TECHNICAL UNIVERSITY OF CRETE
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Distributed Real-time Network Intrusion Detection System on Apache Spark

by

Minas Diomfeas Kalosynakis

October 2021

Abstract

In recent years, the rapid increase of internet based services raises significant information security concerns. A large amount of network traffic data is generated on a daily basis with high speed while security threats become increasingly more complex. Fast and efficient detection of intrusive activities in such conditions is a challenging task. In order to address this issue, we propose a distributed intrusion detection system that utilizes machine learning classifiers to identify malicious network activity in real-time. Specifically, we use the Chi-Squared algorithm to select important features, based on which we build Decision Tree, Random Forest, and Extreme Gradient Boosting classification models on Apache Spark Big Data platform. The proposed system supports scalability in all of its different layers and provides a user-friendly graphical interface to visualize network activity. Experimental results against the NSL-KDD dataset demonstrate that the system can perform binary classification with an area under ROC curve of 97% using the Random Forest machine learning model.

Acknowledgements

First of all I would like to thank my supervisor Prof. Ioannidis Sotirios for overseeing this work and the members of the committee for taking the time to evaluate it. I would also like to express my sincere gratitude to Leonidas Kallipolitis for his guidance throughout the completion of this thesis as well as for his valuable suggestions and advice along the way. Finally, i would not have completed my work if it was not for my friends and family and their continuous support and encouragement.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The rapid increase of internet based services in recent years, leads to a significant rise of cyber security concerns. The amount of data being generated every day exceeds the level of petabytes including information about the activity of internet users. This information can be traces that they leave when they access a website, a mobile application, a network, etc. This influx of log data is caused not only by one but by multiple kinds of sources. The wise utilization of these logs is of paramount importance if we aim to maintain stable, reliable and secure computer networks.

Security incidents are becoming increasingly more complex and frequent. In the latest years, a significant percentage of today's organizations are experiencing burst attacks, based on Cisco Cybersecurity Reports [1]. These attacks are characterized by the fact that they can take place in a small time-frame and can cripple security systems within minutes. The development of real-time network traffic monitoring systems [2] [3] that can detect malicious activity, scale to the amount of data being ingested and act quickly in terms of response time can give an edge over such types of attacks.

Distributed processing platforms, such as Apache Spark [4], Apache Flink [5], and Apache Storm [6], are increasingly being used in order to ingest and process huge amounts of data in real-time with low latency. The ability to act as soon as events are generated improves an organization's responsiveness and effectiveness. Big data stream processing platforms find a good fit in a variety of industries such as:

- **Healthcare** - Real-time analytics allow clinicians to quickly get insights about their patients and enable them to save time, improve care, and acquire critical metrics. The ability to access data in real-time can dramatically aid clinicians in lifesaving decisions.
- **Fraudulent transactions** - In order to prevent fraud, the predictive model must decide if the transaction needs to be accepted or rejected, in real-time.
- **Manufacturing** - In an industry where every delay or shutdown directly impacts the financial stability of an organization, sensors are used to monitor the state of equipment in real-time. This data can also be used to optimize the product quality, supply planning, output forecasting or increase energy efficiency.

With the aid of these big data frameworks, a threat detection system can instantly identify anomalous behaviour or suspicious activities and flag them for immediate investigation while also being able to manage heavy workloads. Therefore, such technologies can prove to be a valuable ally for cybersecurity and assist network monitoring systems in overcoming current challenges.

Another matter that requires further investigation is whether or not traditional monitoring systems that analyze, filter packets, or operate based on specific rules, are sufficient for more complex security cases. In this context, a promising alternative for classifying network traffic and detecting threats is to apply Machine Learning (ML) techniques. These techniques can analyze patterns and learn from them to help prevent attacks and respond to changing behaviour. Rather than creating something to solve a problem, we create something that learns how to solve a problem.

Machine learning techniques work well in conjuction with the strong points of big data processing platforms. In a supervised learning scenario, the more samples we are able to ingest, the better we can train the classifiers to detect potential threats. Thus, machine learning can be further enhanced to provide fast, accurate, efficient, automated security systems that are able to predict threat patterns in real-time, adapt to novel malicious behaviours, and withstand huge workloads.

1.2 Objectives

The overall objective of our work is to develop a data analytics pipeline to process incoming logs and detect network anomalies in real-time. Specifically, it can be divided into separate parts:

- Handle generated logs as soon as they arrive and make them reliably available for the rest of the pipeline.
- Build a Machine learning component that processes logs in real-time and makes predictive analysis.
- Visualize the results of the monitoring process.
- Provide scalability among all components of the system so that its computational efficiency can adapt to increased load.
- Include a layer that ensures ease of deployment and helps in the orchestration and configuration of all services.

For each individual goal of our project, we have employed a different set of tools to tackle each problem accordingly.

1.3 Solution Overview

Our solution combines machine learning, distributed stream processing, virtualization, along with other tools in order to provide an accurate, scalable, real-time threat detection system that meets the current demand. To validate the proposed framework, we perform the classification of network flows of the NSL-KDD [7] dataset, a popular dataset widely used as a benchmark for modern-day internet traffic monitoring systems.

There are three main stages that form the system's pipeline. In the first stage, unlabeled testing network flows are converted into a stream to emulate a real-time traffic data source. The data is continuously passed to a distributed messaging broker, namely Apache Kafka [8], whose role is to work as a fault-tolerant substrate for the stream processing section. It stores and handles the input network connection records reliably in order to feed it to the rest of the pipeline at the rate that it arrives or at another preferable rate dictated by the processing component that follows.

In the second stage, the main processing and classification takes place. It is implemented on Apache Spark [4] framework, an analytics engine for large-scale data processing, and specifically on its Structured Streaming and MLlib APIs. Initially, a feature selection algorithm, namely Chi-Squared algorithm, extracts important features from the input data. This leads to a reduction of the computational power required, and increases the accuracy by removing irrelevant features. Subsequently, three methods of supervised classification, namely Decision Tree, Random Forest and XGBoost(eXtreme Gradient Boosting), are employed to separate the incoming flows between attacks and normal connections in real time. The models of the feature selection algorithm as well as the ML classifiers are already built from a previous training phase based on labeled time-invariable training data.

In the last stage, the storage and visualization of the classified network flows takes place. For this role we use Elasticsearch [9], a distributed search and analytics engine, to provide a fast search and storage service, and its close partner Kibana that retrieves the stored data in ElasticSearch through queries. Kibana exposes the results in real-time or in a specific time range, allowing us to monitor network traffic in a dashboard of our design.

All the aforementioned services are containerized using Docker [10]. Docker is a virtualization tool that enables a flexible, lightweight, easily scalable implementation of our system. Docker containers allow our application to run anywhere and be deployed distributively in a cluster with the only requirement being the installation of Docker on each node. In addition, Docker helps with the orchestration of all the individual components so we can launch each one in the correct order with respect to each other and to the training and testing phases of the threat detection system.

For the evaluation of our system, as well as a base for the implementation of the pipeline, we have chosen NSL-KDD dataset. NSL-KDD dataset [11] is an improved version of the popular KDD'99 dataset [12], commonly used for testing similar network monitoring systems. This dataset was used in the Third International Knowledge Discovery and Data Mining Tools Competition, which was held in conjunction with KDD-99 The Third International Conference on Knowledge Discovery and Data Mining. The competition task was to build a network intrusion detector, a predictive model capable of distinguishing between *bad* connections, called intrusions or attacks, and *good* normal connections. This database contains a standard set of data to be audited, which includes a wide variety of intrusions simulated in a military network environment.

The rest of the report is organized as follows. In Chapter 2, we provide information about each technology used in this work. Chapter 3 presents the proposed system and its components. Specifically, Section 3.1 contains detailed information about the NSL-KDD dataset as well as its improvements compared to its predecessor. In Section 3.2, we describe the architecture of the proposed system and explain the details of its functionality. The containerization of each service of the system using Docker is explained in Section 3.3 and Section 3.4 presents experimental results. Finally, Chapter 4 provides conclusions and offers new possibilities for the development of future work.

Chapter 2

Background

2.1 Big Data Processing

Today developers analyze Terabytes and Petabytes of data for different purposes. There are many projects on which they rely on to speed up their work. All of these projects are based on two aspects, batch processing and stream processing.

The distinction between batch processing and stream processing is one of the most fundamental principles within the big data world. When most people use these terms, they usually mean the following:

- **Batch processing model**: A set of data is collected over time, then fed into an analytics system. In other words, you collect a batch of information, then send it in for processing.
- **Stream processing model**: Data is fed into analytics tools piece-by-piece. The processing is usually done in real time.

2.1.1 Batch Processing

In batch processing, blocks of data that have been stored over a period of time, are scheduled to be processed. For example, processing all the transactions that have been performed by a major financial firm in a week. This data contains millions of records corresponding to each day and are stored as files. These files undergo processing at the end of the day for various analyses that lead to useful insights and conclusions for the firm.

Batch processing works well in situations where you do not need real-time analytics results, and when it is more important to process large volumes of information than it is to get fast analytics results. To generalize, we should lean towards batch processing when:

- We are working with large datasets and are running a complex algorithm that requires access to the entire batch e.g. sorting the entire dataset.
- We get access to the data in batches rather than in streams.
- We are joining tables in relational databases.

2.1.2 Stream Processing

Stream processing requires the ingestion of a sequence of data, also known as data streams. Data streaming is the process of sending data records continuously rather than in batches. The data is generated by multiple sources simultaneously, and in small sizes. Stream processing allows you to analyze data in real-time and gives you insights into a wide range of activities, such as server activity, geolocation of devices, or user activity on a website.

Stream processing is key if we need analytics results in real-time. It is ideally suited to data that has no beginning or end and is optimal for time series and detecting patterns over time. It is often used for real-time aggregation, correlation, filtering of data, or for incremental updates on metrics, reports, and summary statistics in response to each arriving data record. Indications that stream processing is the right approach are:

- Data is generated in a continuous manner.
- Low latency is crucial.

2.2 Apache Spark

Apache Spark [13] is a powerful big data processing platform which follows a hybrid approach. As a hybrid framework, Spark offers support for both batch and stream processing capabilities. It started as a research project at the UC Berkeley AMPLab in 2009, grew into a broad developer community, and moved to the Apache Software Foundation in 2013.

