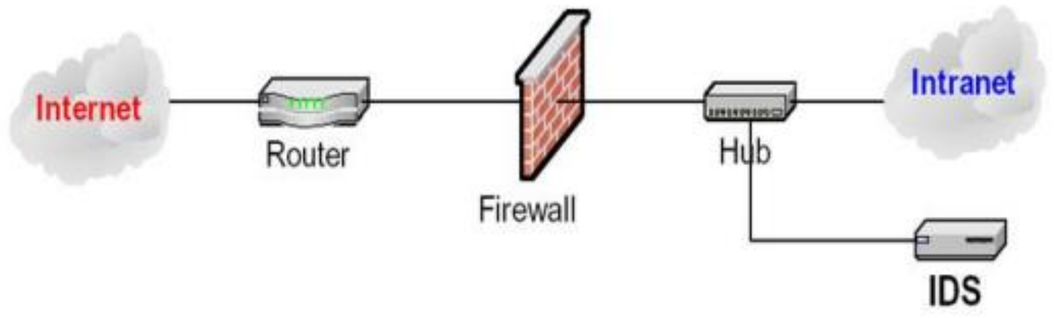


Figure 3.1 Position of an IDS in the network

Intrusions refer to the network attacks against vulnerable services, data-driven attacks on applications, host-based attacks like privilege escalation, unauthorized logins and access to sensitive files, or malware like viruses, worms and trojan horses. These actions attempt to compromise the integrity, confidentiality or availability of a resource. Intrusions result in services being denied, system failing to respond or data stolen or being lost. Intrusion detection means detecting unauthorized use of a system or attacks on a system or network. Intrusion Detection Systems are implemented in software or hardware in order to detect these activities. An Intrusion Detection System (IDS) typically operates behind the firewall as shown in the figure on the previous page, looking for patterns in network traffic that might indicate malicious activity. Thus, IDSs are used as the second and the final level of defense in any protected network against attacks that breach other defences. The need for this second layer of protection is many times questioned like “Do we need an IDS once we have a firewall?” To briefly answer this question, it is required to understand what a firewall does and does not do, and what an IDS does and does not do. This will help in realizing the need for both IDS and firewall to help in securing a network.

The existing network security solutions, including firewalls, were not designed to handle network and application layer attacks such as Denial of Service and Distributed Denial of Service attacks, worms, viruses, and Trojans. Along with the drastic growth of the Internet, the high prevalence of the threats over the Internet has been the reason for the security personnel to think of IDSs.
A basic assumption of anomaly detection is that attacks differ from normal behavior in type and amount. By defining what’s normal, any violation can be identified, whether it is part of threat model or not. However, the advantage of detecting previously unknown attacks is paid for in terms of high false-positive rates in anomaly detection systems. It is also difficult to train an anomaly detection system in highly dynamic environments. The anomaly detection systems are intrinsically complex and also there is some difficulty in determining which specific event triggered the alarms.

3.3 Terminology

Alarm filtering: The process of categorizing attack alerts produced from an IDS in order to distinguish false positives from actual attacks.

Attacker or Intruder: An entity which tries to find a way to gain unauthorized access to information, inflict harm or engage in other malicious activities.
Burglar Alarm: A signal suggesting that a system has been or is being attacked.
Clandestine user: A person who acts as a supervisor and tries to use his privileges so as to avoid being captured.
Confidence value: A value an organization places on an IDS based on past performance and analysis to help determine its ability to effectively identify an attack.
Detection Rate: The detection rate is defined as the number of intrusion instances detected by the system (True Positive) divided by the total number of intrusion instances present in the test set.
False Alarm Rate: defined as the number of 'normal' patterns classified as attacks (False Positive) divided by the total number of 'normal' patterns.
False Negative: When no alarm is raised when an attack has taken place.
False Positive: An event signaling an IDS to produce an alarm when no attack has taken place.
Masquerader: A person who attempts to gain unauthorized access to a system by pretending to be an authorized user. They are generally outside users.
Misfeasor: They are commonly internal users and can be of two types:
An authorized user with limited permissions.
A user with full permissions and who misuses their powers.
Noise: Data or interference that can trigger a false positive or obscure a true positive.
Site policy: Guidelines within an organization that control the rules and configurations of an IDS.
Site policy awareness: An IDS's ability to dynamically change its rules and configurations in response to changing environmental activity.
True Negative: An event when no attack has taken place and no detection is made.
True Positive: A legitimate attack which triggers an IDS to produce an alarm.

CHAPTER 4: PLATFORM / TECHNOLOGIES

Scikit Learn:

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

Scikit-learn is largely written in Python, with some core algorithms written in Cython to achieve performance. Support vector machines are implemented by a Cython wrapper around LIBSVM; logistic regression and linear support vector machines by a similar wrapper around LIBLINEAR.

NumPy

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. The ancestor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open source and has many contributors.

SciPy

SciPy is an open source Python library used by scientists, analysts, and engineers doing scientific computing and technical computing.

SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers and other tasks common in science and engineering.

SciPy builds on the NumPy array object and is part of the NumPy stack which includes tools like Matplotlib, pandas and SymPy. There is an expanding set of scientific computing libraries that are being added to the NumPy stack every day. This NumPy stack has similar users to other applications such as MATLAB, GNU Octave, and Scilab. The NumPy stack is also sometimes referred to as the SciPy stack.

SciPy is also a family of conferences for users and developers of these tools: SciPy (in the United States), EuroSciPy (in Europe) and SciPy.in (in India). Enthought originated the SciPy conference in the United States and continues to sponsor many of the international conferences as well as host the SciPy website.

The SciPy library is currently distributed under the BSD license, and its development is sponsored and supported by an open community of developers. It is also supported by Numfocus which is a community foundation for supporting reproducible and accessible science.

Apache Storm

Apache Storm is a distributed computation framework written predominantly in the Clojure programming language. Originally created by Nathan Marz and team at BackType, the project was open sourced after being acquired by Twitter. It uses custom created "spouts" and "bolts" to define information sources and manipulations to allow batch, distributed processing of streaming data. The initial release was on 17 September 2011.

A Storm application is designed as a "topology" in the shape of a directed acyclic graph (DAG) with spouts and bolts acting as the graph vertices. Edges on the graph are named streams and direct data from one node to another. Together, the topology acts as a data transformation pipeline. At a superficial level the general topology structure is similar to a MapReduce job, with the main difference being that data is processed in real time as opposed to in individual batches. Additionally, Storm topologies run indefinitely until killed, while a MapReduce job DAG must eventually end.

Storm became an Apache Top-Level Project in September 2014 and was previously in incubation since September 2013.

Storm Setup

Our Apache Storm setup consisted of 1 Nimbus (controller), 1 Zookeeper (coordinator), 3 supervisors (slaves).

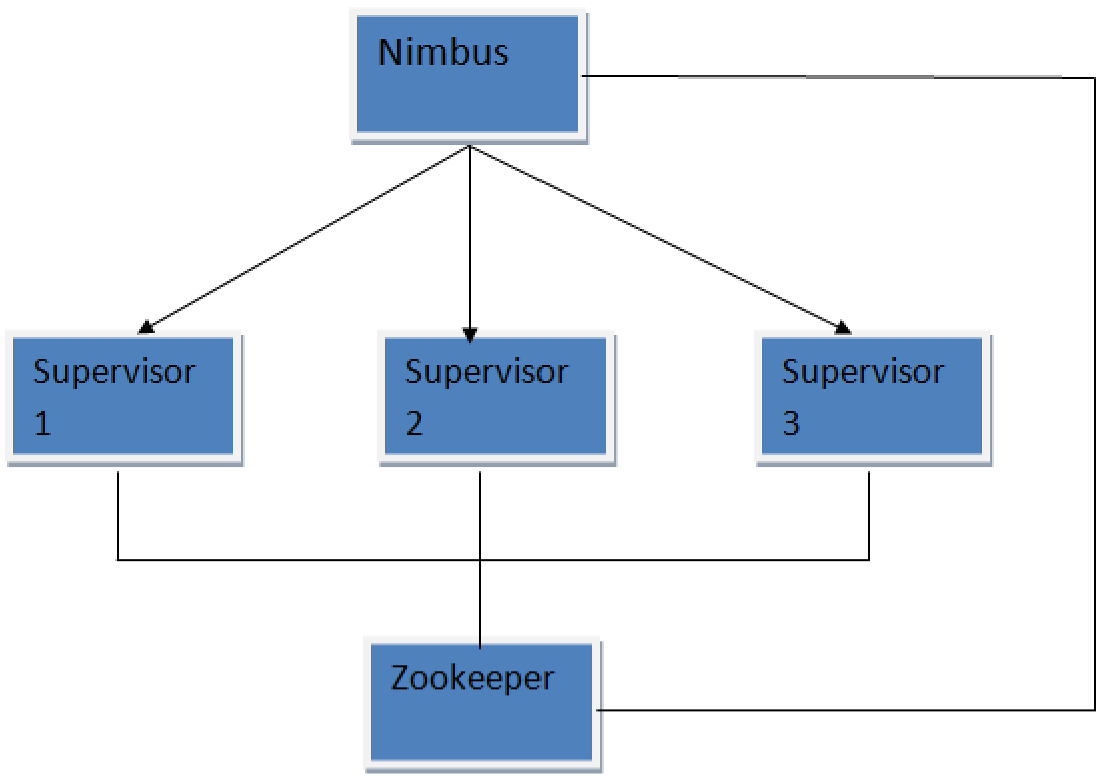


Figure 4.1 Example Storm Topology

Nimbus:

It is the master node in the setup and runs the Nimbus daemon and optionally, the Storm UI. The Nimbus daemon is responsible for assigning tasks to the worker nodes, monitoring the cluster for failures, and distributing code around the cluster. It acts as a Spout and it does the following:

- Feeds the data to the Supervisor nodes.
- Collect the result.
- Aggregate the result.

Zookeeper:

Storm uses Zookeeper for coordinating the cluster. Zookeeper is not used for message passing, so the load Storm places on Zookeeper is quite low. Single node Zookeeper clusters should be sufficient for most cases, but in case of large Storm cluster, larger Zookeeper clusters are formed. The reliability of Zookeeper rests on two basic assumptions.

1. Only a minority of servers in a deployment will fail. *Failure* in this context means a machine crash, or some error in the network that partitions a server off from the majority.

2. Deployed machines operate correctly. To operate correctly means to execute code correctly, to have clocks that work properly, and to have storage and network components that perform consistently.

Apart from using ZooKeeper for coordination purposes the Nimbus and Supervisors also store all their state in Zookeeper or on local disk.

Supervisor:

Each supervisor node runs an instance of the so-called Supervisor daemon. This daemon listens for work assigned (by Nimbus) to the node it runs on and starts/stops the worker processes as necessary. Each worker process executes a subset of a topology.

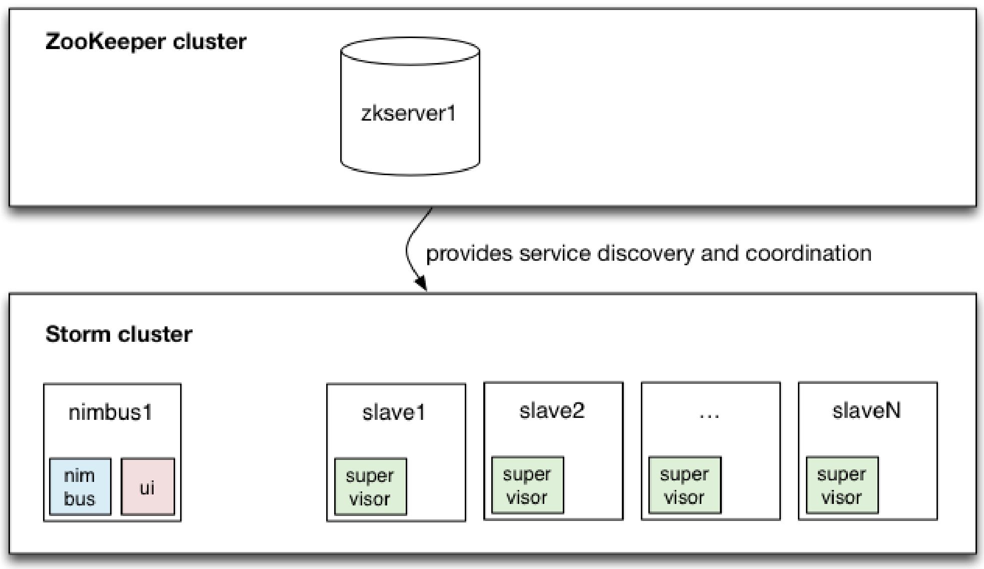


Figure 4.2 Coordination between ZooKeeper, Nimbus and Supervisor nodes

Python NMap:

Python-nmap is a python library which helps in using nmap port scanner. It allows to easily manipulate nmap scan results and will be a perfect tool for systems administrators who want to automatize scanning task and reports. It also supports nmap script outputs.

It can even be used asynchronously. Results are returned one host at a time to a callback function defined by the user.

WEKA

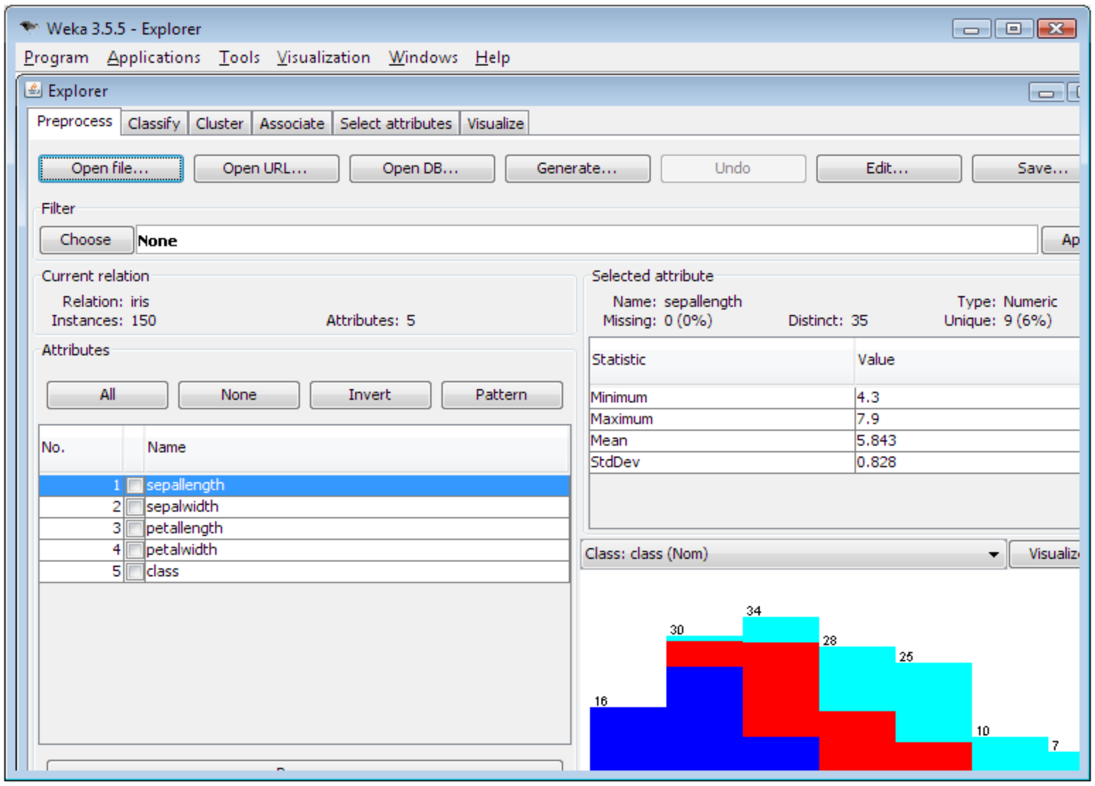
Waikato Environment for Knowledge Analysis (Weka) is a popular suite of machine learning software written in Java, developed at the University of Waikato, New Zealand. It is free software licensed under the GNU General Public License.

Weka is a workbench that contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to these functions.

Advantages of Weka include:

- Free availability under the GNU General Public License.
- Portability, since it is fully implemented in the Java programming language and thus runs on almost any modern computing platform.
- A comprehensive collection of data preprocessing and modeling techniques.
- Ease of use due to its graphical user interfaces.

Weka supports several standard data mining tasks, more specifically, data preprocessing, clustering, classification, regression, visualization, and feature selection


Weka 3.5.5 - Explorer
Program Applications Tools Visualization Windows Help
Explorer
Preprocess Classify Cluster Associate Select attributes Visualize
Open file... Open URL... Open DB... Generate... Undo Edit... Save...
Filter
Choose
Ap
All None Invert Pattern
Visualiz
16
30
34
28
25
10
7

Filter: None

Current relation
Relation: iris
Instances: 150 Attributes: 5

Selected attribute
Name: sepallength Type: Numeric
Missing: 0 (0%) Distinct: 35 Unique: 9 (6%)

Statistic	Value
Minimum	4.3
Maximum	7.9
Mean	5.843
StdDev	0.828

Attributes

No.	Name
1	sepallength
2	sepalwidth
3	petallength
4	petalwidth
5	class

Class: class (Nom)

Figure 4.3 User Interface of WEKA Data Mining Tool

Maven:

Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies. Contrary to preceding tools like Apache Ant, it uses conventions for the build procedure, and only exceptions need to be written down. An XML file describes the software project being built, its dependencies on other external modules and components, the build order, directories, and required plug-ins. It comes with pre-defined targets for performing certain well-defined tasks such as compilation of code and its packaging. Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories such as the Maven 2 Central Repository, and stores them in a local cache. This local cache of downloaded artifacts can also be updated with artifacts created by local projects. Public repositories can also be updated.

Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation.

Maven is built using a plugin-based architecture that allows it to make use of any application controllable through standard input. Theoretically, this would allow anyone to write plugins to interface with build tools (compilers, unit test tools, etc.) for any other language. In reality, support and use for languages other than Java has been minimal. Currently a plugin for the .NET framework exists and is maintained, and a C/C++ native plugin is maintained for Maven 2.

Example POM file:

```xml
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>

  <!-- project coordinates, i.e. a group of values which
       uniquely identify this project -->

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>

  <!-- library dependencies -->

  <dependencies>
    <dependency>
```

```

      <!-- coordinates of the required library -->

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>

      <!-- this dependency is only used for running and compiling tests --
>

      <scope>test</scope>

    </dependency>
  </dependencies>
</project>
```

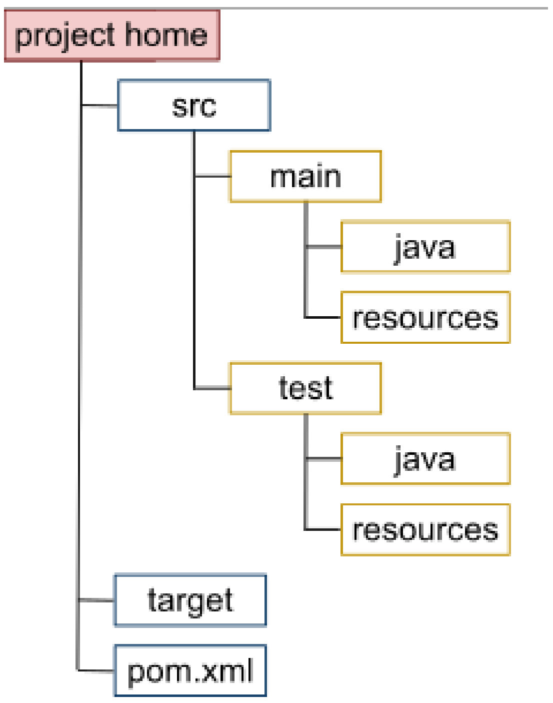


Figure 4.4 Directory structure of Maven

Python CElementtree Module

The cElementTree module is a C implementation of the ElementTree API, optimized for fast parsing and low memory use. On typical documents, cElementTree is 15-20 times faster than the Python version of ElementTree, and uses 2-5 times less memory. On modern hardware, that means that documents in the 50-100 megabyte range can be manipulated in memory, and that documents in the 0-1 megabyte range load in zero time (0.0 seconds). This allows you to drastically simplify many kinds of XML applications.

CHAPTER 5: IMPLEMENTATION

As the Internet continues to grow in terms of scale, complexity, openness and accessibility, Network Security assumes a pivotal role in the Society of the future. With the proliferation of devices connected to the Internet, and the advent of the ‘Internet of Things’, the use cases for Internet connected devices continues to grow and increasingly complex mechanisms are being developed to efficiently use this technology to improve existing services. One good example of this is the Advanced Metering Infrastructure (AMI), which uses Internet connected sensors and predictive analytics to optimize power generation and distribution, reducing costs for consumers and producers alike. It does so by enabling the Power industry to function using Just-In-Time production principles that drastically reduce the cost of generation of power. It also lets consumers monitor their usage patterns and has quite a few interesting use cases in the domain of smart homes. With any opportunity, comes associated threats and the AMI has its fair share when it comes to securing the infrastructure against network based attacks that may even result in financial losses and power theft.

5.1 Advanced Metering Infrastructure

Advanced Metering Infrastructure (AMI) are systems that measure, collect, and analyze energy usage, and communicate with metering devices such as electricity meters, gas meters, heat meters, and water meters, either on request or on a schedule. These systems include hardware, software, communications, consumer energy displays and controllers, customer associated systems, Meter Data Management software, and supplier business systems.

Government agencies and utilities are turning toward advanced metering infrastructure (AMI) systems as part of larger “Smart Grid” initiatives. AMI extends current advanced meter reading (AMR) technology by providing two way meter communications, allowing commands to be sent toward the home for multiple purposes, including “time-of-use” pricing information, demand-response actions, or remote service disconnects. Wireless technologies are critical elements of the “Neighbourhood Area Network” (NAN), aggregating a mesh configuration of up to thousands of meters for back haul to the utility’s IT headquarters.

The network between the measurement devices and business systems allows collection and distribution of information to customers, suppliers, utility companies, and service providers. This enables these businesses to participate in demand response services.

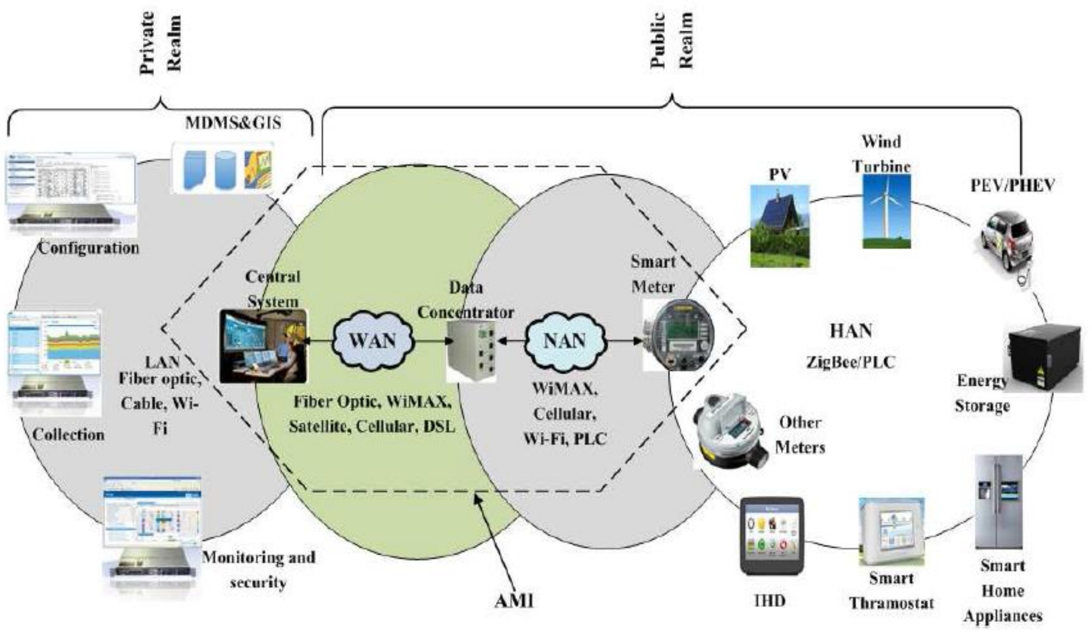
Consumers can use information provided by the system to change their normal consumption patterns to take advantage of lower prices. Pricing can be used to curb growth of peak consumption. AMI differs from traditional automatic meter reading (AMR) in that it enables two-way communications with the meter.

AMI includes several communication networks, identified according to their spatial scope:

• The Wide Area Network (WAN) serves as a communication link between head-ends in the local utility network and either data concentrators or smart meters. This network uses long-range and high-bandwidth communication technologies, such as WiMAX, cellular (3G, EVDO, EDGE, GPRS, or CDMA), satellite, Power Line Communication (PLC), and Metro Ethernet. The scale of this network could reach several million nodes.

• Neighborhood Area Networks (NANs) ensure communication between data concentrators or access points and smart meters that play the role of interfaces with a Home Area Network (HAN). The scale of this network ranges from a few hundred to tens of thousands of nodes.

• Finally, Field Area Networks (FANs) allow the utility workforce to connect to equipment in the field.



AMI overview

Figure 5.1 Architecture of AMI

With the rapid adoption AMI and its associated technology all over the world, there is an increasing concern with respect to the security of such a network. These concerns are fuelled by the size and complexity of the network, as well as the increasingly sophisticated exploits prevalent in today’s day and age. Some attack vectors that are unique to AMI include power theft and manipulation of the physical conditions around the sensors in the Home Area Networks that may result in damage to life and property. A real world example of this would be a situation where in the attacker momentarily manipulates sensor information that is fed into the network at its outer edges, leading to a chain of events that might hamper production at a manufacturing facility, or result in the loss of life and property.

Our solution to the problem is a Distributed Statistical Anomaly based Intrusion Detection System for Advanced Metering Infrastructure (AMI) that works on the foundational principles of Machine Learning. It is a cluster-based system that is designed to be placed at AMI head-ends, which essentially serve as aggregation points for sub-networks.

The primary motivation towards advocating a distributed approach is the overall traffic of the Internet is bound to increase which translates into a direct increase in the amount of traffic that the individual sub-networks of the Internet are expected to handle. This also results in an increase in the amount of traffic that is to be analyzed by the Intrusion Detection System and if not designed well, can lead to a Bottleneck. A cluster based solution offers the best return on investment, provides scalability and fault-tolerance in addition to providing scope for parallelization of Computations that result in a direct increase in system throughput, responsiveness and performance. It also provides the benefit of using commodity hardware as opposed to special purpose hardware.

The security solutions that can be built for the Advanced Metering Infrastructure built usually fall into two categories:

1. Detection: This class of solutions focuses on the detection of potential threats to the system, merely reporting the current state of the system to the person in charge of the network, usually the System/ Network Administrator.
2. Prevention: This class of solutions encompasses both the detection of potential threats and real-time preventive action taken in accordance to these threats.

Our system falls under the first category of solutions, it merely serves to detect anomalous activity in the network that is potentially malicious in nature and report such activity to the system/ network administrator, and it takes no preventive action.

However, our system is designed in a modular fashion keeping in mind future requirements and can be easily extended to take preventive action, including things like changing firewall configurations on a real time basis to blacklist connections belonging to certain IP addresses that are thought to belong to the attacking entities and may be potentially malicious in nature.

5.2 Supervised Learning

The availability of a world-class labelled intrusion detection dataset (ISCX ’12) steered us in the direction of using supervised machine learning as opposed to unsupervised approaches.

Supervised learning is the machine learning task of inferring a function from labelled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way (see inductive bias).The parallel task in human and animal psychology is often referred to as concept learning.

The system uses supervised learning techniques to infer a model of regular network traffic from the training data, proceeding to use this model to classify network packets on a real time basis.

Supervised learning techniques tend to partition the dataset into two halves; the ‘training set’ and the ‘testing set’, which may or may not be of the same size. The training set is used to ‘train’ the model, essentially it is the input to the classifier for the learning process, the classifier makes a best effort attempt to learn the model/pattern underlying the training dataset. The test dataset is used to validate the performance of the model created during the learning phase, as the model is forced to make predictions on unseen data, which simulates a real world situation where in the model has to make accurate predictions on data it has never seen before.

The challenge is to tune the parameters of the model in such a manner that the model is able to correctly classify unknown instances, a model that best fits the data without over-fitting or under-fitting the model to the data.

Under-fitting refers to creating a highly simplified model that misclassifies a large part of the training data set in addition to the test dataset; this naturally results in poor accuracy.

Over-fitting occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. A model that has been over-fit has poor predictive performance, as it overreacts to minor fluctuations in the training data. Over-fitting occurs when a model begins to "memorize" training data rather than "learning" to generalize from trend. As an extreme example, if the number of parameters is the same as or greater than the number of observations, a simple model or learning process can perfectly predict the training data simply by memorizing the training data in its entirety, but such a model will typically fail drastically when making predictions about new or unseen data, since the simple model has not learned to generalize at all.

There challenge is to find the right balance between the two approaches resulting in an optimal system. The underlying philosophy is to choose the simplest hypothesis that best fits the data and is often known as Occam’s razor.

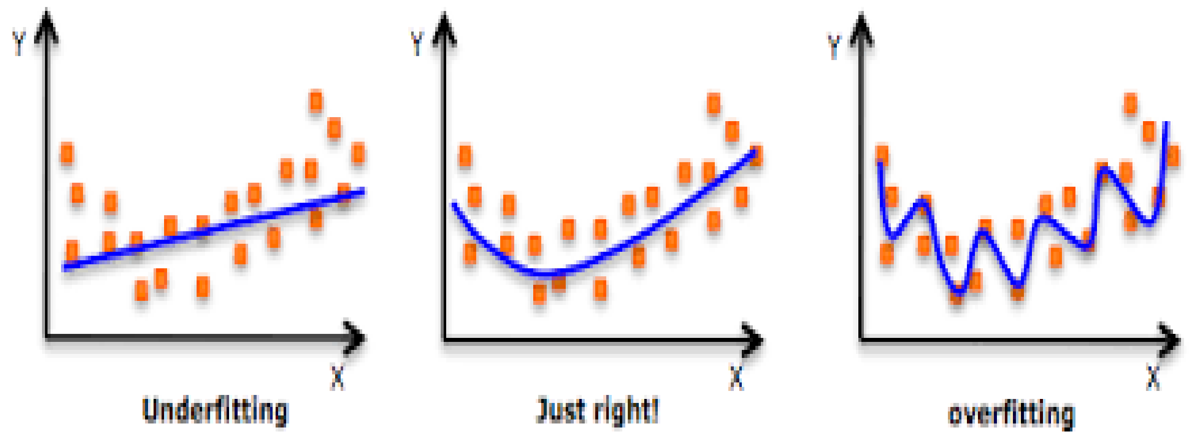


Figure 5.2 Representation of different fits in Classifiers

5.3 Threat Model

Our implementation detects anomalous behaviour limited to the types of attacks it has seen, i.e., it has been trained upon. This is a major limitation of using Supervised Learning techniques as the system cannot generalise to all types of attacks, there are some exploits it is unable to detect. The ISCX ’12 dataset has instances of attacks belonging to four classes of attacks:

1. Denial Of Service (DoS)

   In computing, a denial-of-service (DoS) attack is an attempt to make a machine or network resource unavailable to its intended users, such as to temporarily or indefinitely interrupt or suspend services of a host connected to the Internet.

2. Distributed Denial of Service(DDoS)

   A distributed denial-of-service (DDoS) attack occurs when multiple systems flood the bandwidth or resources of a targeted system, usually one or more web servers. Such an attack is often the result of multiple compromised systems (for example a botnet) flooding the targeted system with traffic.

3. Brute Force SSH

   Brute force (also known as brute force cracking) is a trial and error method used by application programs to decode encrypted data such as passwords or Data Encryption Standard (DES) keys, through exhaustive effort (using brute force) rather than employing intellectual strategies. Brute force SSH refers to the process of using such an attack to break into systems running an SSH server.

4. Attacks from within the network (Port Scanning)

   Port scanning is the act of systematically scanning a computer's ports. Since a port is a place where information goes into and out of a computer, port scanning identifies open doors to a computer. Port scanning has legitimate uses in managing networks, but port scanning also can be malicious in nature if someone is looking for a weakened access point to break into your computer.

5.4 Datasets Selection

Despite the significant contributions of DARPA and KDD datasets in the intrusion detection domain, their accuracy and their ability to reflect real world conditions have been criticized in McHugh (2009) and Brown et al. (2009). Moreover, since our proof of concept deals with a new and evolving form of a network it became a prerequisite for the dataset to be recent, have full payload information & be publically available. These features and the ones listed below were key in leading us to the ISCX dataset:

1. Realistic network traffic
2. Labeled dataset
3. Total interaction capture
4. Complete capture
5. Diverse intrusion scenarios

For the purpose of our research we elected the ISCX dataset. Many datasets which are currently used by IDS researchers particularly lack in reflecting current network trends, some are heavily anonymized and others are not publically available because of privacy concerns. Moreover, we required a dynamically generated dataset for the purposes of our evaluation.

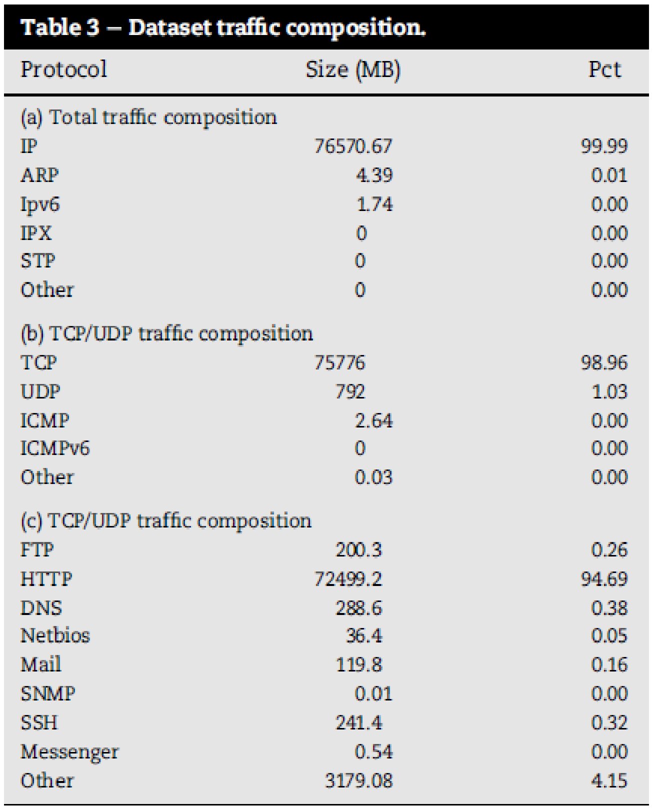


Table 3 – Dataset traffic composition.

Protocol	Size (MB)	Pct
(a) Total traffic composition		
IP	76570.67	99.99
ARP	4.39	0.01
Ipv6	1.74	0.00
IPX	0	0.00
STP	0	0.00
Other	0	0.00
(b) TCP/UDP traffic composition		
TCP	75776	98.96
UDP	792	1.03
ICMP	2.64	0.00
ICMPv6	0	0.00
Other	0.03	0.00
(c) TCP/UDP traffic composition		
FTP	200.3	0.26
HTTP	72499.2	94.69
DNS	288.6	0.38
Netbios	36.4	0.05
Mail	119.8	0.16
SNMP	0.01	0.00
SSH	241.4	0.32
Messenger	0.54	0.00
Other	3179.08	4.15

Figure 5.3 ISCX Dataset traffic composition

Although many real world network traces do exist for the IDS research community, many, if not all of the features mentioned above. As an example, the DARPA datasets were constructed for network security purposes, to simulate the traffic seen in a medium sized US Air Force base. Upon careful examination of DARPA’98 and ’99 in McHugh (2000) and Brown et al. (2009), many issues have been raised including their failing to resemble real traffic, and numerous irregularities due to the synthetic approach to data generation and insertion into the dataset. The KDD’99 dataset is also built based on the data captured in DARPA’98 which nonetheless suffers from the same issues.

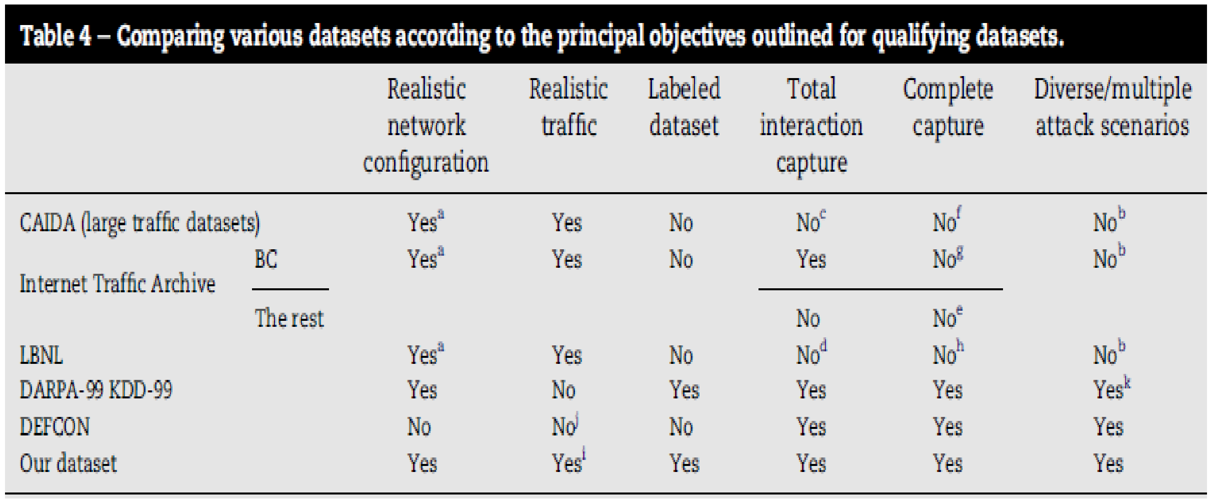


Table 4 – Comparing various datasets according to the principal objectives outlined for qualifying datasets.