Even though Spark uses many similar principles to Hadoops MapReduce engine, Spark outperforms the latter in terms of performance. For instance, given the same batch processing workload, Spark can be faster due to the full in-memory computation feature compared to the traditional read and write to the disk approach of MapReduce.

1) Spark Batch Processing Model: The strongest advantage of Spark over MapReduce is the in-memory computation. Spark interacts with the disk only for two tasks: loading the data initially into the memory and storing the final results back to the disk. All other results in-between are processed in-memory. This in-memory processing makes Spark significantly faster that its batch processing competitor, the Hadoop framework. To support the in-memory computation feature, Spark uses Resilient Distributed Datasets (RDD). RDD is a read-only data structure maintained in memory to make Spark a fault tolerance framework without having to write to the disk after every operation.

2) Spark Stream Processing Model: In addition to batch processing, Spark provides stream processing abilities with the use of micro-batches. Micro-batching data streams are treated as a group of very small batches which are in turn handled as a regular task by Spark batch engine. Even though this micro-batching procedure works well, it could still lead to some differences in terms of performance as opposed to true stream processing frameworks.

Spark provides high-level APIs in Java, Scala, Python, R, and an optimized engine that supports general execution graphs. Specifically, these APIs are:

- Spark SQL for structured data processing.

- MLlib for machine learning.
- GraphX for graph processing.
- Spark (Structured) Streaming for incremental computation and stream processing.

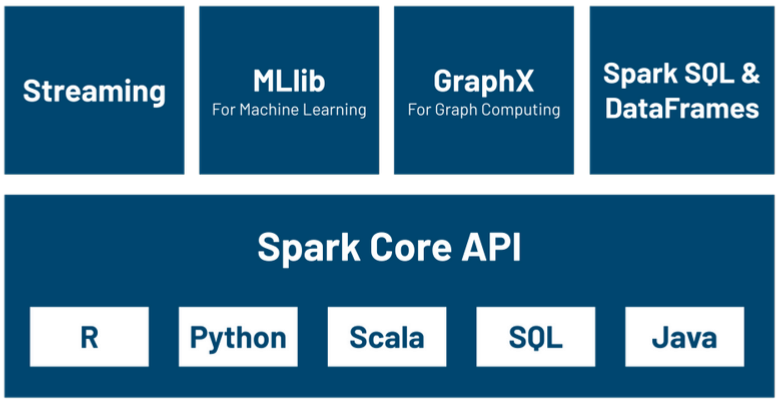


Figure 2.1: Apache Spark Ecosystem

2.2.1 Spark Core

All the functionalities being provided by Apache Spark are built on the top of Spark Core. It delivers speed by providing in-memory computation capability. Spark Core is the the base engine for parallel and distributed processing of huge datasets. Other key features of Apache Spark Core are:

- Interaction with storage systems.
- Scheduling and monitoring jobs on a cluster.
- Distributed task dispatching.
- Memory management and fault recovery.

Resilient Distributed Dataset (RDD)

Spark Core is embedded with a special collection called RDD (resilient distributed dataset). RDDs are immutable distributed collections of elements of data that can be stored in memory or disk across a cluster of nodes [14]. The data is partitioned across nodes in a cluster that can be operated in parallel with a low-level API.

RDDs can only be created through deterministic operations on data that reside in stable storage or on other RDDs. They do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage.

We can control two aspects of RDDs: persistence and partitioning. We can indicate which RDDs we will reuse and choose a storage strategy that best fits our needs such as in-memory storage. We can also set the partitioning method of RDDs across the cluster. For example, we can partition based on a key in each record. This is useful for placement optimizations, such as ensuring that two datasets that will be joined together are hash-partitioned in the same way.

There are two operations performed on RDDs: Transformations and Actions.

- Transformation: It is a function that produces new RDD from the existing RDDs. It takes an RDD as input and produces one or more RDD as output. Applying transformations, builds an RDD lineage. RDD lineage, also known as RDD operator graph, is a logical execution plan i.e., a Directed Acyclic Graph(DAG) of its parent RDDS and their transformations. All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently.

- Action: Transformations create RDDs from each other, but when we want to work with the actual dataset, we need to call an

Action. An Action is a Spark RDD operation that gives non-RDD values. The values of Actions are returned to the driver program after running a computation. They bring the laziness of RDDs into motion.

2.2.2 Spark SQL

Spark SQL is a module in Apache Spark that integrates relational processing with Spark's functional programming API. Spark SQL lets Spark programmers leverage the benefits of relational processing (e.g. declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark [15].

Spark SQL also includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points.

DataFrame

The main abstraction in Spark SQL's API is a DataFrame, an immutable distributed collection of rows with the same schema. A DataFrame is equivalent to a table in a relational database, and can also be manipulated in similar ways to the native distributed collections in Spark, the RDDs. Unlike RDDs, DataFrames keep track of their schema and support various relational operations that lead to more optimized execution.

DataFrames can be constructed from tables in a system catalog (based on external data sources) or from existing RDDs. Once constructed, they can be manipulated with various relational operators, such as `where` and `groupBy`, which take expressions in a domain-specific language. Each DataFrame can also be viewed as an RDD of Row objects, allowing users to call procedural Spark APIs such as `map`.

Spark DataFrames are lazy, in that each DataFrame object represents a logical plan to compute a dataset, but no execution occurs until the user calls an action operation such as `save`.

2.2.3 Spark Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine [16]. The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended. This leads to a stream processing model that is similar to a batch processing model. We can design our streaming computations as standard batch-like queries, and Spark runs them as incremental queries on the unbounded input table.

Consider the input data stream as the "Input Table". Every data item that arrives, is appended to the Input Table as new row.

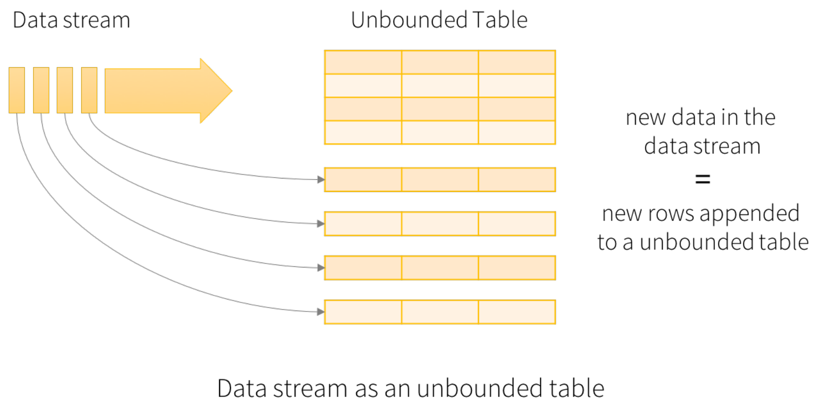

Data stream
Unbounded Table
new data in the data stream
=
new rows appended to a unbounded table
Data stream as an unbounded table

Figure 2.2: Data stream in Spark

The computations are executed on the same optimized Spark SQL engine. Finally, the system ensures exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs. In short, Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing and abstracts away the intricacies of stream processing from the user.

2.2.4 Spark MLlib

MLlib is Spark's machine learning (ML) library [17]. Its goal is to make practical machine learning scalable and easy. At a high level, the MLlib API provides tools such as:

- Implementation of common ML Algorithms such as classification, regression, clustering, and collaborative filtering.
- Feature extraction methods, transformation, dimensionality reduction, and feature selection.
- Tools for constructing, evaluating, and tuning ML Pipelines.
- Persistence through saving and loading built models, and Pipelines.

2.2.5 Spark Architecture

As illustrated in figure 2.3, the architecture of Apache Spark consists of a master node which runs a driver program that is in charge of calling the main program of an application. The driver program is the code written by the user. This driver program is responsible for creating Spark context. Spark context behaves like a gateway to all of the functionalities of Apache Spark. Both Spark context and the driver collectively handle the execution of the job within the cluster.

The cluster manager first takes care of the resource allocation. Then, the job is split into numerous tasks that are assigned to the worker nodes. The worker nodes execute the tasks that are assigned to them by the manager and return the results back to the spark context. The executors carry out the execution of the individual tasks. In order to increase the performance of the system, the number of worker nodes must be increased so that the computations can be divided further into more logical portions.

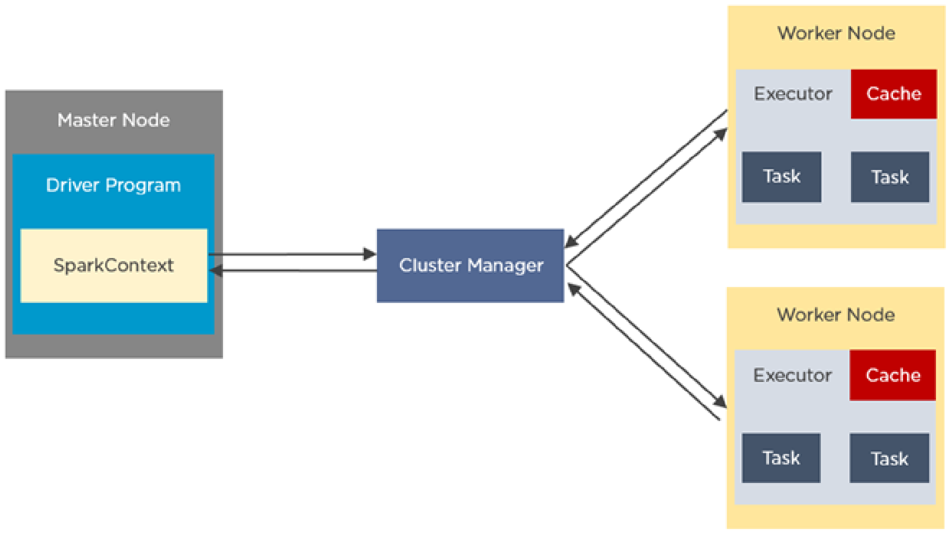


Figure 2.3: Spark Architecture

2.3 Machine Learning

Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behavior. Artificial intelligence systems are used to perform complex tasks in a way that is similar to how humans solve problems.