		Realistic network configuration	Realistic traffic	Labeled dataset	Total interaction capture	Complete capture	Diverse/multiple attack scenarios
CAIDA (large traffic datasets)		Yes[a]	Yes	No	No[c]	No[f]	No[b]
Internet Traffic Archive	BC	Yes[a]	Yes	No	Yes	No[g]	No[b]
	The rest				No	No[e]	
LBNL		Yes[a]	Yes	No	No[d]	No[h]	No[b]
DARPA-99 KDD-99		Yes	No	Yes	Yes	Yes	Yes[k]
DEFCON		No	No[j]	No	Yes	Yes	Yes
Our dataset		Yes	Yes[i]	Yes	Yes	Yes	Yes

Figure 5.4 Comparison of various intrusion datasets

5.5 Feature Selection

Feature Selection or attribute selection is a process by which you automatically search for the best subset of attributes in your dataset. The notion of “best” is relative to the problem at hand, but typically means highest accuracy.

The ISCX dataset is a fully labelled dataset that contains network capture for a span of 7 days. The network capture is available in the form of full packet capture (PCAP) files which contain four different attack scenarios along with normal traffic. Out of the 19 features available in the labelled dataset we selected 8 features that are important for our research work. To select the features we used the State Space Search feature extraction method built in the WEKA Data Mining Software. In addition to the 8 features we added a new feature called duration obtained from combining startDateTime and stopDateTime.

State space search is a process in which successive configurations or states of an instance are considered, with the goal of finding a goal state with a desired property. Problems are often modelled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation that can be performed to transform the first state into the second.

Three key benefits of performing feature selection on datasets are:

Reduces Overfitting: Less redundant data means less opportunity to make decisions based on noise.

Improves Accuracy: Less misleading data means modeling accuracy improves.

Reduces Training Time: Less data means that algorithms train faster.

5.6 Classifier Selection

Selecting a suitable classifier is one of the most important aspect of machine learning. The mandate to achieve good results in terms of accuracy and performance led us to test out several different ones, in which we made sure to try different parameters within each classifier as well.
Often the hardest part of solving a machine learning problem, choosing the right estimator makes use of the following information for the given problem at hand:

1. Data sets size
2. Labelled or Unlabelled dataset
3. Type of prediction or target classes
4. Type of data (text, multimedia etc)

From the point of view of our implementation we considered the following classifiers:

Support Vector Machines (SVM)

High accuracy, nice theoretical guarantees regarding overfitting, and with an appropriate kernel they can work well even if the data isn't linearly separable in the base feature space. Especially popular in text classification problems where very high-dimensional spaces are the norm. SVMs are known for being memory-intensive, hard to interpret, and fairly complex to run and tune.
When we trained our model using Linear SVC, there were minimalistic changes over the results obtained when using the ensemble decision trees.

Decision Trees

By far the biggest advantage of using decision trees is that they are easy to interpret and explain. Decision trees handle feature interactions and they are non-parametric, For example, decision trees

easily take care of cases where you have class A at the low end of some feature x, class B in the mid-range of feature x, and A again at the high end.
A natural problem that tends to associate with decision trees is overfitting. We avoided overfitting by using an ensemble method of trees for classification. Using this method we can make the system as a whole, scalable and fast.

5.7 Preprocessing

Since the essence of the solution is using Machine Learning to classify packets, it is essential to build a mathematically abstract model of the incoming data suitable for classification purposes.
We do this by extracting features relevant to the classification task from each incoming packet. The whole process results in the conversion of incoming packets in Pcap format into a Comma separated values file, with each row consisting of a list of numerical values corresponding to the relevant features selected during the feature selection process. The pre-processing process undertaken during the training phase of the model is very similar to the one taken during real-time packet analysis with the only difference being the presence of a Packet Sniffer as an additional component, and a script to convert the output of the packet sniffer (Pcap file) into a CSV file as outlined above. The workflow of the pre-processing stage which generates the CSV file(s) is as follows:

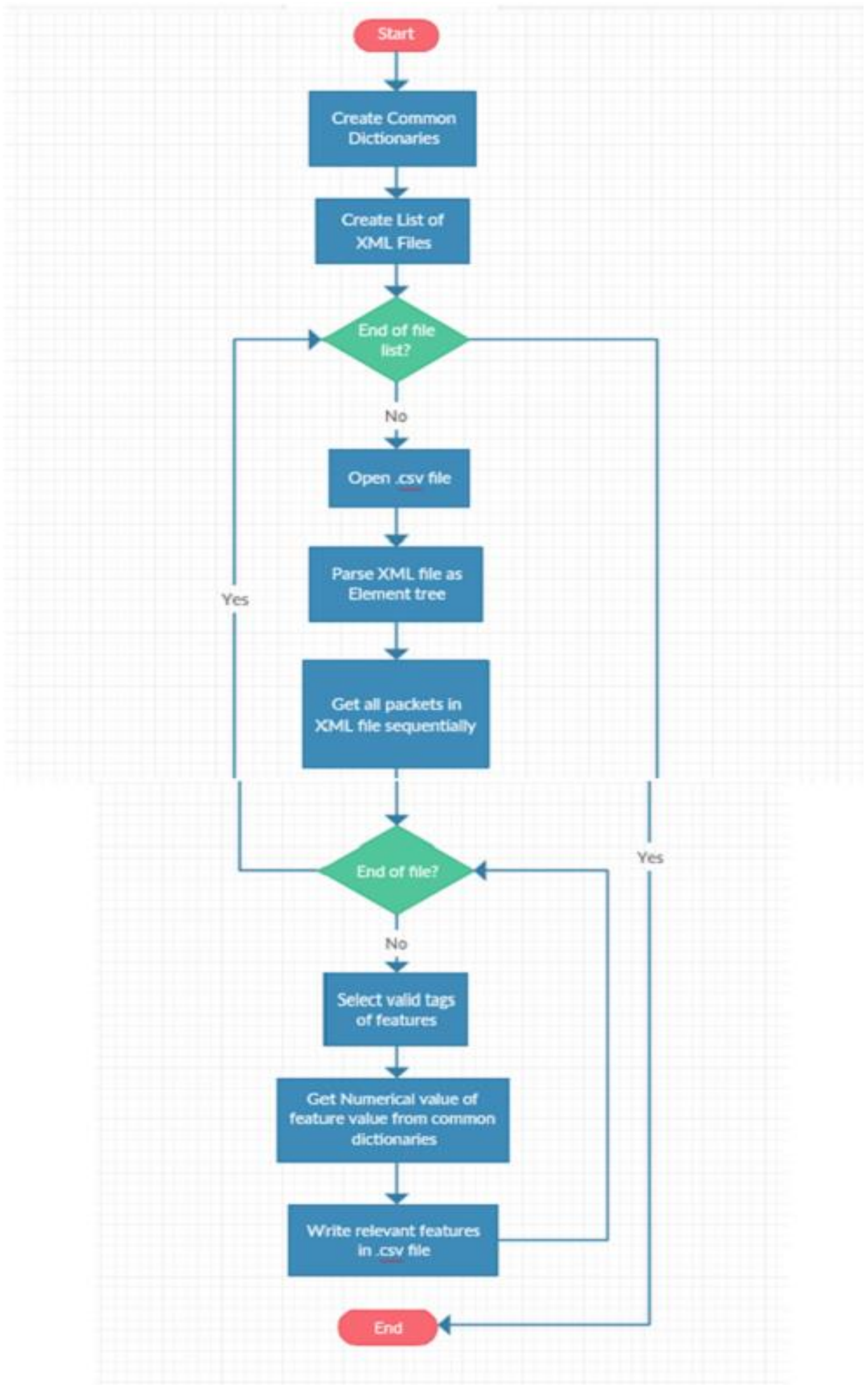

Start
Create Common Dictionaries
Create List of XML Files
End of file list?
No
Yes
Open .csv file
Parse XML file as Element tree
Get all packets in XML file sequentially
End of file?
No
Yes
Select valid tags of features
Get Numerical value of feature value from common dictionaries
Write relevant features in .csv file
End

Figure 5.5 Flowchart of Preprocessing of Packets

Preprocessing - Creation of CSV files:

- Create a list of all the XML files to be parsed. These xml files contain tags which correspond to the features of the packet. Each xml file contains a day's worth of traffic whereas on some days, since the traffic was of a larger size, it was contained in three xml files. There were 12 files in total.
- Create a set of common dictionaries corresponding to features which are not numeric. These dictionaries contain the string value of the features and its corresponding numeric value. This is done since the CSV files need to be numeric. The dictionaries are common for all the xml files.
- The features which are useful to us are: appName, totalDestinationPackets, totalSourcePackets, direction, source, protocolName, destination, Tag, duration (difference of startDateTime and stopDateTime).
- Since the XML files were too large to be entirely stored in the memory, they were reduced in size to contain just the relevant features.
- When the value of the feature is extracted from between the tags, it's fed into a function which tries to find it in the corresponding dictionary. If the value is found, the numerical equivalent of the value is returned. Otherwise, the value is added to the dictionary as a new element and its equivalent numerical value is generated.
- For each packet, a list of numerical values is created which are the values of the features and end of the packet's scope, these values are written as a line in the csv file.
- This process goes on till twelve csv files are generated one for each xml file.

Preprocessing - Creating Test and Train files for each attack:

- There are 4 kinds of attacks in the dataset - DOS, DDOS, Brute force, SSH Infiltration.
- A test and training set was required and each file was supposed to contain attacks and normal packets in the ratio of 30:70.
- These packets were randomly selected from the generated csv files.

Source Code CSV file generator:

```
import xml.etree.ElementTree as ET
import csv
import re
import os
from datetime import datetime as dt


def modDictionary(dic, newKey):
    if not dic.has_key(newKey):
        if len(dic.keys()) == 0:
            val = 1
        else:
            tmp = sorted(dic.values())
            key = tmp[-1]
            val = key+1
        dic[newKey] = val
    return


def addToList(objList, itemName):
    if itemName not in objList:
        objList.append(itemName)
    return

dicappName = {}
dictotalDestinationPackets = {}
dictotalSourcePackets = {}
dicdirection = {}
dicIP = {}
dicprotocolName = {}
dicTag = {}

path = "C:\\Users\\student\\labeled_flows_xml\\csvs2\\"
path2 = "C:\\Users\\student\\labeled_flows_xml\\csvstmp\\"
files = os.listdir(path)
featureList = ["appNme", "totalDestinationPackets",
"totalSourcePackets", "direction", "source", "protocolName",
               "destination", "Tag", "startDateTime",
"stopDateTime"]
for file in files:

    myfile = open(path2 + file.split(".")[0] + ".csv", 'wb')
    writer = csv.writer(myfile)
    context = ET.parse(path+file)
    root2 = context.getroot()
    root = iter(root2)
    packets = []
    startTime = ""
    stopTime = ""
    packetID = 0
    appNameList = []
    for object in root:
        it = iter(object)
        packetID += 1
        packets.append(packetID)
        for item in it:
            if item.tag in featureList:
                tag = item.tag
                val = item.text
                if tag == "appName":
                    if val not in dicappName:
                        modDictionary(dicappName, val)
                    addToList(appNameList, val)
                    packets.append(dicappName[val])
                elif tag == "totalDestinationPackets":
                    packets.append(val)
                elif tag == "totalSourcePackets":
                    packets.append(val)
                elif tag == "direction":
                    if val not in dicdirection:
                        modDictionary(dicdirection, val)
                    packets.append(dicdirection[val])
                elif tag == "source":
                    if val not in dicIP:
                        modDictionary(dicIP, val)
                    packets.append(dicIP[val])
                elif tag == "protocolName":
                    if val not in dicprotocolName:
                        modDictionary(dicprotocolName, val)
                    packets.append(dicprotocolName[val])
                elif tag == "destination":
                    if val not in dicIP:
                        modDictionary(dicIP, val)
                    packets.append(dicIP[val])
                elif tag == "Tag":
                    if val not in dicTag:
                        modDictionary(dicTag, val)
                    packets.append(dicTag[val])
                elif tag == "startDateTime":
                    startTime = dt.strptime(val, '%Y-%m-
%dT%H:%M:%S')
                elif tag == "stopDateTime":
                    stopTime = dt.strptime(val, '%Y-%m-
%dT%H:%M:%S')
                    duration = (stopTime - startTime).seconds
                    packets.append(duration)
                else:
                    continue

        packets = map(int, packets)
        writer.writerow(packets)
        packets = []
        startTime = ""
        stopTime = ""
    root2.clear()
    myfile.close()
```

Packet Sniffer to CSV tool:

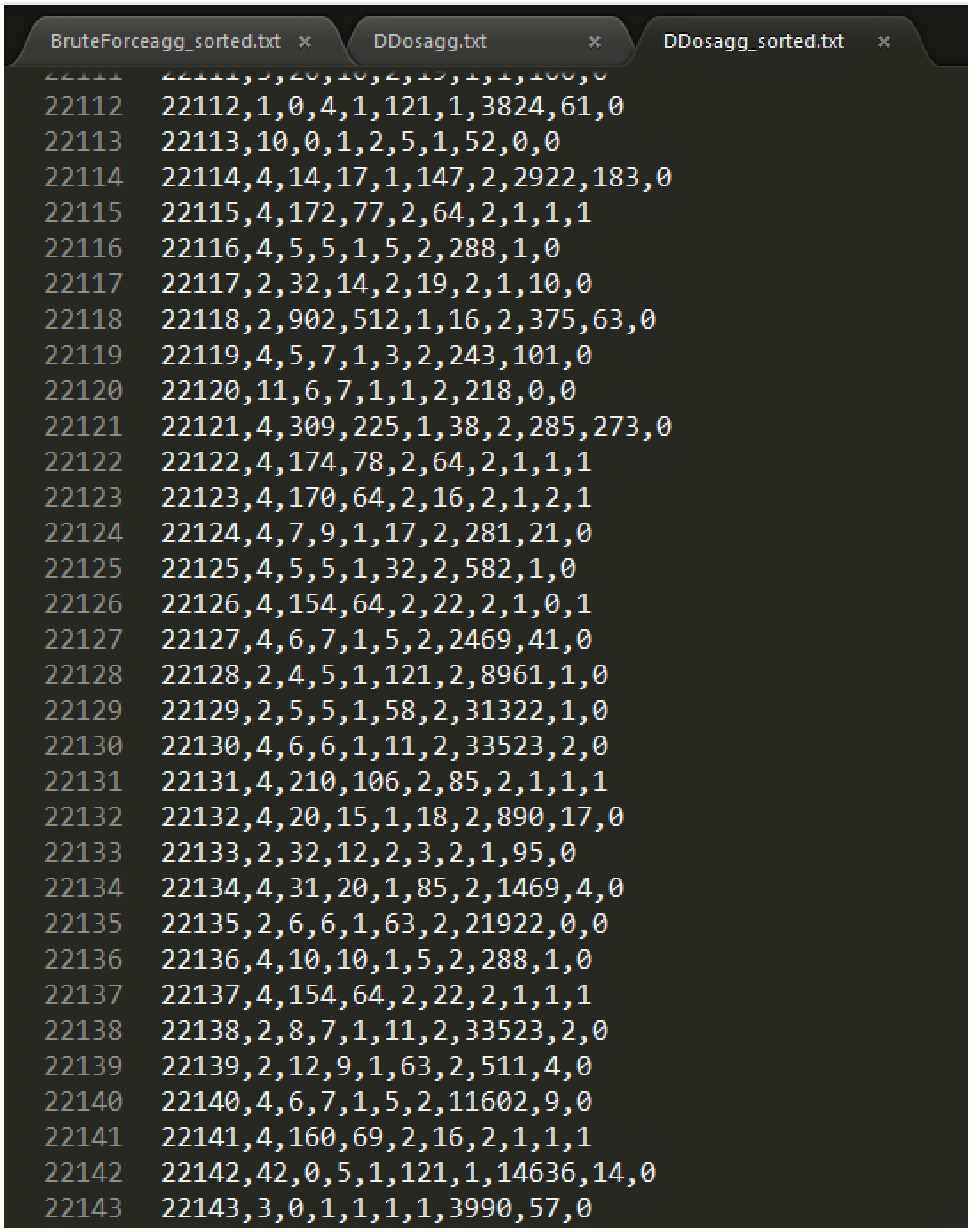
- The packet sniffer to csv is a tool which generates a Test csv file of packets which were captured in real time.
- A pcap file can be generated using any packet sniffer like NMap, Wireshark. This pcap file contains the packet information of the packets captured.
- This information is not labeled to the extent that it also captures failed transactions.
- The script parses this pcap file and with the help of regular expressions, extracts the relevant features useful for the IDS.
- The algorithm refers the common dictionary which was used initially to generate the main csv files.
- The output is a csv file containing the comma separated numerical values of the features of the packets captured in real time.

Source Code:

```python
import csv
import re
from datetime import datetime as dt
i = 0
path = "C:\\Users\\student\\Project\\sniffer\\"
file = "mypcap1.pcap"
csvfile = "C:\\Users\\student\\Project\\sniffer\\captureCSV.csv"
fcsv = open(csvfile, "ab")
fw = csv.writer(fcsv)
UID = 0
with open(path+file, "r") as fptr:
    linetmp = ""
    lines = []
    for line in fptr:
        tokens = []
        line = line.strip()
        linetmp += " " + line
        i += 1
        if i == 2:
            lines.append(linetmp.strip())
            i = 0
            linetmp = ""
    transactionStart = 0
    transaction = []
    tmptrans = []
    for item in lines:
        if "(correct)" in item and transactionStart != 1:
            transactionStart = 1
            tmptrans.append(item)
            continue
        if "(correct)" in item and transactionStart == 1:
            tmptrans.append(item)
            transaction.append(tmptrans)
            tmptrans = []
            transactionStart = 0
    toCSV = []
    for trans in transaction:
        startTime = ""
        timeFlag = 0
        stopTime = ""
        source = ""
        destination = ""
        for item in trans:
            tmpIP = []
            IP = []
            UID += 1
            toCSV.append(UID)
            toCSV.append("SSH")
            matchTime = re.search(r"^\d+:\d+:\d+\.\d+\s", item)
            toCSV.append(matchTime.group().strip())
            if timeFlag == 0:
                startTime = matchTime.group().strip()
                startTime = dt.strptime(startTime,
"%H:%M:%S.%f")
                timeFlag = 1
            else:
                stopTime = matchTime.group().strip()
                stopTime = dt.strptime(stopTime, "%H:%M:%S.%f")
                duration = (stopTime - startTime).seconds
                timeFlag = 0
            matchIP =
re.search(r"\s\d+\.\d+\.\d+\.\d+\.\d+\s>\s\d+\.\d+\.\d+\.\d+\.\d
+", item)
            tmpIP = matchIP.group().split(">")
            for i in tmpIP:
                IP.append(i.strip())
            source = IP[0]
            toCSV.append(source)
            destination = IP[1]
            toCSV.append(destination)
            matchSizeString = re.search(r"length\s\d+\)", item)
            matchSize = re.search(r"\d+",
matchSizeString.group())
            matchProtoString = re.search(r"\sproto\s[A-Za-z0-
9]+\s", item)
            protoList = matchProtoString.group().split()
            protocol = protoList[1].strip()
            toCSV.append(protocol)
            toCSV.append("L2L")
            fw.writerow(toCSV)
            toCSV = []
fcsv.close()
```

The output of the pre-processing are CSV file(s), the lines of which correspond to a set of comma separated values representing the values of the features relevant to the classification task.

The final output is a file whose format resembles the one below:



```
22112,1,0,4,1,121,1,3824,61,0
22113,10,0,1,2,5,1,52,0,0
22114,4,14,17,1,147,2,2922,183,0
22115,4,172,77,2,64,2,1,1,1
22116,4,5,5,1,5,2,288,1,0
22117,2,32,14,2,19,2,1,10,0
22118,2,902,512,1,16,2,375,63,0
22119,4,5,7,1,3,2,243,101,0
22120,11,6,7,1,1,2,218,0,0
22121,4,309,225,1,38,2,285,273,0
22122,4,174,78,2,64,2,1,1,1
22123,4,170,64,2,16,2,1,2,1
22124,4,7,9,1,17,2,281,21,0
22125,4,5,5,1,32,2,582,1,0
22126,4,154,64,2,22,2,1,0,1
22127,4,6,7,1,5,2,2469,41,0
22128,2,4,5,1,121,2,8961,1,0
22129,2,5,5,1,58,2,31322,1,0
22130,4,6,6,1,11,2,33523,2,0
22131,4,210,106,2,85,2,1,1,1
22132,4,20,15,1,18,2,890,17,0
22133,2,32,12,2,3,2,1,95,0
22134,4,31,20,1,85,2,1469,4,0
22135,2,6,6,1,63,2,21922,0,0
22136,4,10,10,1,5,2,288,1,0
22137,4,154,64,2,22,2,1,1,1
22138,2,8,7,1,11,2,33523,2,0
22139,2,12,9,1,63,2,511,4,0
22140,4,6,7,1,5,2,11602,9,0
22141,4,160,69,2,16,2,1,1,1
22142,42,0,5,1,121,1,14636,14,0
22143,3,0,1,1,1,1,3990,57,0
```

The format of each row of the CSV file is

(1),(2),(3),(4),(5),(6),(7),(8),(9),(10)

(1) - Unique Identifier

(2) - Application name (An integer corresponding to the application name)

(3) – Total Source Packets

(4) – Total Destination Packets

(5) – Direction (An integer corresponding to one of L2L, L2R, R2R)

(6) – Source IP Address (An integer representation of the Source IP Address)

(7) – Protocol (An integer corresponding to the Protocol)

(8) – Destination IP Address (An integer representation of the Destination IP Address)

(9) – Duration (Numerical difference between the end-time and start-time)

(10) – Tag (An integer corresponding to Attack or Normal (0/1))

5.8 Training the model

The use of machine learning entails 'learning' with experience. In Supervised learning, this translates to building a model of the classification function from the data and their associated labels. We have advocated the use of four separate classifiers for identifying each type of attack to take advantage of the parallel processing techniques inherent in the use of a distributed system, and storm in particular. The primary task of this stage was building four instances of a Decision Tree Classifier, with each classifier responsible for classifying packets relevant to a class of attacks (DoS, DDoS, Brute Force SSH, Port Scanning). Each decision tree classifier was trained using a pre-defined ratio of normal packet data and attacking instances, the attacking instances all corresponding to the same attack type. For instance, to train the classifier responsible for the Brute Force SSH attack, the input data comprises of a random mix of normal packet data taken from across the seven days of packet capture and attack packet data corresponding to the Brute Force attack in a predefined ratio. The process of converting the raw dataset into processed transformed CSV files suitable for building a mathematical model of the data is as outlined in the pre-processing section. The objects corresponding to mathematical models of classifiers created during the training phase are then serialized to disk for persistence using the pickle module. This persisted module may then be loaded into memory at any stage, we need not retrain the model each time we have to classify packets.

5.9 Software Architecture

The architecture of our system is distributed in nature. It consists of a number of slave nodes (one per attack type), a master node and a Zookeeper node. This architecture is mandated by our use of

Apache Storm as a distributed processing framework. Apache Storm is an open source real-time distributed computing framework that greatly simplifies the task of developing distributed software systems that have to process large swarms of data in real time. It abstracts away the complexities of talking to different machines on a network allowing developers to focus on application functionality and leave most of the complexities of message passing, ensuring fault tolerance and distribution up to the framework. Storm is simple to use and provides support for many languages.

Our choice of Apache Storm as a distributing processing framework was motivated by its emphasis on real-time application support. Most other distributed processing frameworks like Apache Hadoop and Apache Spark do a terrific job in supporting applications that require batch processing, but provide no support for stream-based applications that need to work on data in real-time. Storm was created to fill that very void and provided a perfect fit to our requirements. The only cost incurred by limiting ourselves to the Storm framework was that we had to think of our solution in terms of Storm abstractions, which wasn't much of a limitation in the sense that there was a direct correlation between the way we'd envisioned the system and the way Storm is structured.

Storm provides a couple of core abstractions to its users:

- Stream: The core abstraction in Storm is the "stream". A stream is an unbounded sequence of tuples. Storm provides the primitives for transforming a stream into a new stream in a distributed and reliable way. For example, you may transform a stream of tweets into a stream of trending topics.
- Spout: A spout is a Storm abstraction that serves as the source of Storm streams. For example, a spout may read tuples off of a Kestrel queue and emit them as a stream or a spout may connect to the Twitter API and emit a stream of tweets.
- Bolt: A bolt consumes any number of input streams, does some processing, and possibly emits new streams. Complex stream transformations, like computing a stream of trending topics from a stream of tweets, require multiple steps and thus multiple bolts. Bolts can do anything from run functions, filter tuples, do streaming aggregations, do streaming joins, talk to databases, and more.

- Topology: Networks of spouts and bolts are packaged into a "topology" which is the top-level abstraction that you submit to Storm clusters for execution. A topology is a graph of stream transformations where each node is a spout or bolt. Edges in the graph indicate which bolts are subscribing to which streams. When a spout or bolt emits a tuple to a stream, it sends the tuple to every bolt that subscribed to that stream.

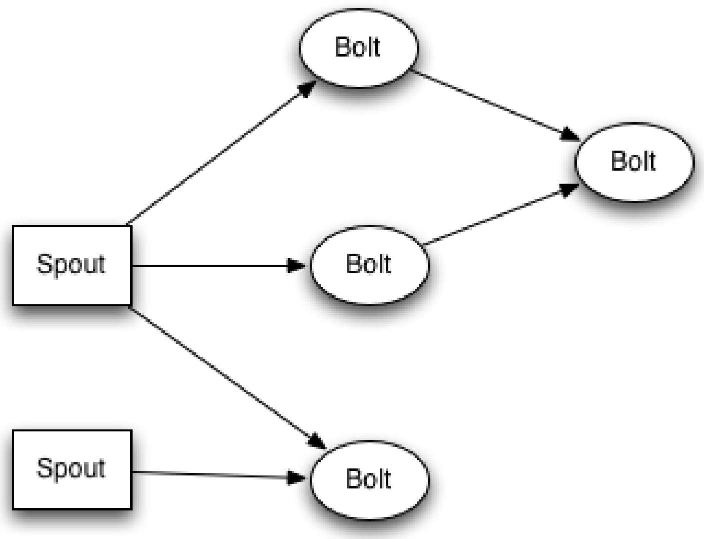


Figure 5.6 Example Storm Topology

Links between nodes in your topology indicate how tuples should be passed around. For example, if there is a link between Spout A and Bolt B, a link from Spout A to Bolt C, and a link from Bolt B to Bolt C, then every time Spout A emits a tuple, it will send the tuple to both Bolt B and Bolt C. All of Bolt B's output tuples will go to Bolt C as well.

Each node in a Storm topology executes in parallel. In your topology, you can specify how much parallelism you want for each node, and then Storm will spawn that number of threads across the cluster to do the execution.

A topology runs forever, or until you kill it. Storm will automatically reassign any failed tasks. Additionally, Storm guarantees that there will be no data loss, even if machines go down and messages are dropped.

The basic workflow of our system in terms of storm components is as outlined below.

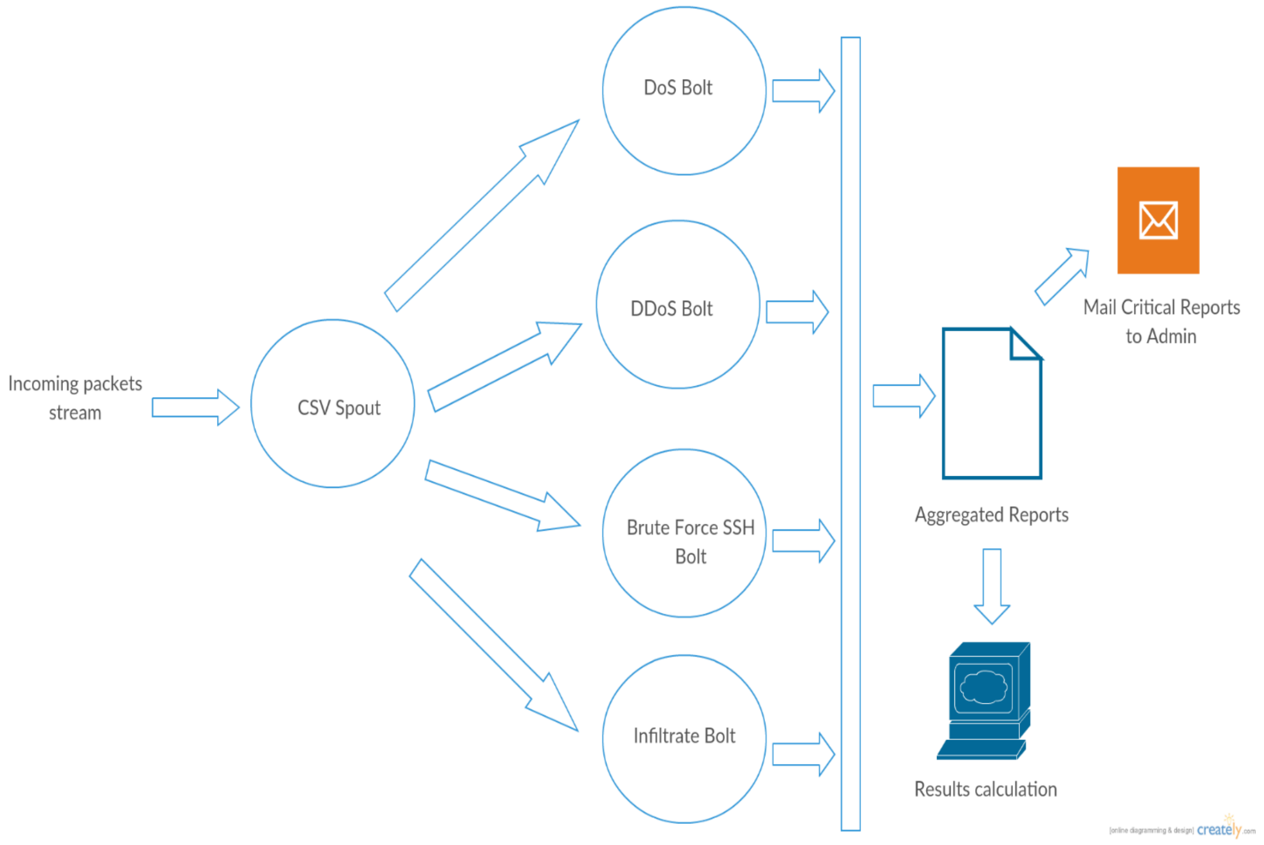
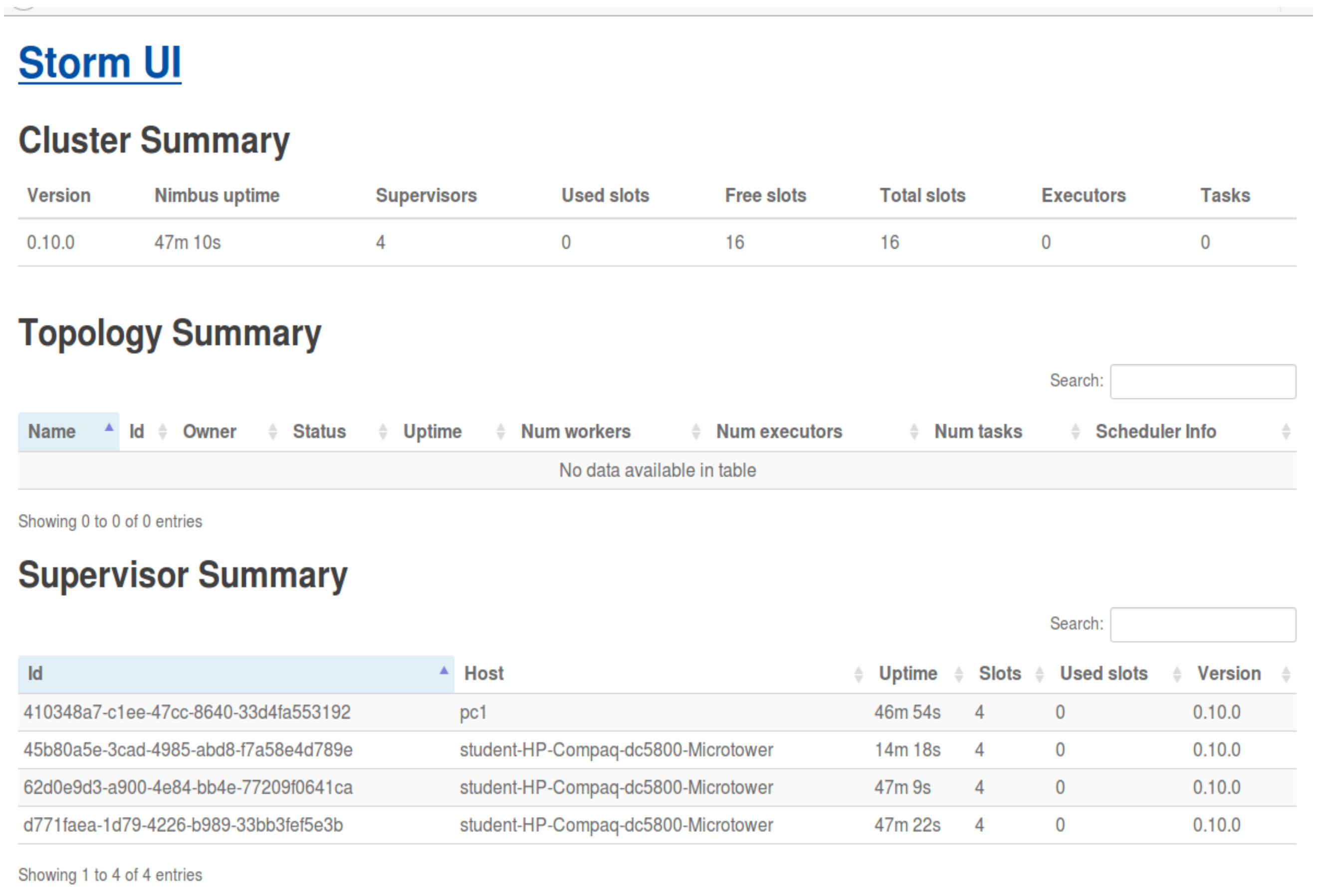


Figure 5.7 Workflow of spouts and bolts

The incoming packet stream corresponds to an abstraction of the incoming packets, manifesting itself in code in the form of a Java stream created using the CSV file outputted during the pre-processing stage. In order to simulate a real-time system that has incoming data round the clock, throughout the year, we read the data stream in a cyclical repetitive manner (Once we reach the end of the CSV file, we start again from the top. This gives the illusion of a continuous non-ending stream of data). The CSV Spout functions as a buffer converting the incoming Java data stream into tuples that can be understood by the Storm subsystem. It is written entirely in Java, the code for which can be found in the Appendix. The tuples are then sent ahead to the four bolts, with each tuple replicated 4 times, one for each Bolt. Each bolt corresponds to a processing entity that performs the classification task in addition to the tasks of loading the persisted classifier into main memory and converting each tuple into a data format compatible with the interface of the classifier.

Every Bolt is a logical abstraction representing one of the four classifiers built during training. Each Bolt serves to identify attack instances of a particular input class, with the classes being restricted to the four types of attacks in the dataset. For the time being, since our system was limited in scope to being a proof of concept, we had each bolt write out its results to a text file in the form of a list of CSV lists, each CSV list corresponding to a list of features of each packet, with the last value corresponding to the target class label predicted by the classifier. These files are then further processed to produce aggregated reports, results and an E-mail notification notifying the system administrator of a possible breach. In the current system, each of these steps is processed by separate scripts after the system is offline. In a production system, these activities can be encapsulated into Bolts themselves so that the system can produce reports in real-time.