The goal of AI is to create computer models that exhibit "intelligent behaviors" like humans. This means models that can recognize a visual scene, understand a text written in natural language, or perform an action in the physical world. Traditional programming requires creating detailed instructions for the computer to follow. But in some cases, writing a program for the machine to follow is time-consuming or impossible, such as training a computer to recognize pictures of different people. While this is an easy task for humans to complete, it is difficult to instruct a computer how to do it through traditional programming. Machine learning takes the approach of letting computers learn to program themselves through experience.

The function of a machine learning system can be:

- **Descriptive**: The system uses the data to explain what happened.
- **Predictive**: The system uses the data to predict what will happen.
- **Prescriptive**: The system uses the data to make suggestions about what action we should take.

Broadly speaking, machine learning approaches are divided into three types: supervised learning, unsupervised learning, and reinforcement learning. In unsupervised machine learning, a program looks for patterns in unlabeled data. Unsupervised machine learning can find patterns or trends that people are not explicitly looking for. For example, an unsupervised machine learning program could look through online sales data and identify different types of clients making purchases. Reinforcement machine learning trains machines through trial and error to take the best action by establishing a reward system. Reinforcement learning can train models to play games or train autonomous vehicles to drive by telling the machine when it made the right decisions, which helps it learn over time what actions it should take.

2.3.1 Supervised Learning

Supervised machine learning models are trained with labeled data sets, which allow the models to learn and become more accurate over time. For example, an algorithm would be trained with pictures of dogs and other things, all labeled by humans, and the machine would learn ways to identify pictures of dogs on its own. Supervised machine learning is a common machine learning approach used today and is the method of learning that we incorporated in this work.

Decision Tree

Decision Tree is a popular classification algorithm and quite straightforward to understand and interpret. Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, the Decision Tree algorithm can be used for solving regression and classification problems too.

The goal of using a Decision Tree is to create a training model that can be used to predict the class or value of the target variable by learning simple decision rules inferred from prior data(training data). In Decision Trees, for predicting a class label for a record we start from the root of the tree. We compare the values of the root attribute with the record's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

Important terminology related to Decision Trees:

1. Root Node: It represents the entire population or sample and this further gets divided into two or more homogeneous sets.

2. Splitting: It is a process of dividing a node into two or more sub-nodes.

3. Decision Node: When a sub-node splits into further sub-nodes, then it is called the decision node.

4. Leaf / Terminal Node: Nodes do not split is called Leaf or Terminal node.

5. Pruning: When we remove sub-nodes of a decision node, this process is called pruning.

6. Branch / Sub-Tree: A subsection of the entire tree is called branch or sub-tree.

7. Parent and Child Node: A node, which is divided into sub-nodes is called a parent node of sub-nodes whereas sub-nodes are the child of a parent node.

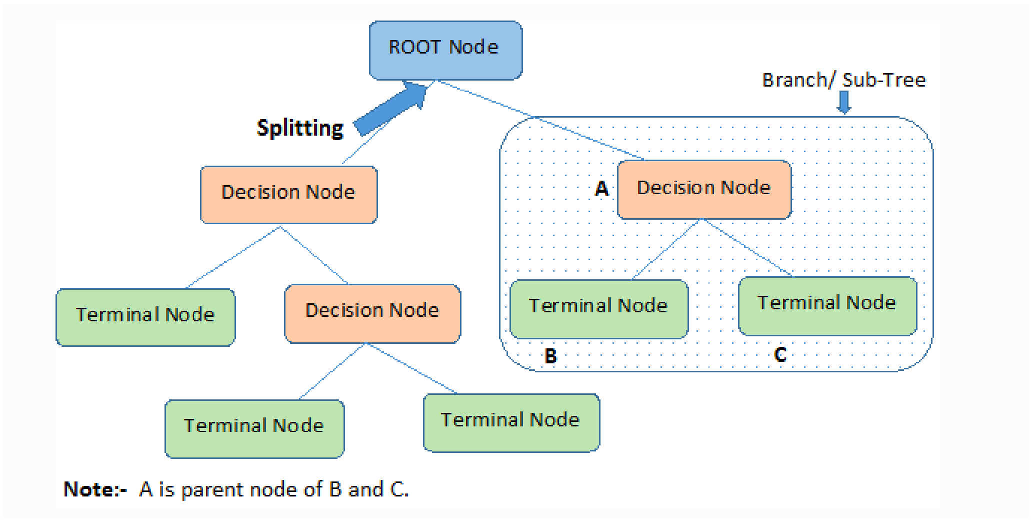


Figure 2.4: Decision Tree depiction

Decision Trees classify the examples by sorting them down the tree from the root to some leaf/terminal node, with the leaf/terminal node providing the classification of the example. Each node in the tree acts as a test case for some attribute, and each edge descending from the node corresponds to the possible answers to the test case. This process is recursive in nature and is repeated for every subtree of the new node.

Random Forest

Random Forest is another supervised learning algorithm. The forest it builds, is an ensemble of decision trees, usually trained with the "bagging" method. The general idea of the bagging method is that a combination of learning models increases the overall result.

Essentially, Random Forest builds multiple decision trees and merges them together to get a more accurate and stable prediction. One big advantage of Random Forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. Below is an image with an example of a Random Forest with two trees.

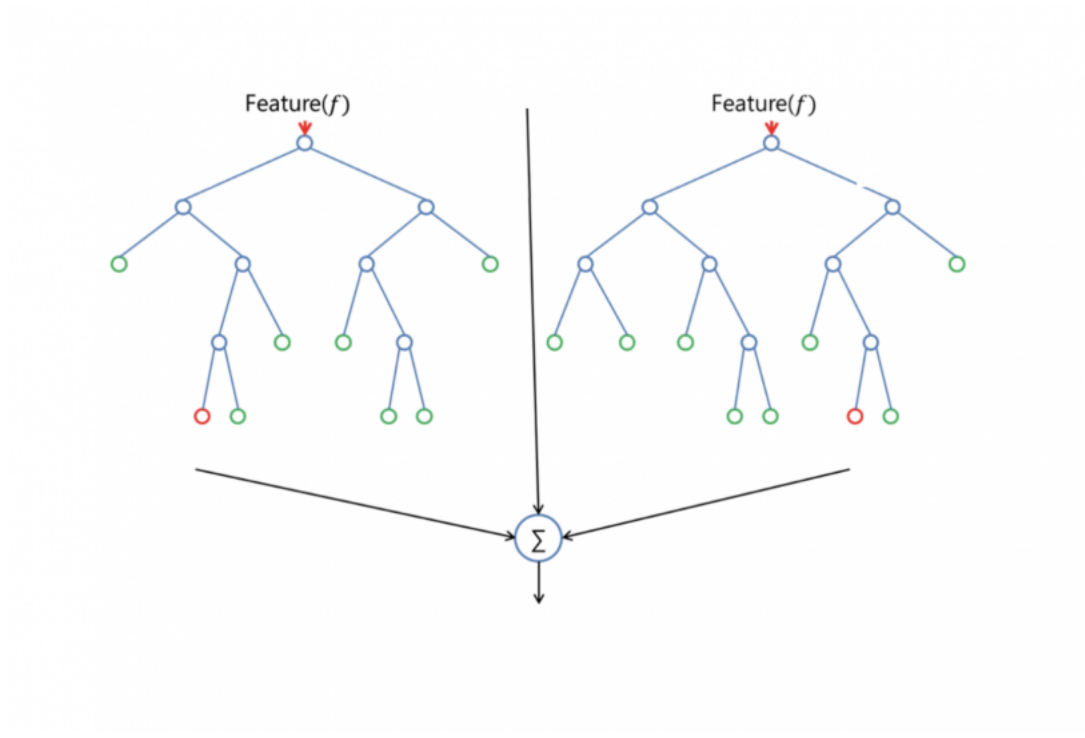


Figure 2.5: Random Forest depiction

Random Forest introduces additional randomness to the model, while growing the trees. Instead of searching for the most important feature while splitting a node, it searches for the best feature among a random subset of features. This results in a wide diversity that generally results in a better model. Therefore, only a random subset of the features is taken into consideration by the algorithm on splitting a node.

Extreme Gradient Boosting

Extreme Gradient Boosting, or otherwise, XGBoost is a specific implementation of the Gradient Boosting method. Like Random Forest, Gradient Boosting is another technique for performing supervised machine learning tasks, like classification and regression. XGBoost is particularly popular because it has been the winning algorithm in a number of recent machine learning competitions.

Similar to Random Forests, Gradient Boosting is an ensemble algorithm. This means it creates a final model based on a collection of individual models. The predictive power of these individual models is weak and prone to overfitting but combining such weak models in an ensemble leads to an overall improved result. In Gradient Boosting machines, the most common type of model used is Decision Trees - another parallel to Random Forests.

Boosting builds models from individual so called weak learners in an iterative way. In boosting, the individual models are not built on completely random subsets of data and features but sequentially by putting more weight on instances with wrong predictions and high errors. The general idea behind this is that instances, which are hard to predict correctly ("difficult" cases) will be focused on during learning, so that the model learns from past mistakes.

XGBoost, compared to other gradient boosting methods, uses more accurate approximations to find the best tree model. While regular gradient boosting uses the loss function of the base model (e.g. decision tree) as a proxy for minimizing the error of the overall model, XGBoost uses the 2nd order derivative as an approximation. It also utilizes advanced regularization, which improves model generalization. Additional advantages of XGBoost are that the training is very fast and can be distributed across clusters.

2.4 Docker

Docker is an open source software platform to create, deploy and manage virtualized application containers on different operating systems.

Docker packages, provisions and runs containers. Container technology is available through the operating system. A container packages the application service or function with all of the libraries, configuration files, dependencies, and other necessary parts required to operate. Each container shares the services of the underlying operating system.

Docker uses resource isolation in the OS kernel to run multiple containers on the same OS. This is different than virtual machines (VMs), which encapsulate an entire OS with executable code on top of an abstracted layer of physical hardware resources.