Outlined below, are some screenshots of our system in action



Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.10.0	47m 10s	4	0	16	16	0	0

Topology Summary

Search:

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Scheduler Info
No data available in table								

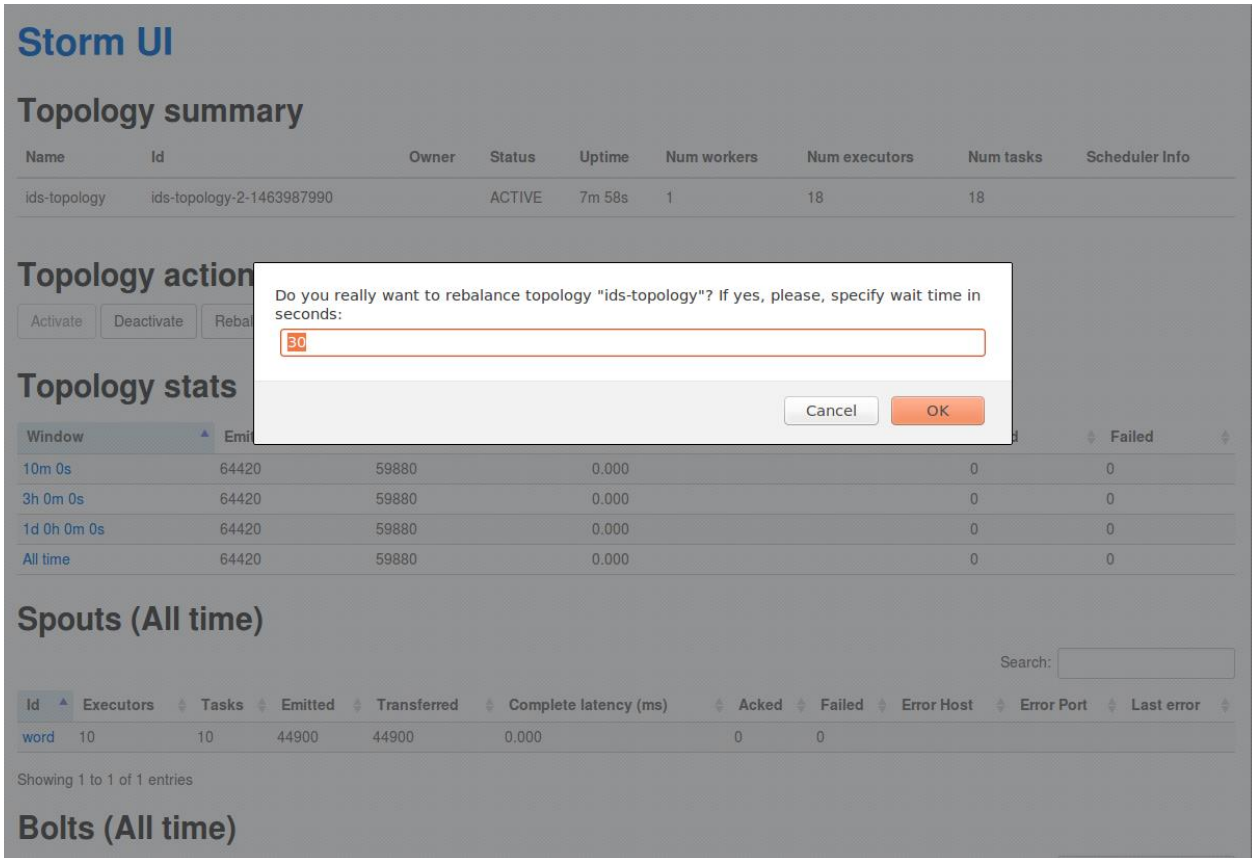
Showing 0 to 0 of 0 entries

Supervisor Summary

Search:

Id	Host	Uptime	Slots	Used slots	Version
410348a7-c1ee-47cc-8640-33d4fa553192	pc1	46m 54s	4	0	0.10.0
45b80a5e-3cad-4985-abd8-f7a58e4d789e	student-HP-Compaq-dc5800-Microtower	14m 18s	4	0	0.10.0
62d0e9d3-a900-4e84-bb4e-77209f0641ca	student-HP-Compaq-dc5800-Microtower	47m 9s	4	0	0.10.0
d771faea-1d79-4226-b989-33bb3fef5e3b	student-HP-Compaq-dc5800-Microtower	47m 22s	4	0	0.10.0

Showing 1 to 4 of 4 entries



Storm UI

Topology summary

Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Scheduler Info
ids-topology	ids-topology-2-1463987990		ACTIVE	7m 58s	1	18	18	

Topology action

Activate Deactivate Rebal

Do you really want to rebalance topology "ids-topology"? If yes, please, specify wait time in seconds:

30

Cancel OK

Topology stats

Window	Emit			d	Failed
10m 0s	64420	59880	0.000	0	0
3h 0m 0s	64420	59880	0.000	0	0
1d 0h 0m 0s	64420	59880	0.000	0	0
All time	64420	59880	0.000	0	0

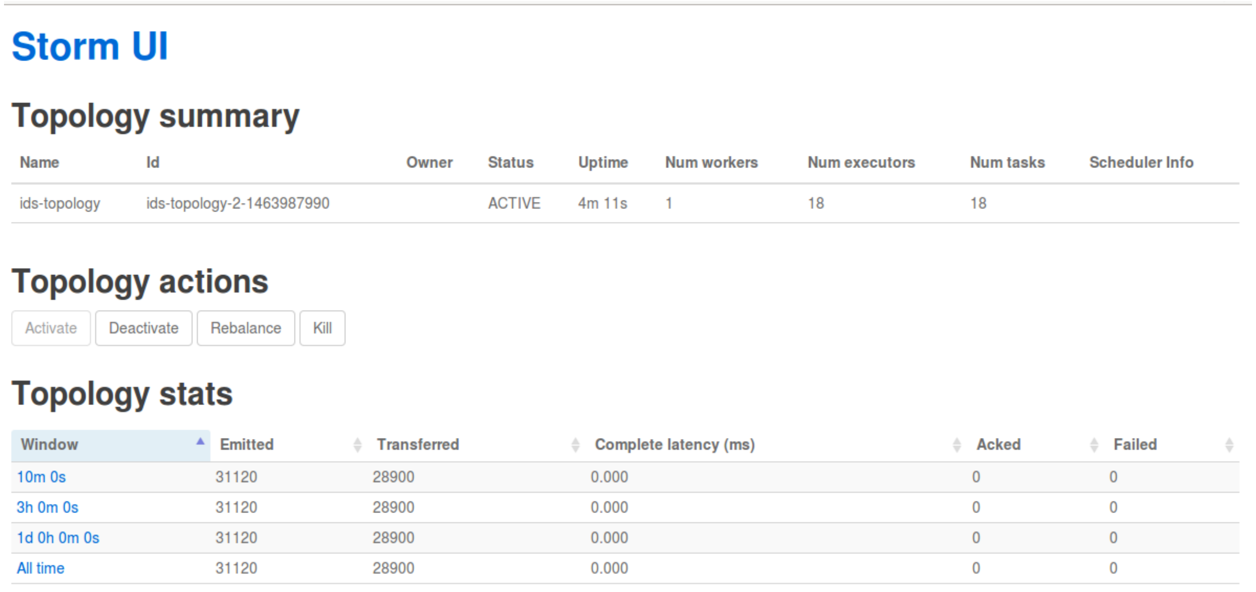
Spouts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Error Host	Error Port	Last error
word	10	10	44900	44900	0.000	0	0			

Showing 1 to 1 of 1 entries

Bolts (All time)



Storm UI

Topology summary

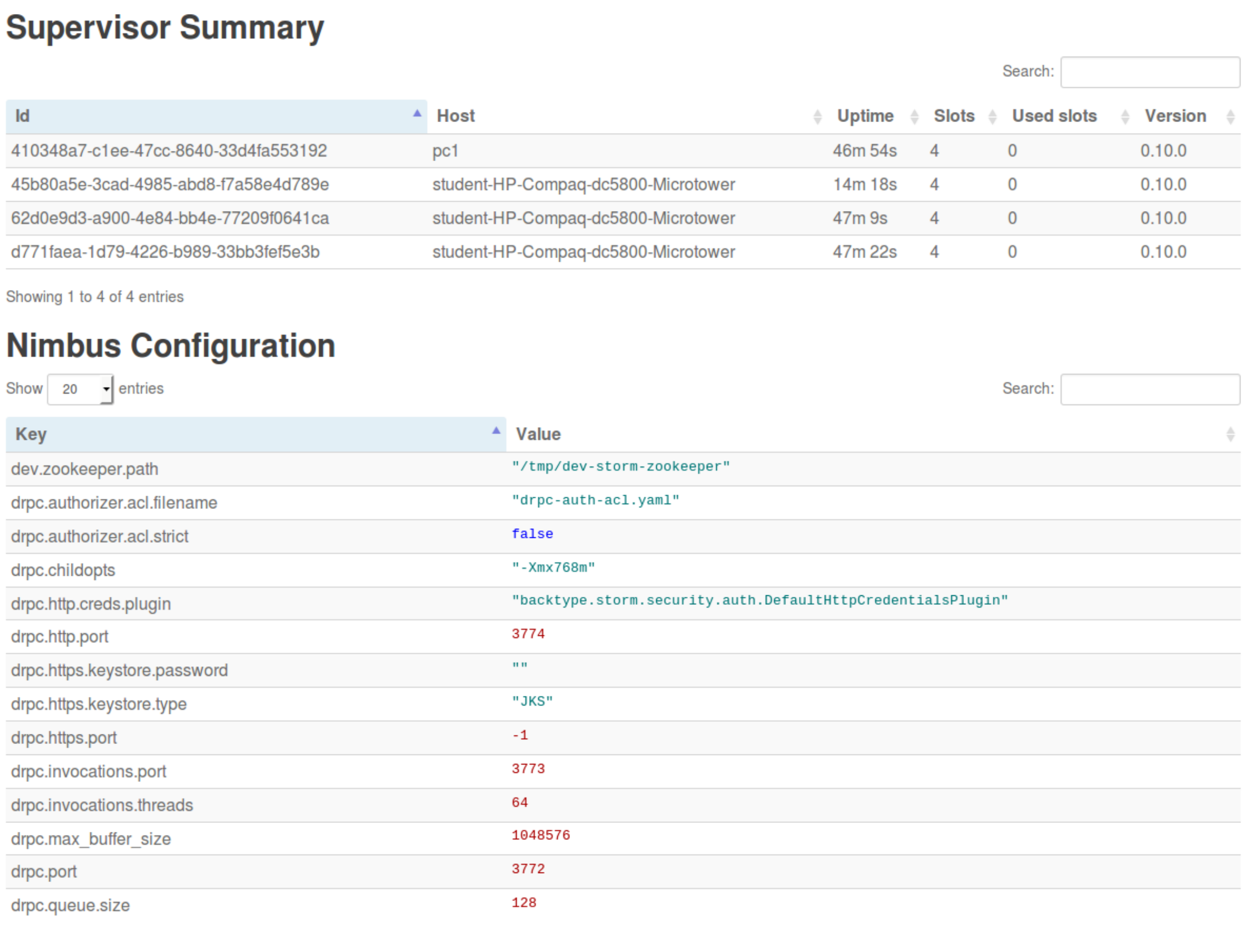
Name	Id	Owner	Status	Uptime	Num workers	Num executors	Num tasks	Scheduler Info
ids-topology	ids-topology-2-1463987990		ACTIVE	4m 11s	1	18	18	

Topology actions

Activate Deactivate Rebalance Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	31120	28900	0.000	0	0
3h 0m 0s	31120	28900	0.000	0	0
1d 0h 0m 0s	31120	28900	0.000	0	0
All time	31120	28900	0.000	0	0



Supervisor Summary

Search:

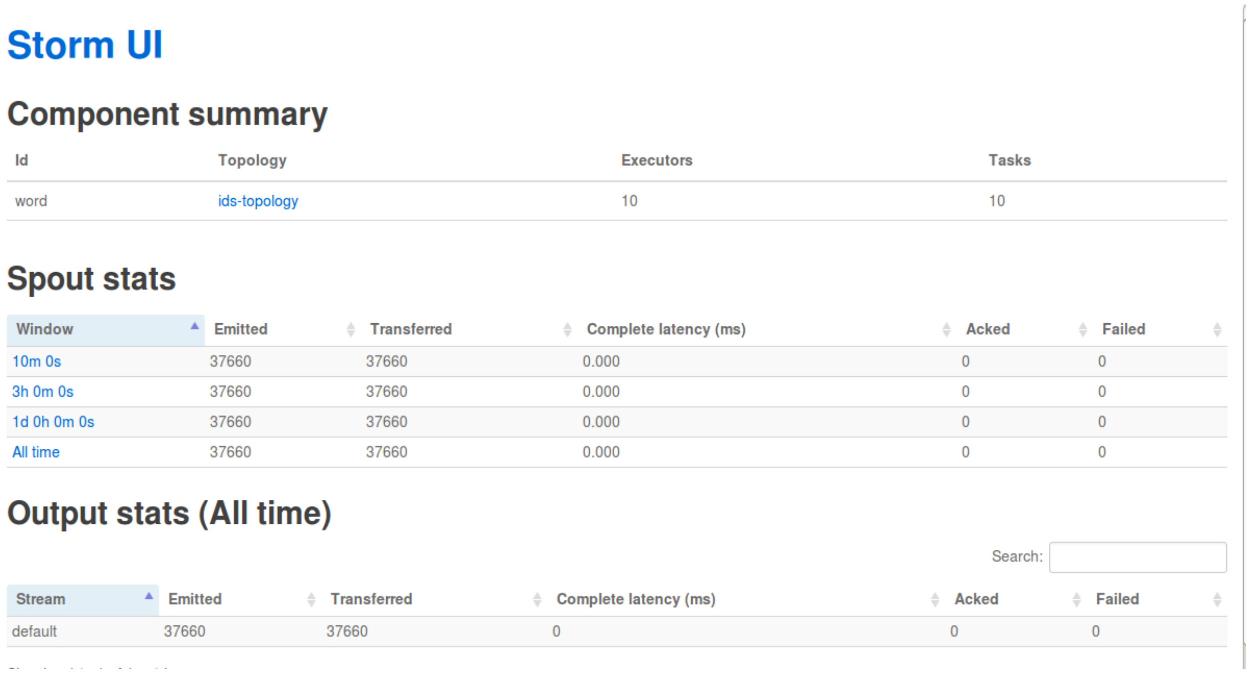
Id	Host	Uptime	Slots	Used slots	Version
410348a7-c1ee-47cc-8640-33d4fa553192	pc1	46m 54s	4	0	0.10.0
45b80a5e-3cad-4985-abd8-f7a58e4d789e	student-HP-Compaq-dc5800-Microtower	14m 18s	4	0	0.10.0
62d0e9d3-a900-4e84-bb4e-77209f0641ca	student-HP-Compaq-dc5800-Microtower	47m 9s	4	0	0.10.0
d771faea-1d79-4226-b989-33bb3fef5e3b	student-HP-Compaq-dc5800-Microtower	47m 22s	4	0	0.10.0

Showing 1 to 4 of 4 entries

Nimbus Configuration

Show 20 entries Search:

Key	Value
dev.zookeeper.path	"/tmp/dev-storm-zookeeper"
drpc.authorizer.acl.filename	"drpc-auth-acl.yaml"
drpc.authorizer.acl.strict	false
drpc.childopts	"-Xmx768m"
drpc.http.creds.plugin	"backtype.storm.security.auth.DefaultHttpCredentialsPlugin"
drpc.http.port	3774
drpc.https.keystore.password	""
drpc.https.keystore.type	"JKS"
drpc.https.port	-1
drpc.invocations.port	3773
drpc.invocations.threads	64
drpc.max_buffer_size	1048576
drpc.port	3772
drpc.queue.size	128



Storm UI

Component summary

Id	Topology	Executors	Tasks
word	ids-topology	10	10

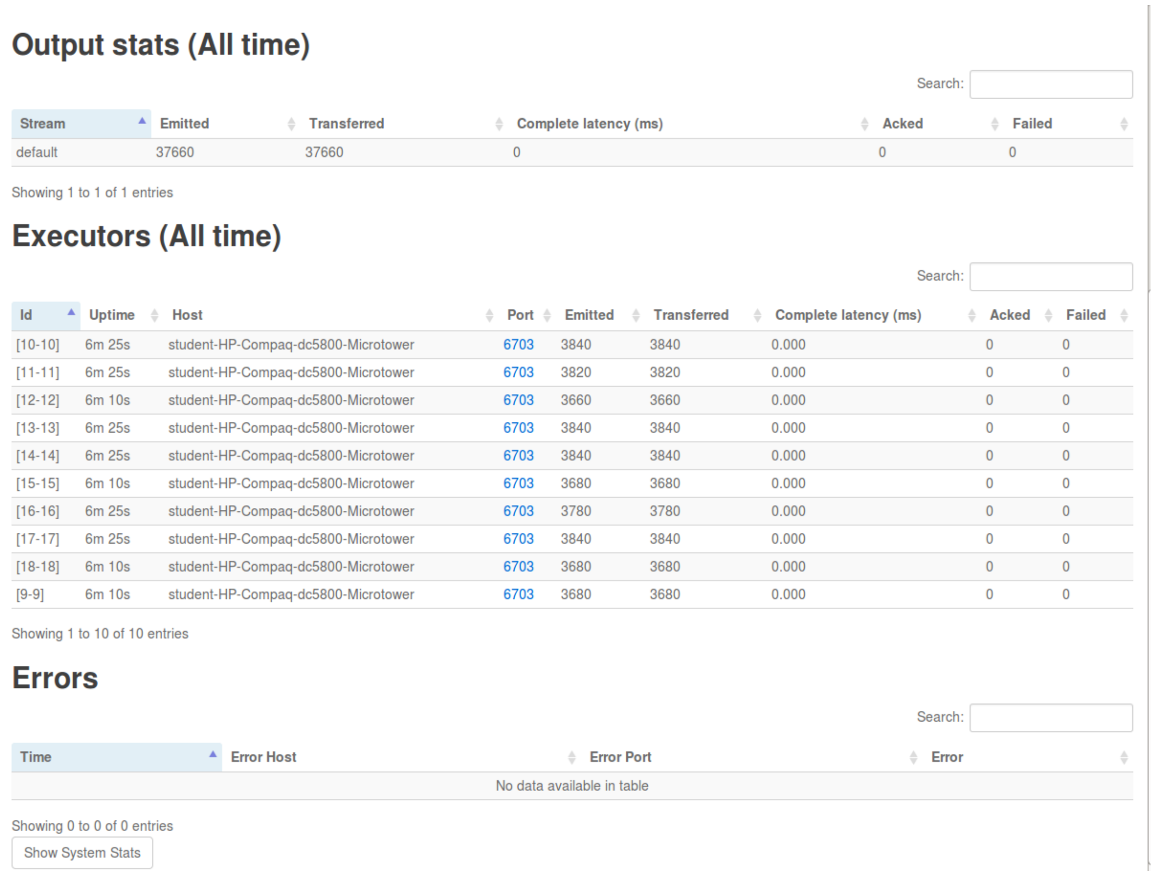
Spout stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	37660	37660	0.000	0	0
3h 0m 0s	37660	37660	0.000	0	0
1d 0h 0m 0s	37660	37660	0.000	0	0
All time	37660	37660	0.000	0	0

Output stats (All time)

Search:

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
default	37660	37660	0	0	0



Output stats (All time)

Search:

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
default	37660	37660	0	0	0

Showing 1 to 1 of 1 entries

Executors (All time)

Search:

Id	Uptime	Host	Port	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[10-10]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3840	3840	0.000	0	0
[11-11]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3820	3820	0.000	0	0
[12-12]	6m 10s	student-HP-Compaq-dc5800-Microtower	6703	3660	3660	0.000	0	0
[13-13]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3840	3840	0.000	0	0
[14-14]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3840	3840	0.000	0	0
[15-15]	6m 10s	student-HP-Compaq-dc5800-Microtower	6703	3680	3680	0.000	0	0
[16-16]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3780	3780	0.000	0	0
[17-17]	6m 25s	student-HP-Compaq-dc5800-Microtower	6703	3840	3840	0.000	0	0
[18-18]	6m 10s	student-HP-Compaq-dc5800-Microtower	6703	3680	3680	0.000	0	0
[9-9]	6m 10s	student-HP-Compaq-dc5800-Microtower	6703	3680	3680	0.000	0	0

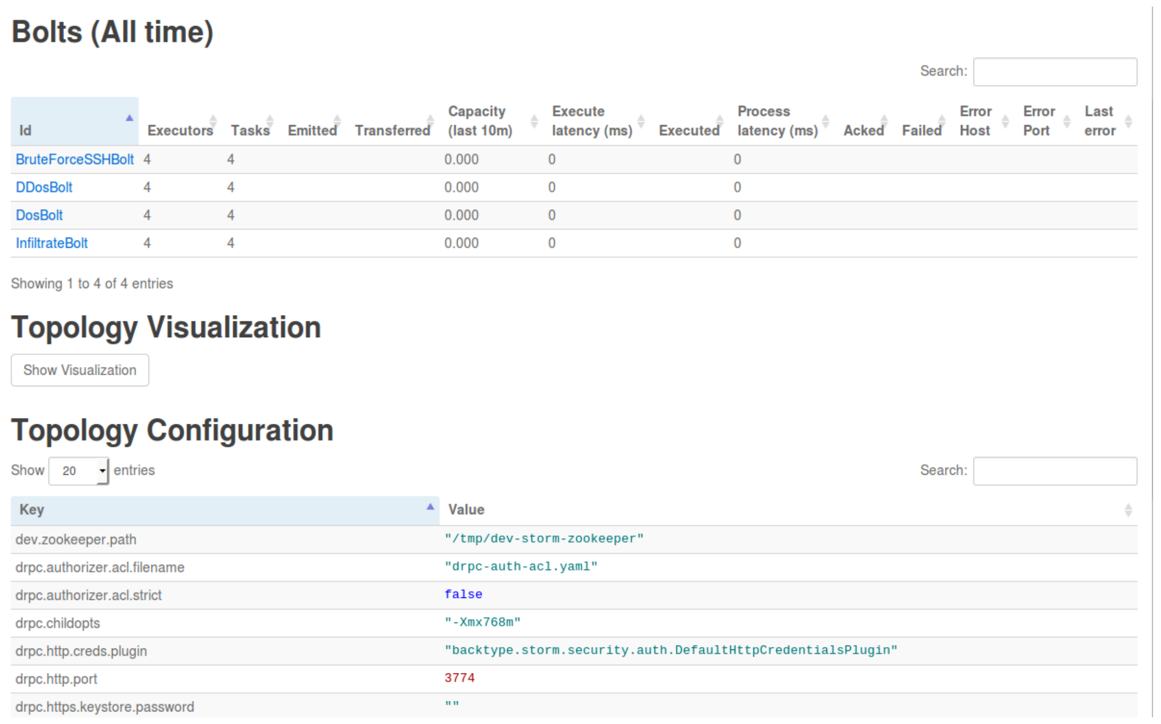
Showing 1 to 10 of 10 entries

Errors

Search:

Time	Error Host	Error Port	Error
	No data available in table		

Showing 0 to 0 of 0 entries



Bolts (All time)

Search:

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Error Host	Error Port	Last error
BruteForceSSHBolt	4	4			0.000	0		0					
DDosBolt	4	4			0.000	0		0					
DosBolt	4	4			0.000	0		0					
InfiltrateBolt	4	4			0.000	0		0					

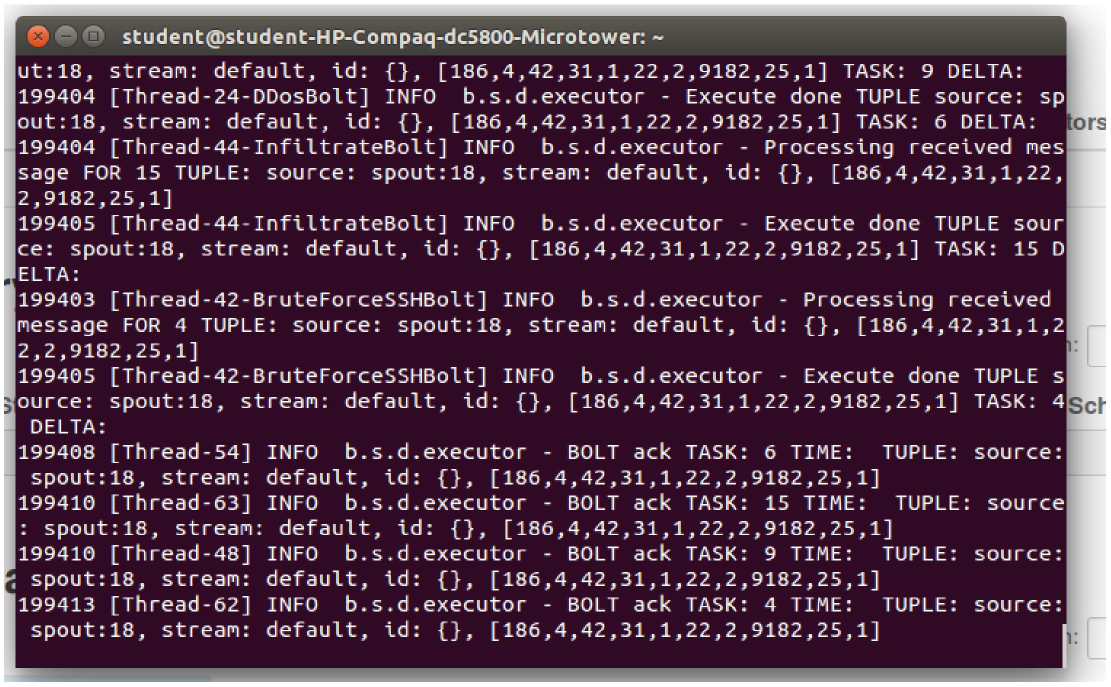
Showing 1 to 4 of 4 entries

Topology Visualization

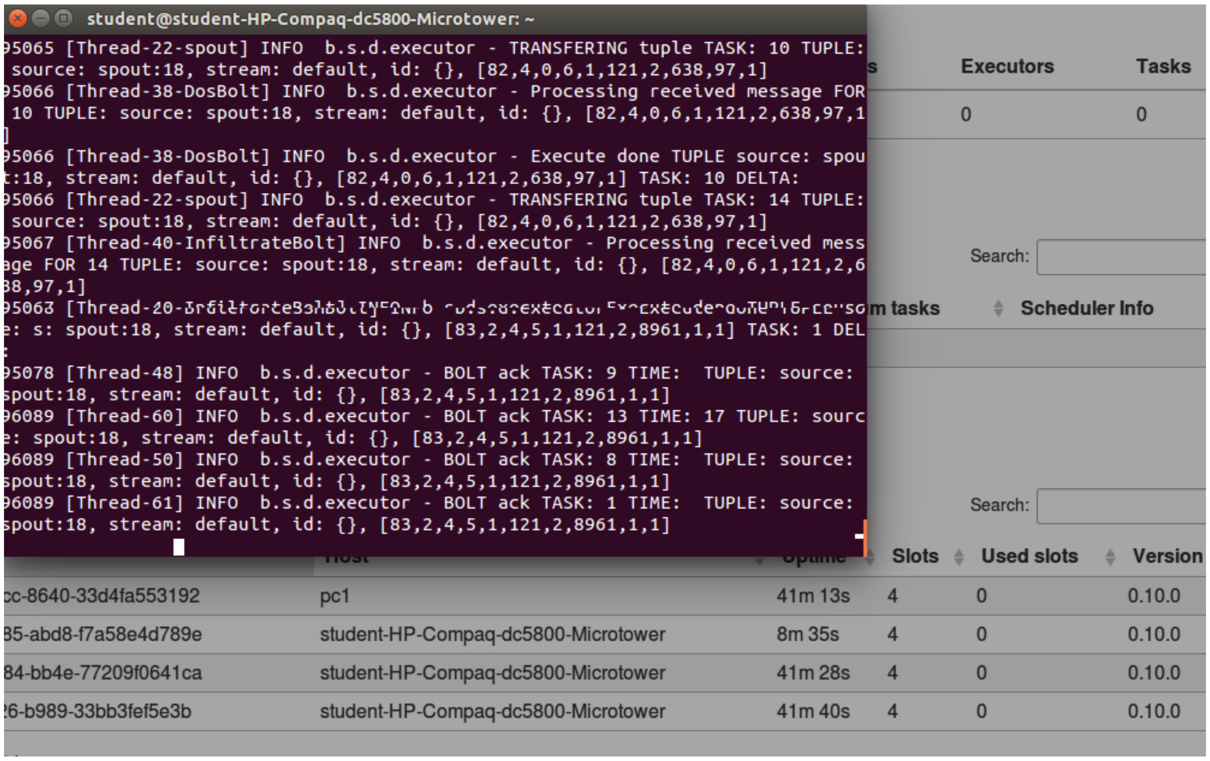
Topology Configuration

Show 20 entries Search:

Key	Value
dev.zookeeper.path	"/tmp/dev-storm-zookeeper"
drpc.authorizer.acl.filename	"drpc-auth-acl.yaml"
drpc.authorizer.acl.strict	false
drpc.childopts	"-Xmx768m"
drpc.http.creds.plugin	"backtype.storm.security.auth.DefaultHttpCredentialsPlugin"
drpc.http.port	3774
drpc.https.keystore.password	""



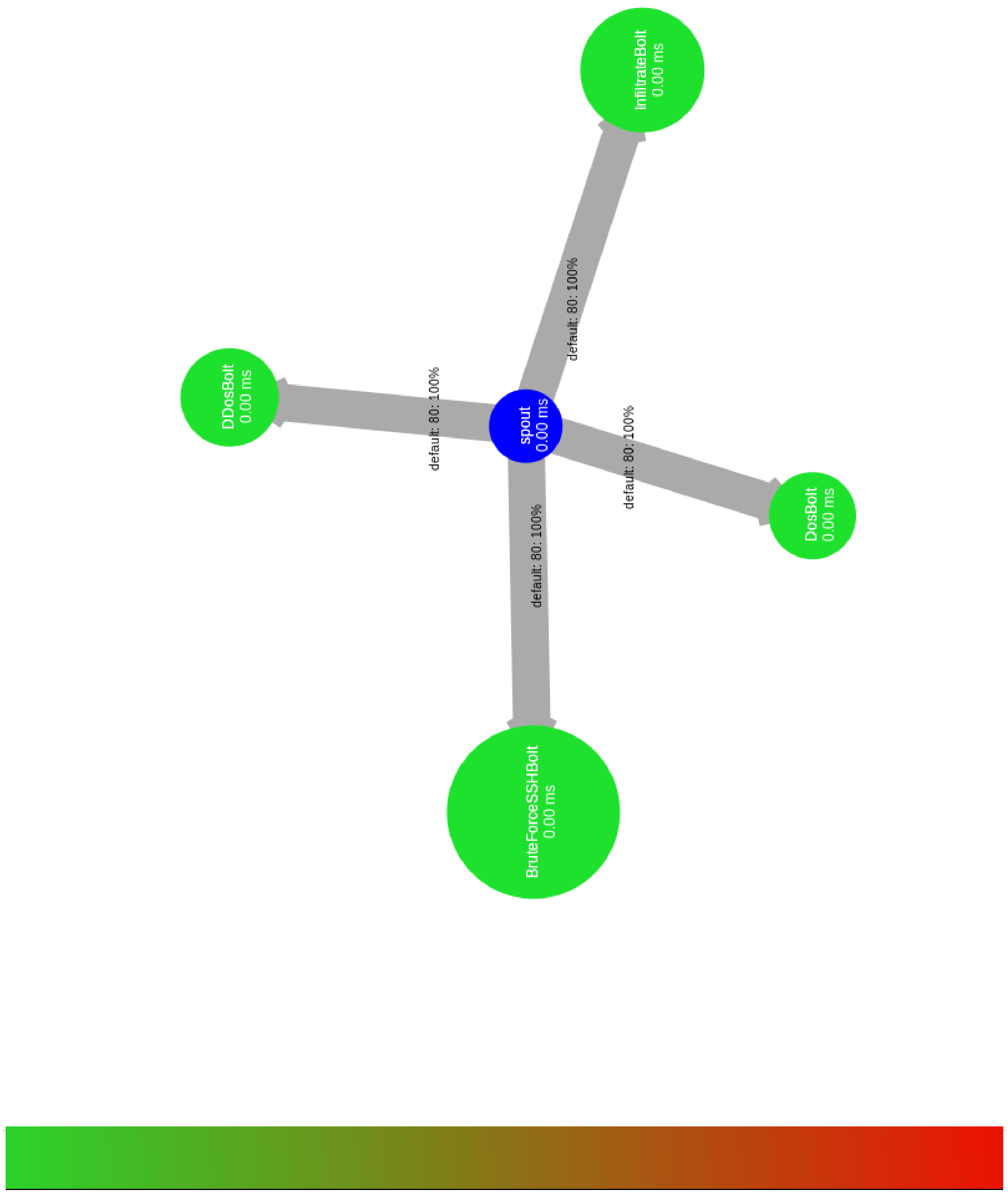
```
student@student-HP-Compaq-dc5800-Microtower: ~
ut:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1] TASK: 9 DELTA:
199404 [Thread-24-DDosBolt] INFO  b.s.d.executor - Execute done TUPLE source: sp
out:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1] TASK: 6 DELTA:
199404 [Thread-44-InfiltrateBolt] INFO  b.s.d.executor - Processing received mes
sage FOR 15 TUPLE: source: spout:18, stream: default, id: {}, [186,4,42,31,1,22,
2,9182,25,1]
199405 [Thread-44-InfiltrateBolt] INFO  b.s.d.executor - Execute done TUPLE sour
ce: spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1] TASK: 15 D
ELTA:
199403 [Thread-42-BruteForceSSHBolt] INFO  b.s.d.executor - Processing received
message FOR 4 TUPLE: source: spout:18, stream: default, id: {}, [186,4,42,31,1,2
2,2,9182,25,1]
199405 [Thread-42-BruteForceSSHBolt] INFO  b.s.d.executor - Execute done TUPLE s
ource: spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1] TASK: 4
 DELTA:
199408 [Thread-54] INFO  b.s.d.executor - BOLT ack TASK: 6 TIME:  TUPLE: source:
 spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1]
199410 [Thread-63] INFO  b.s.d.executor - BOLT ack TASK: 15 TIME:  TUPLE: source
: spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1]
199410 [Thread-48] INFO  b.s.d.executor - BOLT ack TASK: 9 TIME:  TUPLE: source:
 spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1]
199413 [Thread-62] INFO  b.s.d.executor - BOLT ack TASK: 4 TIME:  TUPLE: source:
 spout:18, stream: default, id: {}, [186,4,42,31,1,22,2,9182,25,1]
```