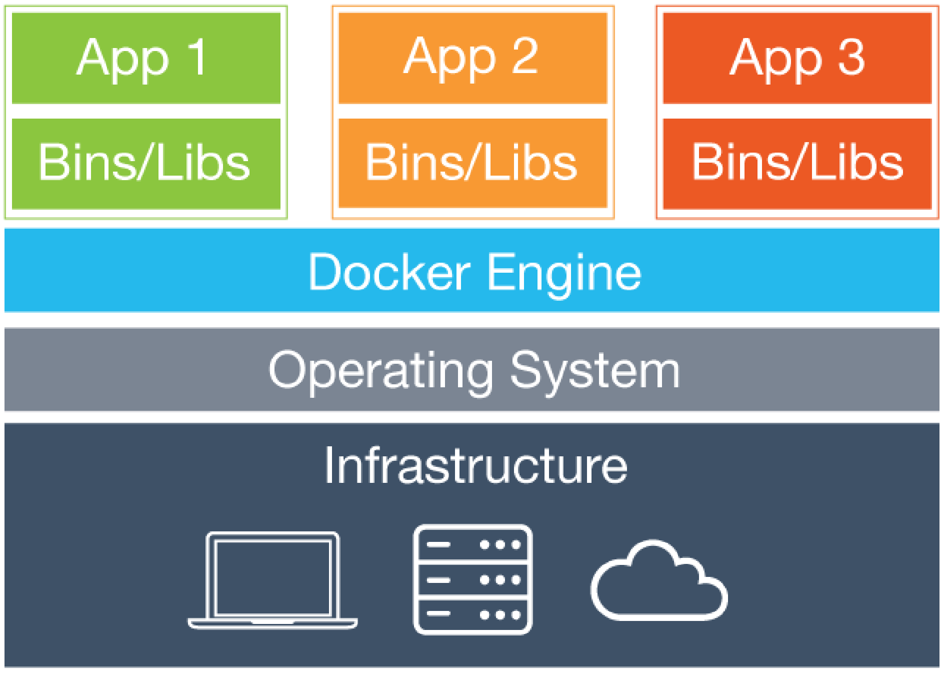


Figure 2.6: Docker Stack

Each container starts with a **Dockerfile**. This text file provides a set of instructions to build a Docker image, including the operating system, languages, environmental variables, file locations, network ports, and any other components it needs to run.

A **Docker image** is a portable, read-only, executable file containing instructions for creating a container and specifications based on which, software components of the container will run and how.

The **Docker Engine** is the underlying technology that handles the tasks and workflows involved in building container-based applications. The engine creates a server-side daemon process that hosts images, containers, networks and storage volumes. The daemon also provides a client-side command-line interface (CLI) for users to interact with the daemon through the Docker application programming interface.

Docker Swarm is a mode in Docker Engine that supports cluster load balancing for Docker. Multiple Docker host resources are pooled together to act as one, which enables users to quickly scale up container deployments to multiple hosts.

Docker Compose is a command-line tool to define and run multi-container Docker applications. It allows you to create, start, stop, rebuild all the services from your configuration, view the status and log output of all running services.

2.5 Apache Kafka

Apache Kafka is an open source project for a distributed publish-subscribe messaging system rethought as a distributed commit log. Kafka stores messages in topics that are partitioned and replicated across multiple brokers in a cluster. Kafka provides three main functions to its users:

- Publish and subscribe to topics.
- Reliably store streams of records.
- Process streams of records in real-time.

Kafka is primarily used to build real-time streaming data pipelines and applications that adapt to the data streams. It combines messaging, storage, and stream processing to allow storage and analysis of both historical and real-time data.

Commit Log

The Kafka commit log provides a persistent ordered data structure. Records cannot be directly deleted or modified, only appended onto the log. The order of items in Kafka logs is guaranteed. The Kafka cluster creates and updates a partitioned commit log for each topic that exists. All messages sent to the same partition are stored in the order that they arrive. Because of this, the sequence of the records within this commit log structure is ordered and immutable. Kafka also assigns each record a unique sequential ID known as an offset, which is used to retrieve data.

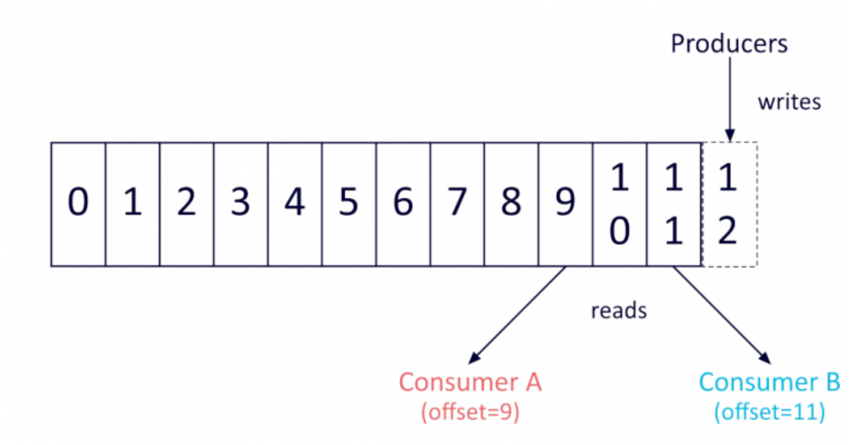


Figure 2.7: Commit Log

Kafka APIs

Apache Kafka offers four key APIs: the Producer API, Consumer API, Streams API, and Connector API.

- **Producer API**: used to publish a stream of records to a Kafka topic.
- **Consumer API**: used to subscribe to topics and process their streams of records.
- **Streams API**: enables applications to behave as stream processors, which take in an input stream from topic(s) and transform it to an output stream which goes into different output topic(s).
- **Connector API**: allows users to seamlessly automate the addition of another application or data system to their current Kafka topics.

A **Kafka topic** defines a channel through which data is streamed. Producers publish messages to topics, and consumers read messages from the topic they subscribe to. Topics organize and structure messages, with particular types of messages published to particular topics. Topics are identified by unique names within a Kafka cluster, and there is no limit on the number of topics that can be created.

How it works

Applications (producers) send messages (records) to a Kafka node (broker) to be processed by other applications called consumers. These records get stored in a topic and consumers subscribe to the topic to receive new messages.

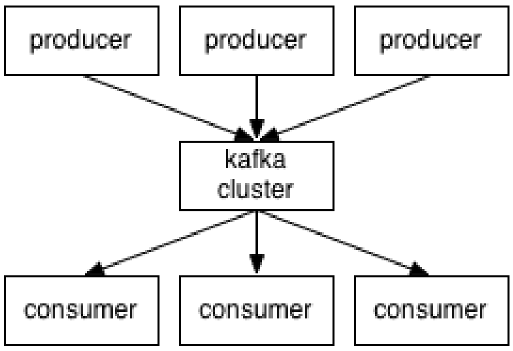


Figure 2.8: Kafka Functionality

As topics can get quite big, they get split into partitions of a smaller size for better performance and scalability. Kafka guarantees that all messages inside a partition are ordered in the sequence they came in.

2.6 Elasticsearch and Kibana

Elasticsearch and Kibana are two of the three main components of the ELK Stack. The ELK Stack is a popular log management platform. It is designed to provide analytics solutions by monitoring modern applications and the infrastructure they are deployed on.

2.6.1 Elasticsearch

Elasticsearch is a distributed, open-source search engine built on Apache Lucene and developed in Java. It started as a scalable version of the Lucene open-source search framework and added the ability to horizontally scale Lucene indices. Elasticsearch allows you to store, search, and analyze huge volumes of data quickly and in near real-time and give back answers in milliseconds. It is able to achieve fast search responses because instead of searching the text directly, it searches an index. It uses a structure based on documents instead of traditional relational tables. Elasticsearch comes with extensive REST APIs for storing and searching the data.

Document

Documents are the basic unit of information that can be indexed in Elasticsearch. They are expressed in JSON format, which is the global internet data interchange format. A document is similar to a row of a relational database, representing a given entity. In Elasticsearch, a document can be more than just text, it can be any structured data encoded in JSON. That data can be things like numbers, strings, and dates. Each document has a unique ID and a given data type, which describes what kind of entity the document is.

Index

An index is a collection of documents that have similar characteristics. It is the highest level entity that you can query against in Elasticsearch. You can think of the index as being similar to a database in a relational database schema. Any documents in an index are typically logically related.

Cluster

An Elasticsearch cluster is a group of one or more node instances that are connected together. The power of an Elasticsearch cluster lies in the distribution of tasks, searching, and indexing across all the nodes in the cluster.

Node

A node is a single server that is a part of a cluster. A node stores data and participates in the cluster's indexing and search capabilities. An

Elasticsearch node can be configured in different ways:

- **Master Node**: Controls the Elasticsearch cluster and is responsible for all cluster-wide operations like creating or deleting an index as well as adding or removing nodes.
- **Data Node**: Stores data and executes data-related operations such as search and aggregation.
- **Client Node**: Forwards cluster requests to the master node and data-related requests to data nodes.

Shards

Elasticsearch provides the ability to subdivide the index into multiple pieces called shards. Each shard is in itself a fully-functional and independent index that can be hosted on any node within a cluster. By distributing the documents in an index across multiple shards, and those shards across multiple nodes, Elasticsearch ensures redundancy. Redundancy protects against hardware failures and increases query performance as nodes are added to a cluster.

Replicas

Elasticsearch allows us to make one or more copies of our index's shards which are called replica shards or just replicas. Basically, a replica shard is a copy of a primary shard. Each document in an index belongs to one primary shard. Replicas provide redundant copies of your data to protect against hardware failure and increase capacity to serve read requests like searching or retrieving a document.

2.6.2 Kibana

Kibana is a data visualization and management tool for Elasticsearch that provides real-time histograms, line graphs, pie charts, and maps. It enables us to navigate and give shape to Elasticsearch data. With Kibana, we can:

- Search and observe: From discovering documents to analyzing logs, Kibana is a portal for accessing this capabilities and more.
- Visualize and analyze data: Search for hidden insights, visualize them in charts, maps, line graphs, and combine them in a dashboard.
- Manage and monitor the cluster: Manage indices, ingest pipelines, monitor the health of the Elasticsearch cluster, and control users permissions.

However, a major drawback is that every visualization can only work against a single index pattern. So, if we have indices with strictly different data, we have to create separate visualizations.

Chapter 3

Network Intrusion Detection Application

In this chapter we describe the architecture of our network intrusion detection system and present the details of its implementation and functionality.

The backbone of the system is the use of Apache Spark framework as the main distributed processing core. We chose Spark and its Structured Streaming API over other stream-processing platforms. This decision was made due to the ease of implementation that Spark provides, its native and community-wide support for using machine learning algorithms with MLlib API, and because it presents the best fault tolerance performance [18] compared to Apache Storm or Apache Flink. Spark ensures end-to-end exactly-once semantics under any failure, making our application robust. Spark runs in a cluster following the master/worker model, where workers can expand and reduce resources, providing scalability to the system.