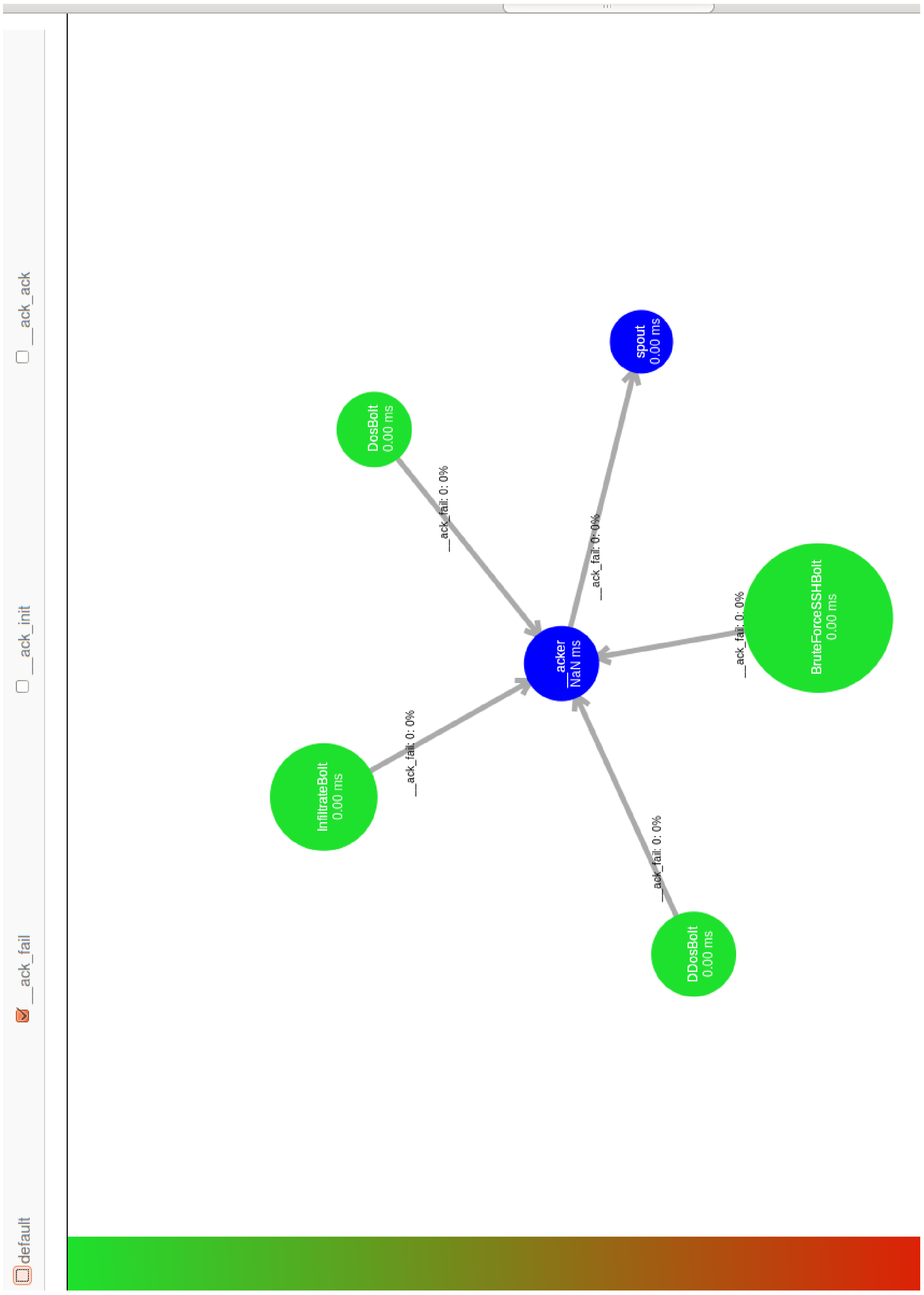


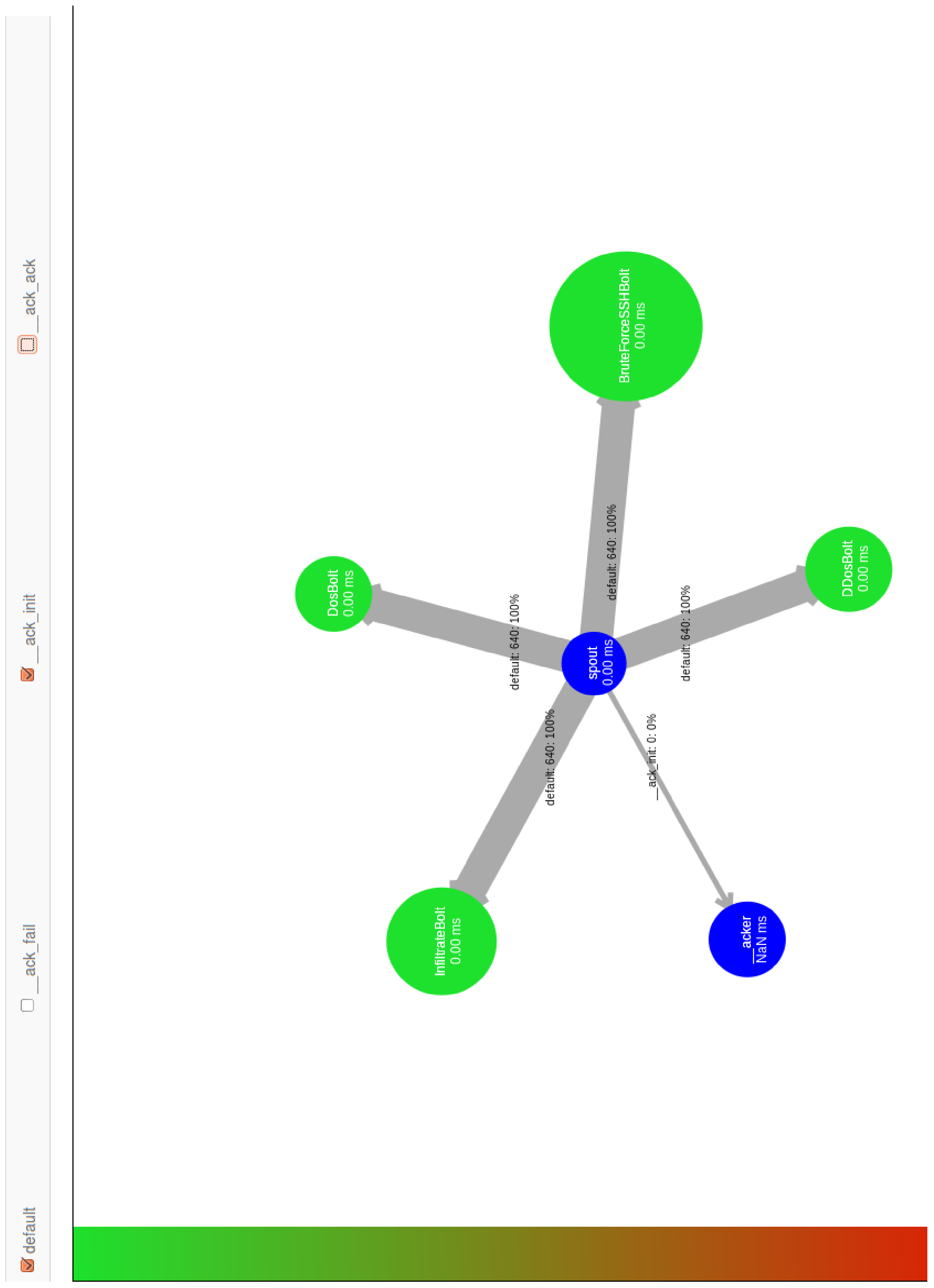
```
student@student-HP-Compaq-dc5800-Microtower: ~
95065 [Thread-22-spout] INFO  b.s.d.executor - TRANSFERING tuple TASK: 10 TUPLE:
 source: spout:18, stream: default, id: {}, [82,4,0,6,1,121,2,638,97,1]
95066 [Thread-38-DosBolt] INFO  b.s.d.executor - Processing received message FOR
 10 TUPLE: source: spout:18, stream: default, id: {}, [82,4,0,6,1,121,2,638,97,1
]
95066 [Thread-38-DosBolt] INFO  b.s.d.executor - Execute done TUPLE source: spou
t:18, stream: default, id: {}, [82,4,0,6,1,121,2,638,97,1] TASK: 10 DELTA:
95066 [Thread-22-spout] INFO  b.s.d.executor - TRANSFERING tuple TASK: 14 TUPLE:
 source: spout:18, stream: default, id: {}, [82,4,0,6,1,121,2,638,97,1]
95067 [Thread-40-InfiltrateBolt] INFO  b.s.d.executor - Processing received mess
age FOR 14 TUPLE: source: spout:18, stream: default, id: {}, [82,4,0,6,1,121,2,6
38,97,1]
95063 [Thread-20-[illegible]
e: s: spout:18, stream: default, id: {}, [83,2,4,5,1,121,2,8961,1,1] TASK: 1 DEL
:
95078 [Thread-48] INFO  b.s.d.executor - BOLT ack TASK: 9 TIME:  TUPLE: source:
spout:18, stream: default, id: {}, [83,2,4,5,1,121,2,8961,1,1]
96089 [Thread-60] INFO  b.s.d.executor - BOLT ack TASK: 13 TIME: 17 TUPLE: sourc
e: spout:18, stream: default, id: {}, [83,2,4,5,1,121,2,8961,1,1]
96089 [Thread-50] INFO  b.s.d.executor - BOLT ack TASK: 8 TIME:  TUPLE: source:
spout:18, stream: default, id: {}, [83,2,4,5,1,121,2,8961,1,1]
96089 [Thread-61] INFO  b.s.d.executor - BOLT ack TASK: 1 TIME:  TUPLE: source:
spout:18, stream: default, id: {}, [83,2,4,5,1,121,2,8961,1,1]
```

Executors	Tasks
0	0

	Host	Uptime	Slots	Used slots	Version
cc-8640-33d4fa553192	pc1	41m 13s	4	0	0.10.0
85-abd8-f7a58e4d789e	student-HP-Compaq-dc5800-Microtower	8m 35s	4	0	0.10.0
84-bb4e-77209f0641ca	student-HP-Compaq-dc5800-Microtower	41m 28s	4	0	0.10.0
26-b989-33bb3fef5e3b	student-HP-Compaq-dc5800-Microtower	41m 40s	4	0	0.10.0


InfiltrateBolt
0.00 ms
default: 80: 100%
DDosBolt
0.00 ms
default: 80: 100%
spout
0.00 ms
default: 80: 100%
DosBolt
0.00 ms
default: 80: 100%
BruteForceSSHBolt
0.00 ms


default
__ack_fail
__ack_init
__ack_ack
spout
0.00 ms
DosBolt
0.00 ms
__acker
NaN ms
BruteForceSSHBolt
0.00 ms
InfiltrateBolt
0.00 ms
DDosBolt
0.00 ms
__ack_fail: 0: 0%
__ack_fail: 0: 0%
__ack_fail: 0: 0%
__ack_fail: 0: 0%
__ack_fail: 0: 0%


✓ default
__ack_fail
✓ __ack_init
__ack_ack
BruteForceSSHBolt
0.00 ms
DosBolt
0.00 ms
DDosBolt
0.00 ms
spout
0.00 ms
InfiltrateBolt
0.00 ms
__acker
NaN ms
default: 640: 100%
default: 640: 100%
default: 640: 100%
default: 640: 100%
__ack_init: 0: 0%

5.10 Deployment Model

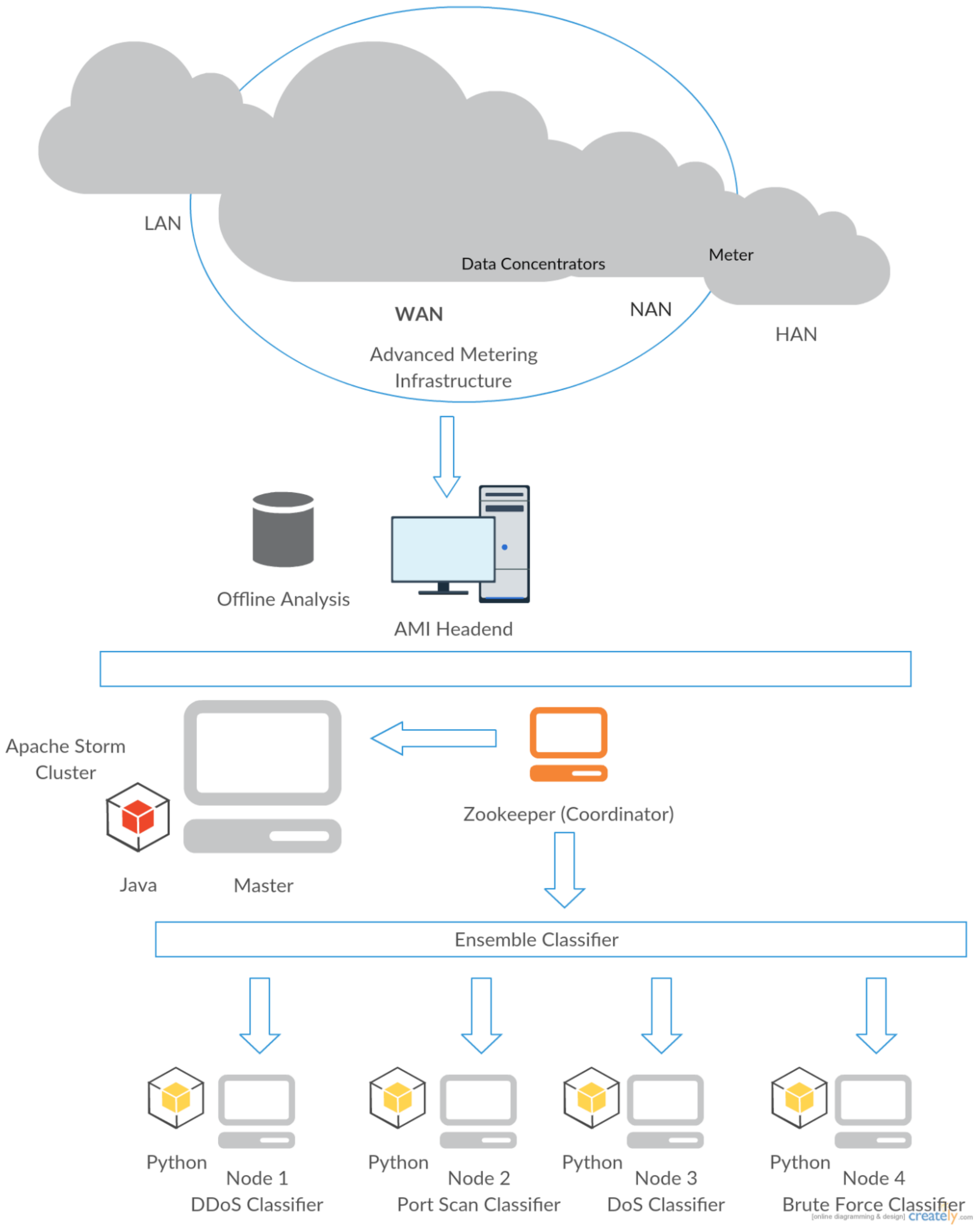


Figure 5.8 Deployment Model on the AMI Infrastructure

The physical mapping between software components and the underlying physical hardware is as shown in the figure on the previous page. The system consists of one master node (known as Nimbus in storm terminology), a number of slave nodes (ideally 4 or multiples of 4, for each classifier) and one Zookeeper node, which is responsible for maintaining the state of the system and interfacing between the master node and the slaves. The system as a whole is deployed as a cluster running on the AMI head-end. The topology and the input Spout are both written using the Java programming language. This choice was dictated by Storm's built in support for Java. The Bolts are written using the Python programming language was by our choice of sklearn as the base library to take advantage of Python's fantastic support for scientific and mathematical computing. Apache Storm does the logical to physical mapping at runtime. It does this via the topology construct built into storm. A machine in a Storm cluster may run one or more than one worker processes for one or more topologies. Each worker process runs executors for a specific topology. A worker process executed a subset of a topology, in our case the IDS topology.
To configure the parallelism of a topology, we can modify:

1. Number of worker processes: How many worker processes to create *for the topology* across machines in the cluster
2. Number of executors (threads): How many executors to spawn *per component*.
3. Number of tasks: How many tasks to create *per component*

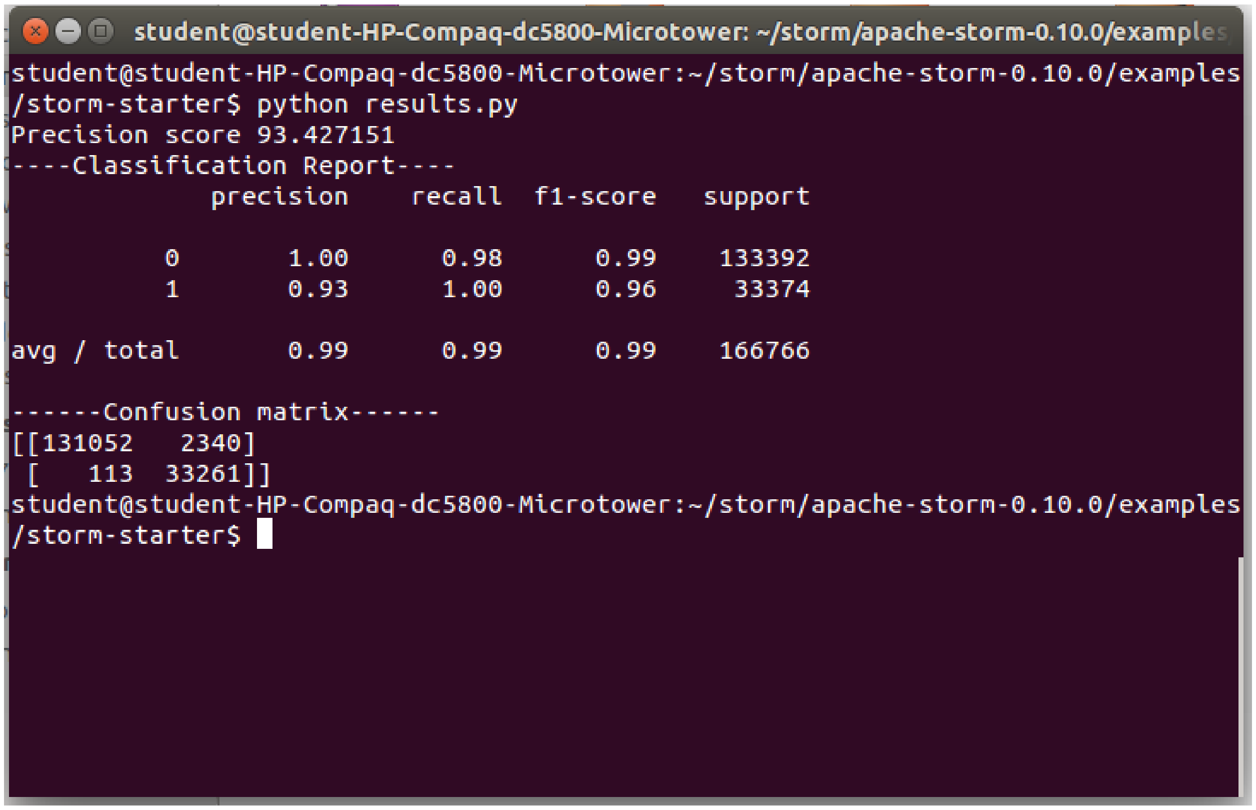
A nifty feature built into Storm is that of *rebalancing.* Rebalancing is increasing or decreasing the number of worker processes and/or executors without being required to start the cluster or the topology.

CHAPTER 6: CONCLUSION AND FUTURE SCOPE

We received promising results when we tested our trained ensemble Decision Trees classifier against 166766 network packets which include diverse intrusion scenarios. We received an accuracy of 93.42%.

Recognition systems, in most cases cannot have 100% accurate results, since they are trained using data, which may or may not match the nature of data that is used for testing. Hence, there are false positives and false negatives of the recognition system. In our work we used the ensemble Decision trees classifier which yielded the following results:

1. True positive: 131052
2. False negative: 2340 (1.40%)
3. False Positive: 113 (0.06%)
4. True negative: 33261



student@student-HP-Compaq-dc5800-Microtower: ~/storm/apache-storm-0.10.0/examples

```
student@student-HP-Compaq-dc5800-Microtower:~/storm/apache-storm-0.10.0/examples
/storm-starter$ python results.py
Precision score 93.427151
----Classification Report----
             precision    recall  f1-score   support

          0       1.00      0.98      0.99    133392
          1       0.93      1.00      0.96     33374

avg / total       0.99      0.99      0.99    166766

------Confusion matrix------
[[131052   2340]
 [   113  33261]]
student@student-HP-Compaq-dc5800-Microtower:~/storm/apache-storm-0.10.0/examples
/storm-starter$ 
```

A novel way of implementing an efficient real-time Distributed Intrusion Detection System has been developed, that consists of an anomaly detection for four different types of network attacks – Denial of Service, Port Scanning, Distributed Denial of Service and Brute Force SSH. The entire proof of concept was performed inside the Apache Storm cluster with the ISCX 2012 intrusion dataset.

The feature set offered by Apache Storm makes it a true contender for building a reliable, scalable and real-time IDS. At the AMI headend infrastructure it is easy to deploy our Apache Storm cluster using commodity hardware as per requirement.

For future work, we can take advantage of other supervised learning methods to classify different network attacks apart from the 4 already being detected. Also, we can add unsupervised learning methods to predict novel attacks such as with the use of Neural Networks.

BIBLIOGRAPHY

[1] Robin Berthier, William H. Sanders and Himanshu Khurana, "*Intrusion Detection for Advanced Metering Infrastructures: Requirements and Architectural Directions*", 350-355, Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference.

[2] Zuech, Richard, Taghi M. Khoshgoftaar, and Randall Wald. "Intrusion detection and big heterogeneous data: A survey." Journal of Big Data 2.1 (2015): 1-41.