We also decided to develop our application using the Scala programming language instead of Java or Python. Since Spark is written in Scala, it is faster in processing data and offers better user APIs. Python and Bash have also been employed to create scripts for specific purposes along the way as will be explained later on.

Additionally, Docker, Apache Kafka, Elasticsearch, and Kibana were clear choices for the roles that they were called to fulfill. Docker enables us to package our system with a virtualization layer so that it can run regardless of the underlying infrastructure. Apache Kafka acts as a reliable messaging service that handles the input streaming data. Elasticsearch and kibana allow us to store and visualize results in real-time. Also, all of these techonologies contribute to the intended distributed and scalable nature of our threat detection system.

The proposed system is available as an opensource solution that can be found at the corresponding repository along with instructions on how to deploy and use it.

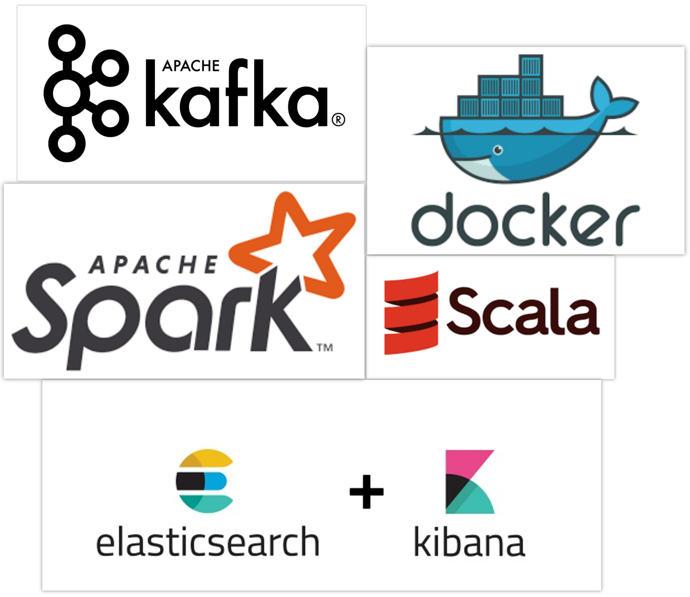


Figure 3.1: Technologies used in this work.

3.1 Dataset Selection

Intelligent intrusion detection systems can only be built if there is availability of an effective dataset. The internet traffic record data that we selected to evaluate our intrusion detection system and base our implementation is one of the most common datasets used as a benchmark for modern-day internet traffic monitoring tools. NSL-KDD dataset [19] is a refined, cleaned-up version of the KDD'99 dataset.

The dataset contains 43 features per record, with 41 of the features referring to the traffic input itself while the last two are labels describing whether it is a normal connection or an attack, and the severity of the traffic input. There are four different classes of attacks within the dataset: Denial of Service (DoS), Probe, User to Root(U2R), and Remote to Local (R2L). A brief description of each attack can be seen below:

- DoS is an attack meant to shut down a machine or network, making it inaccessible to its intended users. DoS attacks accomplish this by flooding the target with traffic, or sending it information that triggers a crash. In both instances, the attack deprives legitimate users (i.e. employees, members, or account holders) of the service or resource they expected. DoS attacks often target web servers of high-profile organizations such as banking, commerce, and media companies, or government and trade organizations [20].

- Probing is a type of attack in which the intruder scans network devices to determine weaknesses in topology design or some opened ports and then use them in the future for illegal access to personal information [21].

- A U2R attack occurs when an attacker who has already achieved user access on a system, tries to gain privileged access. Various buffer overflow attacks against network services fall in this category.

- An R2L attack occurs when an attacker who has the ability to send packets to a machine over a network but who does not have an account on that machine, exploits some vulnerability to gain local access as a user of that machine.

Although these attacks exist in the dataset, the distribution is heavily skewed. Essentially, more than half of the records that exist in each dataset are normal traffic, and the distributions of U2R and R2L are extremely low. Despite its bias, this is an accurate representation of the distribution of modern-day internet traffic attacks, where the most common attack is DoS while U2R and R2L are hardly ever seen.

Breakdown of features

A connection record of this dataset summarizes the packets of a communication session between a connection initiator with a specified source IP address and a destination IPaddress over a pair of TCP/UDP ports [22]. These traffic records can be broken down into four categories: Basic, Content-based, Host-based, and Time-based.

1. Basic features can be derived from the header of the packet without looking into the payload itself, and hold the basic information about the packet such as the protocol, service, and duration of a connection. This category contains the first up until the ninth feature.

2. The content-based features hold information about the content, such as the login activity. With this information, the system can access the payload. This category contains features from number 10 to 22.

3. Time-based features hold the analysis of the traffic input over a two-second window and contain information about connections related to the same host. These features are mostly counts and rates rather than information about the content of the traffic input. This category contains features 23–31.

4. Host-based features are similar to Time-based features, except instead of analyzing over a two-second window, they are designed to describe attacks, which span longer than a two-second window. This category contains features 32–41.

There are four types of features in this dataset:

- Categorical: Text type data that generally take a limited number of possible values.
- Binary: Discrete data that can be in only one of two categories.
- Discrete: Numeric variables that have a finite number of values within a specific range.
- Continuous: Numeric variables that have an infinite number of values within a specific range.

A subset of the total features and their meaning is shown in the table below:

#	Feature Name	Description	Type
1	Duration	Length of time duration of the connection	Continuous
2	Protocol Type	Protocol used in the connection	Categorical
3	Service	Destination network service used	Categorical
4	Flag	Status of the connection – Normal or Error	Categorical
5	Src Bytes	Number of data bytes transferred from source to destination in single connection	Continuous
6	Dst Bytes	Number of data bytes transferred from destination to source in single connection	Continuous
7	Land	If source and destination IP addresses and port numbers are equal then, this variable takes value 1 else 0	Binary
8	Wrong Fragment	Total number of wrong fragments in this connection	Discrete
9	Urgent	Number of urgent packets in this connection. Urgent packets are packets with the urgent bit activated	Discrete
10	Hot	Number of "hot" indicators in the content such as: entering a system directory, creating programs and executing programs	Continuous

Improvements on the original KDD'99 dataset

NSL-KDD is a dataset suggested to solve some of the inherent problems of the KDD'99 [23]. The advantages over the original set are:

- No redundant records in the train set, so the classifiers will not produce any biased result.
- There is no duplicate records in the proposed test set, therefore, the performance of the learners is not biased towards the methods which have better detection rates on the frequent records.

- The number of selected records from each difficulty level group is inversely proportional to the percentage of records in the original KDD dataset. As a result, the classification rates of distinct machine learning methods vary in a wider range, which makes it more efficient to have an accurate evaluation of different learning techniques.

Despite its improvements, this new version of the KDD dataset still suffers from some of the problems of the original set and may not be a perfect representative of existing real networks. However, it can, certainly, still be applied as an effective benchmark for different intrusion detection methods.

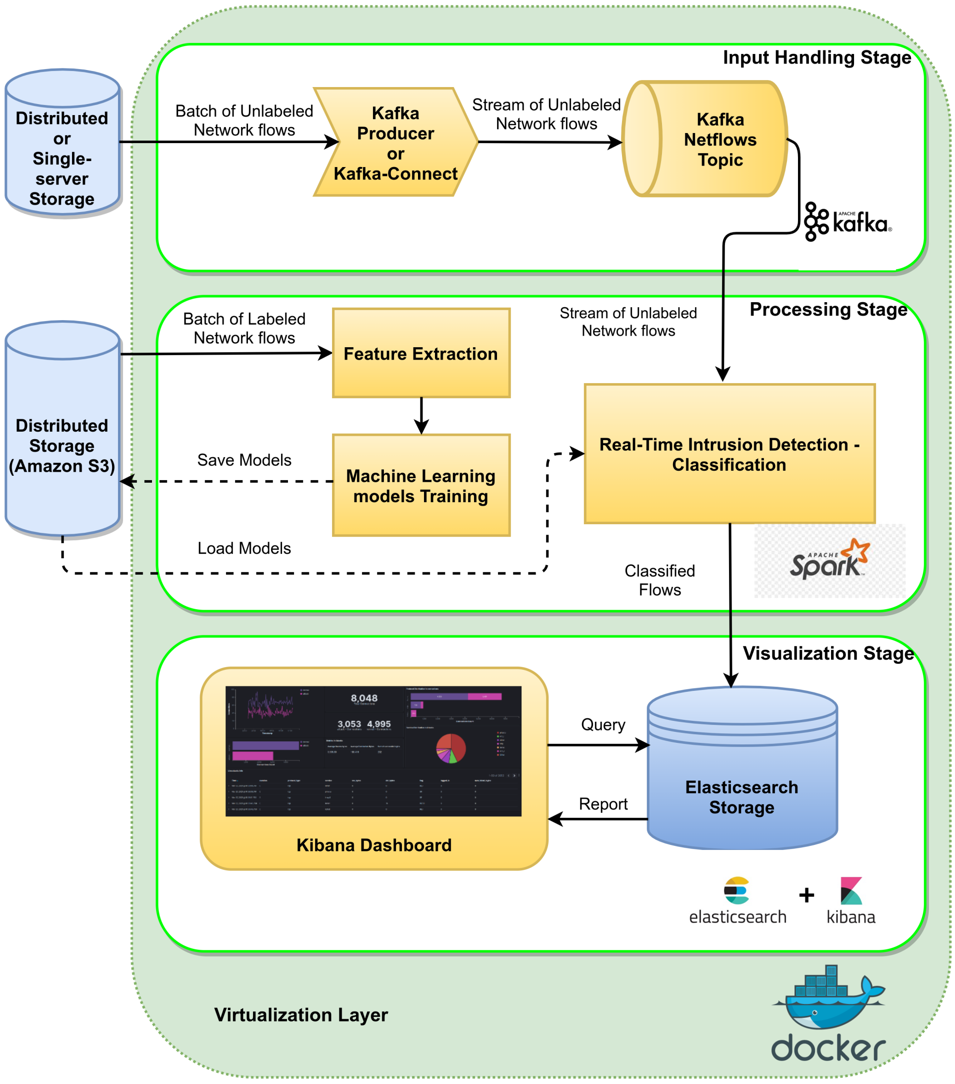
3.2 System Architecture

The functionality of the system follows a Supervised Learning approach and consists of two phases. In the Offline preprocessing phase, the training dataset is inserted into the system in order to build the required feature extraction and machine learning models.In the Online phase, the pre-built models are loaded in order to classify the incoming network traffic in real-time by labeling each flow as an attack or as a normal connection based on the algorithm's prediction. The flow of data, mainly during the Online phase, forms a pipeline which is essentially split into three stages. The architecture of the system as a whole and its different stages can be viewed in figure 3.2.

1. **Input Handling Stage**: In this stage, Apache Kafka, as a distributed messaging queue, is responsible for ingesting the incoming streaming network data. This data is stored in a Kafka topic that is created during the launch of our system. A Kafka console producer sends connection logs of the NSL-KDD testing set into the partitions of the topic at random intervals in order to test the performance of the system. In a real world scenario, a Kafka Connector named Spooldir is constantly watching a directory for new network traffic logs and immediately sends them to the Kafka topic.

2. **Processing Stage**: Apache Spark is the processing backbone of the application. It is in charge of all data operations within the Processing Stage. The two phases of the system, Offline and Online, are implemented as two separate Spark jobs that are submitted to the Spark cluster consecutively. The first to run is the Offline phase that produces and saves the required models. As soon as the job finishes, the Online Spark job gets submitted to the cluster, loads the pre-built models, starts "listening" to the Kafka topic for new network logs and passes them through the trained ML models to label them as malicious or benign connections.

3. **Visualization Stage**: The classified records are stored in the Elasticsearch search engine through a direct Spark-Elasticsearch connection. Kibana queries the search engine, receives the response and presents it in a user-friendly dashboard to provide insights and various metrics of the network traffic in near real-time or in a specific time range, based on user's choice.



Input Handling Stage
Distributed or Single-server Storage
Batch of Unlabeled Network flows
Kafka Producer or Kafka-Connect
Stream of Unlabeled Network flows
Kafka Netflows Topic
kafka
Processing Stage
Batch of Labeled Network flows
Feature Extraction
Stream of Unlabeled Network flows
Distributed Storage (Amazon S3)
Save Models
Machine Learning models Training
Real-Time Intrusion Detection - Classification
Load Models
Classified Flows
Spark
Visualization Stage
Query
Report
Kibana Dashboard
Elasticsearch Storage
elasticsearch
kibana
Virtualization Layer
docker

Figure 3.2: Architecture Overview of the Network Intrusion Detection System

The individual components of the system run in Docker containers and with Docker Compose, we deploy them to all the available machines of the Docker Swarm cluster. The virtualization layer added by employing Docker enables us to easily deploy each service. It also allows us to configure the scalability level of each part of the system by simply, tuning the parameters in the running scripts.

Docker makes it possible for a user to deploy the system regardless of the underlying infrastructure. The only requirement is the installation of Docker (instead of all the individual technologies and dependencies used) and the assignment of Swarm-Labels on each machine, based on our preferences, so that Docker knows what services to deploy on each node.

3.2.1 Offline Phase

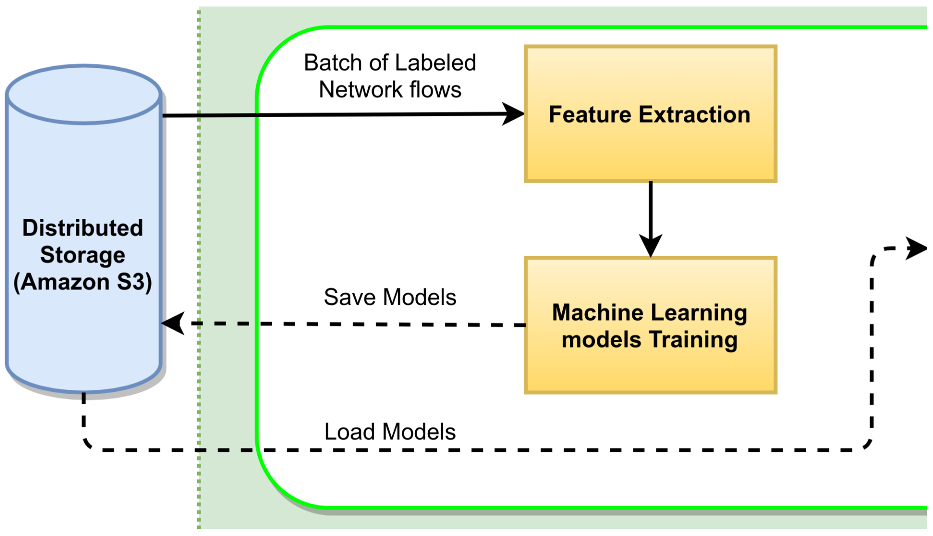

Distributed Storage (Amazon S3)
Batch of Labeled Network flows
Feature Extraction
Save Models
Machine Learning models Training
Load Models

Figure 3.3: Offline Phase

We have to note that the Offline phase or, in other words, the training of the Machine Learning and Feature extraction models, takes place entirely within the Processing Stage of the pipeline as shown in figure 3.3 and therefore uses only Apache Spark for its batch processing. A more detailed flowchart with all data processes of this phase can be viewed in figure 3.4 at the end of this section.

At launch of the application in a cluster setup, the training network flows of the NSL-KDD dataset are stored in the form of CSV files in a distributed storage supported by Hadoop such as HDFS, Cassandra, HBase, Amazon S3, etc. Due to the ease of use that it provides, we employed Amazon S3 as a distributed storage service in order to conduct our experiments.

As a first step, the CSV training file is read from an Amazon S3 bucket by Spark Context into a Spark Dataframe with an already designed data schema that corresponds to the fields of the KDD dataset. We, then, clean the dataframe by removing the unnecessary Score column and possible empty rows or rows containing null values. An extra field is added at this point with the function `categorizeKdd2Labels` that describes the connection as attack or normal connection based on the already existing class label of the dataset. NSL-KDD contains a variety of attacks but since we perform binary classification we need to categorize all connections to only two classes. Our initial goal was to perform multi-class classification but due to dataset-specific issues that are explained in Experimental Results, we resorted to binary.

Before walking through the next steps let us go briefly through the concepts of Spark ML pipelines. ML pipelines are a set of high level APIs built on top of the DataFrames which make it easier to combine multiple algorithms into a single process. The main elements of a pipeline are the transformer and the estimator. The first can represent an algorithm that can transform a DataFrame into another DataFrame, and the latter is an algorithm that can fit on a DataFrame to produce a transformer.

As a first transformer we use the `StringIndexer()` to encode the categorical columns from String format to Numeric indices since the following algorithms require numeric features. The feature extraction method that we employed is only applicable to categorical data so, after the StringIndexer step, we need to discretize the continuous features before we apply the algorithm. For this purpose, we use Spark ML's `QuantileDiscretizer()` transformer which bucketizes the continuous features and turns them into binned categorical ones. Afterwards, we assemble the prepared columns into a single feature vector that is added to the dataframe as a new column using `VectorAssembler()` such as in the example:

Feat1	Feat2	Feat3	...	Feat41
18	1	0.0	...	4.0

$\rightarrow$

Feature Vector
[18.0, 1.0, 0.0, ..., 4.0]

We can now perform feature extraction on the assembled dataframe.

Feature Extraction

Feature Extraction is a machine learning technique which reduces the amount of data to be analyzed by decreasing the dimensionality of a dataset to include only important features and discard others that do not add much value to the learning process or contain meaningless noise [24]. Choosing relevant features out of the 41 features of our dataset is a vital step in the process as this has a notable impact on improving its performance in terms of training and detection time but also on increasing its accuracy. In this work, we separate the optimal set of features by using the well-known filter-based Chi-Squared selection algorithm.

Chi-Squared algorithm uses the statistical Chi-Squared test of independence to determine the dependency of two variables. If a target variable (e.g. the class label) is independent of a feature variable, we can discard that feature variable as non-important. Therefore, the more dependent a feature variable is to a target variable, the more it can contribute to the prediction process and should be included in the final set of features. The chi-squared statistic (X^2), based on which we make this decision, derives from the following equation and is essentially a measure of the difference between the observed and expected frequencies of the outcomes of a set of events or variables.

$$X_c^2 = \sum \frac{(O_i - E_i)^2}{E_i}$$

Where c translates to the degrees of freedom, O is the observed value(s) and E is the expected value(s).

The function `applyChiSqSelection()` performs the Chi-Squared test of Spark MLlib on the dataframe with the assembled features and returns a vector containing the calculated Chi-Squared statistics that correspond to each feature. These statistics are then indexed and sorted so we can choose the 10 best features. The number of final selected features was empirically chosen because it produced the best results in our experiments but it can change on runtime since it is a parameter of the function. The final Chi-Squared model is built by passing this array with the indices of best features to a `VectorSlicer()` transformer that extracts the selected features out of all the features.

Instead of using the `ChiSqSelector()` that MLlib provides and directly produce the selection model, we thought it would be better to perform the test manually and then use a VectorSlicer because, in this way, we have slightly more control over the implementation.

Building the Machine Learning models

After passing the dataframe through the Chi-Squared selection model, a new column is added that contains a vector with the final set of features. Based on this column along with the class label column, we train and build the Machine Learning models.

Selecting a fitting Machine Learning approach and suitable classifiers is an important stage in the development process. The availability of an effective world-class dataset with a sizable amount of quality labeled training and testing data has lead us to follow a Supervised Learning approach instead of an Unsupervised one. More specifically, the classifiers that we train are:

1. Decision Tree

   Decision Trees are a popular and effective method of classification. They support both numerical and categorical data and they can handle large datasets. A Decision Tree can be represented with nodes and edges. It is composed of a root node that performs the first split and leaf nodes in which we can find the predicted results. In this work we used Decision Tree due to the fact that they are supported by Spark MLlib, they are easy to interpret, do not require feature scaling, are able to handle feature interactions and they are non-parametric, therefore, more flexible.

2. Random Forest

   Many of the strong points of Decision Trees are inherited by Random Forests since they are ensembles of decision trees. Random Forest grows, in parallel, many decision trees, each giving a classification , and merges them while selecting the classification with the most votes in order to get a more accurate and stable prediction. They inject randomness inside the training process by selecting a random subset of features so that each decision tree is different from the other. In addition to the strengths of decision trees, random forests reduce the risk of overfitting that sometimes 'deep' decision trees have and are generally great predictors. Random Forest is natively supported by Spark MLlib.