[3] Sivaraman, Vijay, et al. "Network-level security and privacy control for smart-home IoT devices." Wireless and Mobile Computing, Networking and Communications (WiMob), 2015 IEEE 11th International Conference on. IEEE, 2015.

[4] Devikrishna, K. S., and B. B. Ramakrishna. "An Artificial Neural Network based Intrusion Detection System and Classification of Attacks." (1959).

[5] Mylavarapu, Goutam, Johnson Thomas, and Ashwin Kumar TK. "Real-Time Hybrid Intrusion Detection System Using Apache Storm." High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, 2015.

[6] Shiravi, Ali, et al. "Toward developing a systematic approach to generate benchmark datasets for intrusion detection." Computers & Security 31.3 (2012): 357-374.

[7] Balupari, Ravindra, et al. Real-time network-based anomaly intrusion detection. Nova Science Publishers, Inc., 2003.

[8] Zuech, Richard, et al. "A New Intrusion Detection Benchmarking System." The Twenty-Eighth International Flairs Conference. 2015.

[9] Engen, Vegard, Jonathan Vincent, and Keith Phalp. "Exploring discrepancies in findings obtained with the KDD Cup'99 data set." Intelligent Data Analysis 15.2 (2011): 251-276.

[10] Markov, Zdravko, and Ingrid Russell. "An introduction to the WEKA data mining system." ACM SIGCSE Bulletin. Vol. 38. No. 3. ACM, 2006.

[11] Suo, Hui, et al. "Security in the internet of things: a review." Computer Science and Electronics Engineering (ICCSEE), 2012 International Conference on. Vol. 3. IEEE, 2012.

[12] Mylavarapu, Goutam, Johnson Thomas, and Ashwin Kumar TK. "Real-Time Hybrid Intrusion Detection System Using Apache Storm." High Performance Computing and

Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE, 2015.

[13] Faisal, Mustafa Amir, et al. "Data-stream-based intrusion detection system for advanced metering infrastructure in smart grid: A feasibility study." Systems Journal, IEEE 9.1 (2015): 31-44.

[14] (2012) Massive Online Analysis, release 2012.03. http://moa.cs.waikato.ac.nz

[15] Faisal, Mustafa Amir, et al. "Securing advanced metering infrastructure using intrusion detection system with data stream mining." Intelligence and Security Informatics. Springer Berlin Heidelberg, 2012. 96-111..

[16] Tavallaee, Mahbod, et al. "A detailed analysis of the KDD CUP 99 data set." Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications 2009. 2009.

[17] “Waikato environment for knowledge analysis (weka) version 3.5.7.” Available on: http://www.cs.waikato.ac.nz/ml/weka/, June, 2008.

[18] L. Breiman, “Random Forests,” Machine Learning, vol. 45, no. 1, pp. 5–32, 2001

[19] KDD Cup 1999. Available on: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, Ocotber 2007

[20] Shiravi, Ali, et al. "Toward developing a systematic approach to generate benchmark datasets for intrusion detection." Computers & Security 31.3 (2012): 357-374.

[21] Bifet, Albert, et al. "Moa: Massive online analysis." The Journal of Machine Learning Research 11 (2010): 1601-1604..

[22] Garcia-Teodoro, Pedro, et al. "Anomaly-based network intrusion detection: Techniques, systems and challenges." computers & security 28.1 (2009): 18-28.

[23] http://effbot.org/zone/celementtree.htm

[24] https://maven.apache.org/

[25] https://en.wikipedia.org/wiki/Weka_(machine_learning)

[26] http://www.cs.waikato.ac.nz/ml/weka/

[27] https://en.wikipedia.org/wiki/Random_forest

[28] https://en.wikipedia.org/wiki/Support_vector_machine

[29] http://www.michael-noll.com/tutorials/running-multi-node-storm-cluster/

APPENDIX

IDSTopology.java

```
package storm.starter;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.StormSubmitter;
import backtype.storm.task.ShellBolt;
import backtype.storm.topology.BasicOutputCollector;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.topology.base.BaseBasicBolt;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;
import storm.starter.spout.CSVSpout;
import java.util.HashMap;
import java.util.Map;


/**
 * This topology demonstrates Storm's stream groupings and multilang
capabilities.
 */
public class IDSTopology {

  public static class DosBolt extends ShellBolt implements IRichBolt {

    public DosBolt() {
      super("python","DosBolt.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      //declarer.declare(new Fields(String[]));
      declarer.declare(new Fields("packet"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }
```

```
    public static class DDosBolt extends ShellBolt implements IRichBolt {

    public DDosBolt() {
      super("python","DDosBolt.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("packet"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }

        public static class InfiltrateBolt extends ShellBolt implements
IRichBolt {

    public InfiltrateBolt() {
      super("python","InfiltrateBolt.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("packet"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }

   public static class BruteForceSSHBolt extends ShellBolt implements
IRichBolt {

    public BruteForceSSHBolt() {
      super("python","BruteForceSSHBolt.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("packet"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }

  /*
  public static class AggregatorBolt extends ShellBolt implements IRichBolt {

    public AggregatorBolt() {
      super("python","AggregatorBolt.py");
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
      declarer.declare(new Fields("packet"));
    }

    @Override
    public Map<String, Object> getComponentConfiguration() {
      return null;
    }
  }*/


  public static void main(String[] args) throws Exception {

    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("spout", new CSVSpout(), 1);
          builder.setBolt("BruteForceSSHBolt", new BruteForceSSHBolt(),
4).shuffleGrouping("spout");
    builder.setBolt("DosBolt", new DosBolt(), 4).shuffleGrouping("spout");
    builder.setBolt("DDosBolt", new DDosBolt(), 4).shuffleGrouping("spout");
    builder.setBolt("InfiltrateBolt", new InfiltrateBolt(),
4).shuffleGrouping("spout");
    //builder.setBolt("AggregatorBolt",new AggregatorBolt(),
4).shuffleGrouping("DosBolt").shuffleGrouping("DDosBolt").shuffleGrouping("In
filtrateBolt").shuffleGrouping("BruteForceSSHBolt");
    Config conf = new Config();
    conf.setDebug(true);
    if (args != null && args.length > 0) {
      conf.setNumWorkers(3);
      StormSubmitter.submitTopologyWithProgressBar(args[0], conf,
builder.createTopology());
    }
    else {
      conf.setMaxTaskParallelism(4);

      LocalCluster cluster = new LocalCluster();
      cluster.submitTopology("ids-topology", conf, builder.createTopology());
      Thread.sleep(10000);
    }
  }
}
```

CSVSpout.java

```
package storm.starter.spout;

import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.utils.Utils;

import java.util.Map;
import java.util.Random;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class CSVSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  FileReader file;
  BufferedReader br = null;
  String cvsSplitBy = ",";
  String packets[] = new String[166766];
  String line = "";
  static int index = 0;


  @Override
  public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {
    _collector = collector;
    int i = 0;
    String csvFile = "/home/student/storm/apache-storm-0.10.0/examples/storm-
starter/src/jvm/storm/starter/test-data/MasterTest.csv";

        try {
                br = new BufferedReader(new FileReader(csvFile));
                while( (line = br.readLine()) != null)
                {
                        packets[i++] = line;
                }

        } catch (FileNotFoundException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        } finally {
                if (br != null) {
                        try {
                                br.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }

  }

  @Override
  public void nextTuple() {
        Utils.sleep(1000);
        if(index > 166765)
        {
            index = 0;
        }
        else
        {
         _collector.emit(new Values(packets[index++]));
        }

   }

  @Override
  public void ack(Object id) {
  }

  @Override
  public void fail(Object id) {
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("packet"));
  }

}
```

BruteForceSSHBolt.py

```
import storm
import pickle
from sklearn.externals import joblib
from sklearn import *
import numpy as np
import sys

class BruteForceSSHBolt(storm.BasicBolt):
        counter = 0
        def process(self,tup):
                global counter
                features = tup.values[0].split(",")
                features_int = map(int, features)
                relevant_features_list = features_int[1:-1]
                relevant_features_np = np.asarray(relevant_features_list)
                relevant_features = relevant_features_np.reshape(1,-1)
                clf = joblib.load('/home/student/storm/apache-storm-
0.10.0/project/code/BruteForceTrain.pkl')
                predict = clf.predict(relevant_features)

        #       if predict[0] == 2:
        #               with open("", "a") as f:


                if predict[0] == 1:
                        predict[0] = 0
                else:
                        predict[0] = 1

                UID = np.asarray(features[0])
                if BruteForceSSHBolt.counter < 166766:
                        with open("/home/student/BruteForceagg.txt", "a") as f:
                                UID.tofile(f, sep = ",")
                                f.write(",")
                                for element in relevant_features_list:
                                        relevant_feature = np.asarray(element)
                                        relevant_feature.tofile(f, sep=",")
                                        f.write(",")
                                predict.tofile(f, sep=",")
                                f.write("\n")
                        BruteForceSSHBolt.counter += 1

BruteForceSSHBolt().run()
```

DoSBolt.py

```
import storm
import pickle
from sklearn.externals import joblib
from sklearn import *
import numpy as np
import sys

class DosBolt(storm.BasicBolt):
        counter = 0
        def process(self,tup):
                global counter
                features = tup.values[0].split(",")
                features_int = map(int, features)
                relevant_features_list = features_int[1:-1]
                relevant_features_np = np.asarray(relevant_features_list)
                relevant_features = relevant_features_np.reshape(1,-1)
                clf = joblib.load('/home/student/storm/apache-storm-
0.10.0/project/code/DosTrain.pkl')
                predict = clf.predict(relevant_features)

                if (predict[0] == 1):
                        predict[0] = 0
                else:
                        predict[0] = 1
                UID = np.asarray(features[0])
                if DosBolt.counter < 166766:
                        with open("/home/student/Dosagg.txt", "a") as f:
                                UID.tofile(f, sep = ",")
                                f.write(",")
                                for element in relevant_features_list:
                                        relevant_feature = np.asarray(element)
                                        relevant_feature.tofile(f, sep=",")
                                        f.write(",")
                                predict.tofile(f, sep=",")
                                f.write("\n")
                        DosBolt.counter += 1

DosBolt().run()
```

DDoSBolt.py

```
import storm
import pickle
from sklearn.externals import joblib
from sklearn import *
import numpy as np
import sys

class DDosBolt(storm.BasicBolt):
        counter = 0
        def process(self,tup):
                global counter
                features = tup.values[0].split(",")
                features_int = map(int, features)
                relevant_features_list = features_int[1:-1]
                relevant_features_np = np.asarray(relevant_features_list)
                relevant_features = relevant_features_np.reshape(1,-1)
                clf = joblib.load('/home/student/storm/apache-storm-
0.10.0/project/code/DDosTrain.pkl')
                predict = clf.predict(relevant_features)

                if predict[0] == 1:
                        predict[0] = 0
                else:
                        predict[0] = 1
                UID = np.asarray(features[0])
                if DDosBolt.counter < 166766:
                        with open("/home/student/DDosagg.txt", "a") as f:
                                UID.tofile(f, sep = ",")
                                f.write(",")
                                for element in relevant_features_list:
                                        relevant_feature = np.asarray(element)
                                        relevant_feature.tofile(f, sep=",")
                                        f.write(",")
                                predict.tofile(f, sep=",")
                                f.write("\n")
                        DDosBolt.counter += 1

DDosBolt().run()
```

InfiltrateBolt.py

```
import storm
import pickle
from sklearn.externals import joblib
from sklearn import *
import numpy as np
import sys

class InfiltrateBolt(storm.BasicBolt):
        counter = 0
        def process(self,tup):
                global counter
                features = tup.values[0].split(",")
                features_int = map(int, features)
                relevant_features_list = features_int[1:-1]
                relevant_features_np = np.asarray(relevant_features_list)
                relevant_features = relevant_features_np.reshape(1,-1)
                clf = joblib.load('/home/student/storm/apache-storm-
0.10.0/project/code/InfiltrateTrain.pkl')
                predict = clf.predict(relevant_features)

                if (predict[0] == 1):
                        predict[0] = 0
                else:
                        predict[0] = 1

                UID = np.asarray(features[0])
                if InfiltrateBolt.counter < 166766:
                        with open("/home/student/Infiltrateagg.txt", "a") as f:
```

```
                                                        UID.tofile(f, sep = ",")
                                                        f.write(",")
                                                        for element in relevant_features_list:
                                                                relevant_feature = np.asarray(element)
                                                                relevant_feature.tofile(f, sep=",")
                                                                f.write(",")
                                                        predict.tofile(f, sep=",")
                                                        f.write("\n")
                                                InfiltrateBolt.counter += 1

InfiltrateBolt().run()
```

Load.py

```
import os
import csv
import sys
import shutil
from os import environ
from os.path import join
import sklearn.cross_validation

import numpy as np
from sklearn.cross_validation import StratifiedKFold

class Bunch(dict):
    """Container object for datasets
    Dictionary-like object that exposes its keys as attributes.
    >>> b = Bunch(a=1, b=2)
    >>> b['b']
    2
    >>> b.b
    2
    >>> b.a = 3
    >>> b['a']
    3
    >>> b.c = 6
    >>> b['c']
    6
    """

    def __init__(self, **kwargs):
        dict.__init__(self, kwargs)

    def __setattr__(self, key, value):
        self[key] = value

    def __getattr__(self, key):
        try:
            return self[key]
        except KeyError:
            raise AttributeError(key)

    def __setstate__(self, state):
        # Bunch pickles generated with scikit-learn 0.16.* have an non
        # empty __dict__. This causes a surprising behaviour when
        # loading these pickles scikit-learn 0.17: reading bunch.key
        # uses __dict__ but assigning to bunch.key use __setattr__ and
        # only changes bunch['key']. More details can be found at:
        # https://github.com/scikit-learn/scikit-learn/issues/6196.
        # Overriding __setstate__ to be a noop has the effect of
        # ignoring the pickled __dict__
        pass

def load_iscx():
    """Load and return the ISCX dataset (classification).
    The ISCX dataset is a classic and very easy multi-class classification
    dataset.
    =================   ==============
    Classes                          2
    Samples per class
    Samples total                  150
    Dimensionality                   9
    Features            real, positive
    =================   ==============
    -------
    data : Bunch
        Dictionary-like object, the interesting attributes are:
        'data', the data to learn, 'target', the classification labels,
        'target_names', the meaning of the labels, 'feature_names', the
        meaning of the features, and 'DESCR', the
        full description of the dataset.
    """
    #These are the 4 CSV Files

    #'csvBruteForceRandom.csv', 'csvDosRandom.csv', csvDDosRandom.csv,
csvInfiltrateRandom.csv

    randomcsv = '../data/csvInfiltrateTrain.csv'
    with open(randomcsv) as csv_file:
        data_file = csv.reader(csv_file)
        temp = next(data_file)
        n_samples = int(temp[0])
        n_features = int(temp[1]) - 2
        target_names = np.array(temp[2:])
        data = np.empty((n_samples, n_features))
        target = np.empty((n_samples,), dtype=np.int)

  #      print n_samples, n_features, target_names, data, target

        for i, ir in enumerate(data_file):
            data[i] = np.asarray(ir[1:-1], dtype=np.int)
            target[i] = np.asarray(ir[-1], dtype=np.int)

   #     print data, target

    print "For " + randomcsv

    skf = StratifiedKFold(target, n_folds = 2)



    return Bunch(data=data, target=target,
                 target_names=target_names,

feature_names=['appName','totalDestinationPackets','totalSourcePackets','dire
ction',

'sourceIP','protocolName','destinationIP','duration','target']), skf


#    printskf.
```

Train.py

```
import load
from sklearn import tree
from sklearn.externals.six import StringIO
import sklearn.cross_validation
from sklearn.ensemble import RandomForestClassifier
from  sklearn.svm import SVC
from sklearn.cross_validation import train_test_split
import pickle
from sklearn.externals import joblib
```

```
#import time
data, skf= load.load_iscx()

X, y = data.data, data.target


clf = tree.DecisionTreeClassifier(criterion='entropy')
clf = clf.fit(X,y)
#clf = SVC()


scores = sklearn.cross_validation.cross_val_score(clf, data.data,
data.target, cv=skf)

print scores

#for train_index, test_index in skf:
#       clf = clf.fit(X[train_index], y[train_index])
#       print
        sklearn.metrics.recall_score(y[test_index],clf.predict(X[test_index]))


joblib.dump(clf, 'InfiltrateTrain.pkl')

#clf = joblib.load('filename.pkl')

    f = tree.export_graphviz(clf, out_file=f)
```

Results.py

```
from sklearn.metrics import precision_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix

f_Dos = open("/home/student/Dosagg_sorted.txt")
f_DDos = open("/home/student/DDosagg_sorted.txt")
f_Infiltrate = open("/home/student/Infiltrateagg_sorted.txt")
f_BruteForceSSH = open("/home/student/BruteForceagg_sorted.txt")
f_testData = open("/home/student/storm/apache-storm-0.10.0/examples/storm-
starter/src/jvm/storm/starter/test-data/MasterTest.csv")
```

```
predList = []
testList = []
for i in range(0,166766):
        dos_line = f_Dos.readline()
        ddos_line = f_DDos.readline()
        infil_line = f_Infiltrate.readline()
        brute_line = f_BruteForceSSH.readline()
        testData_line = f_testData.readline()
        pred_dos = int(dos_line.split(",")[9])
        pred_ddos = int(ddos_line.split(",")[9])
        pred_infil = int(infil_line.split(",")[9])
        pred_brute = int(brute_line.split(",")[9])
        pred = pred_dos | pred_ddos | pred_brute | pred_infil
        pred_test = int(testData_line.split(",")[9])
        if pred_test == 1:
                pred_test = 0
        else:
                pred_test = 1
        predList.append(pred)
        testList.append(pred_test)
precscore = precision_score(testList, predList)
print "Precision score %f" % (precscore * 100)

print "----Classification Report----"
report = classification_report(testList, predList)


print report

print "------Confusion matrix------"

cf = confusion_matrix(testList, predList)

print cf
```

attach.py

```
import smtplib
from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText
from email.MIMEBase import MIMEBase
from email import encoders
```

```
fromaddr = "p.blesson@sitpune.edu.in"
toaddr = "psblesson@gmail.com"

msg = MIMEMultipart()

msg['From'] = fromaddr
msg['To'] = toaddr
msg['Subject'] = "Intruder Alert!"

body = "Please check the attachment in relation to suspicious activity on the
network"

msg.attach(MIMEText(body, 'plain'))

filename = "attacks.txt"
attachment = open("attacks.txt", "rb")

part = MIMEBase('application', 'octet-stream')
part.set_payload((attachment).read())
encoders.encode_base64(part)
part.add_header('Content-Disposition', "attachment; filename= %s" % filename)

msg.attach(part)

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login(fromaddr, "onlyfortoday")
text = msg.as_string()
server.sendmail(fromaddr, toaddr, text)
server.quit()
```