3. Extreme Gradient Boosting(XGBoost)

   Extreme Gradient Boosting is another decision-tree-based ensemble algorithm that uses a gradient boosting framework. It grows decision trees consecutively while, each time, trying to improve the prediction rate by making small adjustments to the prediction function in order to optimize it. What XGBoost adds to the

Gradient Boosting method is that it uses regularization methods to avoid over-fitting. XGBoost is widely used in solving Machine learning problems and has gained popularity as a strong contender in competitions. It is not currently supported by the official Spark MLlib but we incorporate it as an external library built on top of Spark's API.

We included all three of them in our implementation not only to provide the user with the opportunity to select his preffered method but also to experiment with and compare their performances. The construction of these three machine learning models concludes the Offline phase.

All the aforementioned models (i.e the Chi-Squared model, the ML models and the Pipeline model that contains the data preparation steps) are stored in Amazon S3 to be loaded and used during the Online phase.

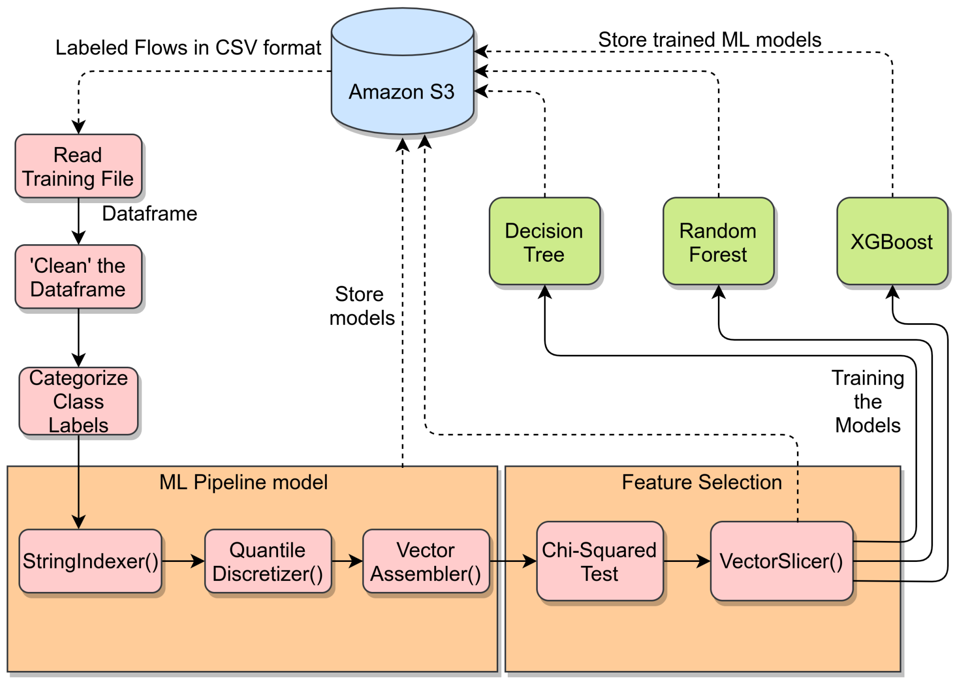


Figure 3.4: Offline Phase Flowchart

3.2.2 Online Phase

As soon as the machine learning models are trained and stored, the training Spark job exits. The Docker service responsible for submitting it to the cluster gets terminated and the container is stopped. A separate service is constantly pinging the training service through a bash script to find out whether it is still running or not. When the ping command returns one, it means that it did not receive a reply, therefore the training is complete and we can kickstart the Online Phase. This separate service is also in charge of submitting the Real-Time Classification Spark job to the spark cluster.

It should be noted that all the Docker services that take part in the Online Phase are already up and running at this point and with submitting this second Spark job, the pipeline is complete and ready to process the incoming flow of data.

Input Handling

In order to simulate a realistic flow of network logs we utilized two python scripts that send the unlabeled NSL-KDD network records to Kafka. The first script sends each flow at random intervals, between zero and four seconds with zero being the most probable interval, to monitor how well the system responds to random network connections and how soon we can see the results in the visualization dashboard. The second script streams the testing CSV file at a specific rate, such as 3000 records/sec to test the processing efficiency of the spark cluster.

A Kafka Console Producer sends this stream of unlabeled records to the partitions of the Kafka Topic that is created at launch. The producer connects to the Docker IPs of these partitions via network and Kafka distributes the incoming records to all available brokers. The number of partitions should be at least the same as the number of Spark workers in the cluster for the optimal parallel processing of data.

In a real world case scenario, some other hypothetical application dumps network logs to a specific location. In such a case, we have included a Kafka Connector named Spooldir that is tasked with watching this directory for new logs and sending them immediately, tuple by tuple, to the Kafka topic, essentially playing the role of a monitoring tool.

Real-Time Classification

When the corresponding Docker service submits the Live Classification job, Spark starts a Structured Streaming application, the progress and details of which can be viewed live in the Spark UI on the configured ip and port through the browser.

At the beginning, all the pre-built models are loaded back into the Spark Workers from Amazon S3. Those are the Decision Tree, Random Forest, XGBoost, Chi-Squared Selection, and Machine Learning preprocessing pipeline models. The user has the option of selecting which of the ML models wishes to be used as a classifier of the incoming network flows.

Spark workers are listening to the Kafka topic and read the stream of new network flows that arrive into a Spark DataFrame. This DataFrame represents an unbounded table containing the streaming network data. Structured Streaming treats a live data stream as a table that is being continuously appended.

The following transformations resemble the course of action of the Offline Phase. A cleaning step takes place to remove the Score column and possible null containing rows. Subsequently, the dataframe is fit on the Pipeline model that contains all the preparation operations required which are now performed on each and every row of the incoming stream. The resulting stream containing now an assembled feature vector passes through the pre-built Chi-Squared model which is essentially a VectorSlicer estimator that directly separates the already selected features.

Afterwards, based on our initial choice, each network tuple passes through the corresponding Classifier model that adds a label based on its prediction. Using an `IndexToString()` transformer, the predictions of numeric type are translated back to a String type for readability. Finally, with the addition of a Timestamp column, the classified stream of tuples is translated into documents and stored in Elasticsearch through the use of elasticsearch-hadoop library that provides native integration.

Visualization

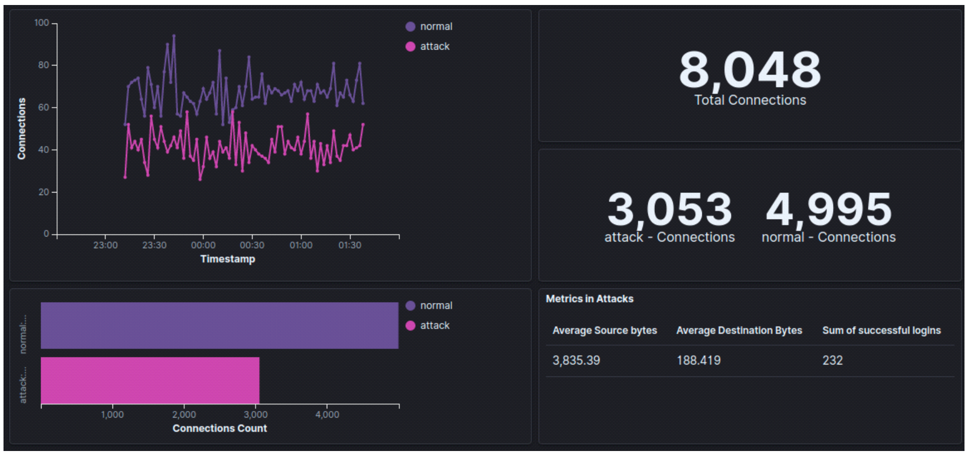
Elasticsearch acts as a highly scalable document oriented database that receives the classified flows and stores them as serialized JSON documents on an index that is created during launch. Based on the number of Elasticsearch nodes that we setup, the stored documents are distributed across the cluster and can be searched for from Kibana in near real-time (i.e. within one second).

Kibana's interface allows us to query the data stored in the Elasticsearch index and then visualize the results through a collection of charts, graphs, metrics, and searches that together form the designed dashboard. This pre-created dashboard is loaded on our Kibana service on setup and, with the usage of the Timestamp column that was added to the data, can be configured to present its insights based on a specific time range or in a continuous fashion. In our case, we included metrics that we thought would be of value such as:

- Comparison between the number of attacks and normal connections with respect to the total number of connections recorded so far in figure 3.5.

- Time chart displaying the number of attacks or normal connections that happened within the selected time range in figure 3.5.

- The number of source bytes, destination bytes, and of successfull login attempts in attack connections recorded so far in figure 3.5.
- The distribution of protocols (tcp-udp-icmp) recorded in all connections in figure 3.7.
- The service distribution in connections displayed as a pie chart in figure 3.7.
- Information about the latest attacks containing the values of the fields stored in the Elasticsearch index in figure 3.8.



8,048
Total Connections

3,053
attack - Connections

4,995
normal - Connections

Metrics in Attacks

Average Source bytes	Average Destination Bytes	Sum of successful logins
3,835.39	188.419	232

Figure 3.5: Kibana Dashboard - Connection distribution and different metrics

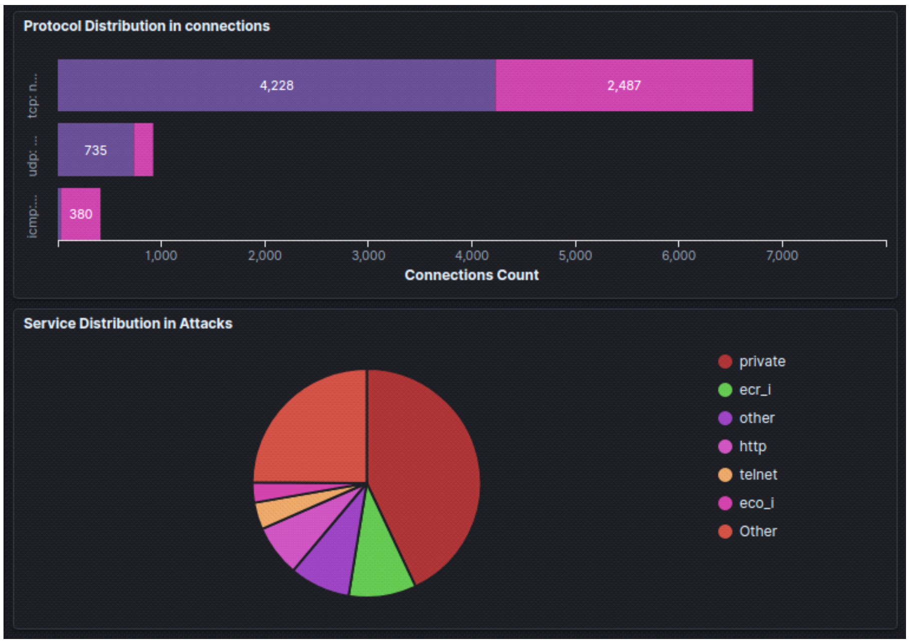
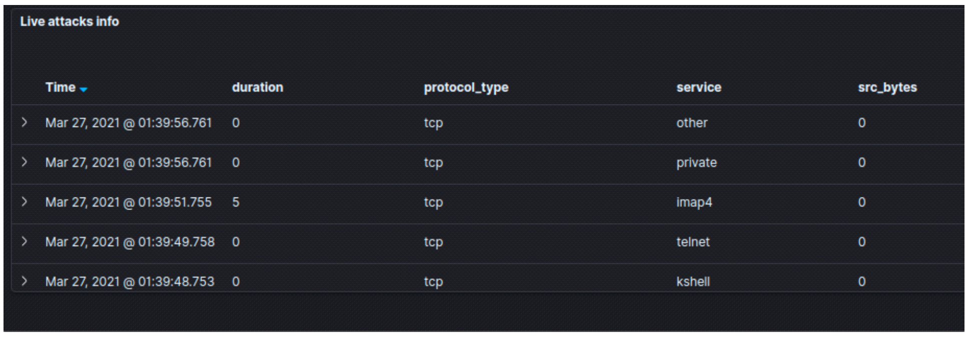
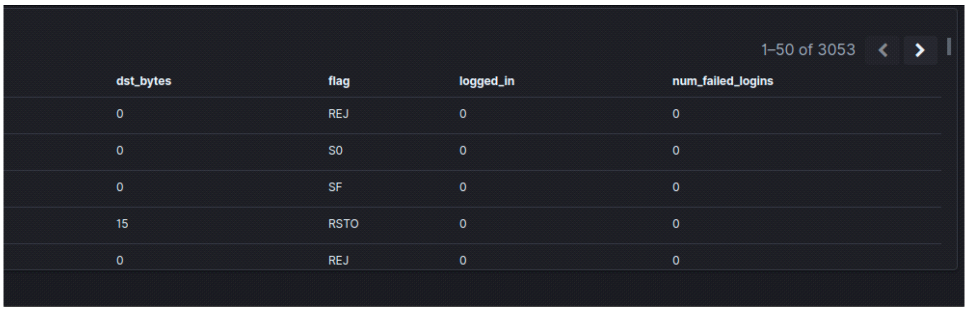


Figure 3.6: Kibana Dashboard - Protocol and service distribution


Live attacks info

Time	duration	protocol_type	service	src_bytes
Mar 27, 2021 @ 01:39:56.761	0	tcp	other	0
Mar 27, 2021 @ 01:39:56.761	0	tcp	private	0
Mar 27, 2021 @ 01:39:51.755	5	tcp	imap4	0
Mar 27, 2021 @ 01:39:49.758	0	tcp	telnet	0
Mar 27, 2021 @ 01:39:48.753	0	tcp	kshell	0

Figure 3.7: Kibana Dashboard - Latest attacks information (1)



1–50 of 3053

dst_bytes	flag	logged_in	num_failed_logins
0	REJ	0	0
0	S0	0	0
0	SF	0	0
15	RSTO	0	0
0	REJ	0	0

Figure 3.8: Kibana Dashboard - Latest attacks information (2)

3.3 Containerization and Orchestration

Docker allows the automated deployment of the whole system on a Docker Swarm cluster with a single command. Every technology used in this work is launched with a configuration detailed in the docker-compose file.

Initially, Docker starts up all services in the form of containers and deploys them. Using Docker Compose, we can either deploy it distributively in a cluster among all nodes in Swarm mode or as a single-server application in local mode for testing.

The services launched as containers are:

- Zookeeper: A service required by Kafka. Kafka uses Zookeeper to do leadership election of Kafka Broker and Topic Partition pairs. It acts as a manager and sends changes of the topology to Kafka, so each node in the cluster knows when a new broker joined, a Broker died, a topic was removed or a topic was added, etc. Zookeeper provides an in-sync view of Kafka Cluster configuration.

- Kafka: Creates the cluster of brokers that provide the Kafka service and scales them accordingly, based on the Docker Swarm labels of the nodes. Depends on Zookeeper. Also, configured to automatically create a topic for the network traffic with the number of partitions being equal to the number of spark workers.

- Spark Master: In the master-worker architecture of a spark standalone cluster, Spark Master is the resource manager for the cluster in charge of allocating the resources among the Spark applications. The resources are used to run the Spark Driver and Executors. Configured to deploy on a node with the master role label and communicates with workers through the exposed ports of the container. The Spark UI can also be viewed through the browser on one of these ports.

- Spark Worker(s): Launches the fleet of workers that constitute the spark cluster and enlists them through communication with the Spark Master on its exposed ports. The replicas configuration option dictates the numbers of workers.

- Spark-submit for Training job: Service that utilizes the Spark-submit script provided by Apache Spark to launch applications on the cluster. It submits the jar package that contains our code and its dependencies to the Spark master. This service runs a specific main class corresponding to the Training process and starts after the Spark Master and Worker(s) services have already started. In other words, the training starts after the spark cluster have been setup. The image that we created for this service packages the spark-submit service of Apache Spark along with the jar file of our application and some additional functionality.

- Spark-submit for Real-Time Classification job: We use the same image as the previous service but through a parameter setting in Docker, a different main class is used and the Real-time Classification Spark job is submitted to the cluster to start the new Structured Streaming application. Within the script that submits the application, we ping the previous service to figure out whether or not it has completed so we can kickstart the Online Phase.

- Elasticsearch: Creates the search engine cluster. The names of the Swarm nodes that correspond to Elasticsearch nodes have to be provided for inter-node discovery. The number of nodes is equal to the number of nodes with the 'esnode' role label.

- Kibana: Launches the visualization service. Single instance since that is all that is required. The port 5601 is exposed by this container in order for a user to connect through his browser on the ip of the node that this service runs and view the created dashboard.

- Elasticsearch-Kibana Setup service: Single container that contains a simple environment running a script that creates Elasticsearch index for the storage of the classified network flows and also loads the pre-created Kibana dashboard. This separate setup service was required because the dashboard and index have to be created after the Elasticsearch and Kibana services are already up and running.

Every main component of the system is by design distributed and able to support scalability, one of the intended features of our intrusion detection system. The number of Kafka brokers, Spark workers and Elasticsearch nodes can be increased or decreased based on our demands. To add new nodes to the Swarm and make these resources available for scaling-up the services, we just have to enlist and label them with the corresponding Docker commands.

Instead of managing the deployment of each part of our system manually, Docker offers us a more automated way. It enables us to adjust the desired amount of resources simply by configuring the parameters regarding the number of replicas of each service. Docker Swarm handles the necessary management and offers clustering, scheduling and integration capabilities for our distributed application.

One other advantage of Docker is that it provides restart policies to control whether your containers start automatically when they exit, or when Docker restarts. If one of service fails, Docker will automatically attempt to restart it on one of the available nodes. This policy can be configured in the Docker-Compose file. In our case, the policy of the Spark-submit Training job service is set to only restart upon failure and not on completion. This is because we need it to run exactly once successfully for the machine learning algorithms to be trained.

3.4 Experimental Results

3.4.1 Algorithm Performance Metrics

Our initial intention, when designing the intrusion detection system, was to perform multi-class classification and thus, train the machine learning algorithms to be able to successfully recognize whether the incoming network records are normal connections, dos, probe, r2l or u2r attacks. These are the five label categories included in the NSL-KDD dataset. Though, as explained below, due to certain issues regarding the number of training samples corresponding to each attack class of the dataset we switched to binary classification.

Multi-class Classification

The issue that arises when trying to classify the records into its five classes is due to class imbalance. Class imbalance is a problem that occurs in machine learning classification problems and it tells us that the occurence of one of the classes is very high compared to other classes present. Due to the difference in class frequencies, the algorithms tend to get biased towards the majority values and do not perform well on the minority values. This affects the overall predictability of the models since the algorithms do not have enough data to learn the patterns present in the minority classes.

In an attempt to combat this issue, we added class weights to take into account the skewed distribution of the classes. The difference in weights will influence the classification of the classes during the training phase. The whole purpose is to penalize the misclassification made by a minority class by setting a higher class weight and at the same time reducing weight for the majority class. The weights that we added are calculated during the training phase based on the number of occurences of each class.

We also tuned hyperparameters such as the number of features selected by Chi-Squared selection or the depth of the trees of the three algorithms in order to find the optimal settings.

For the evaluation and performance comparison of our classifiers, we utilized a suite of metrics provided by Apache Spark's MLlib that are calculated by comparing the predicted labels against the already existing labels of the NSL-KDD testing set. The metrics are as follows:

Metric	**Description**	**Formula**
Accuracy	Accuracy measures precision across all labels	$AC = \frac{TP+TN}{TP+FP+TN+FN}$
Precision	Proportion of correct labels that were classified over all labels	$P = \frac{TP}{TP+FP}$
Recall	Proportion of correct labels that were classfied over all positive labels	$R = \frac{TP}{TP+FN}$
F-measure	Harmonic average of Precision and Recall	$FM = 2 * \frac{P*R}{P+R}$

Where TP = True Positives, TN = True Negatives, FP = False Positives and FN = False Negatives.

After experimenting with different values for the number of selected features out of the total 41 features and the depth parameter, the best results generated are:

- Decision Tree - With the number of selected features = 25 and tree depth = 7

	Accuracy	F-measure
With Class Weights	0.755	0.731
Without Class Weights	0.732	0.689

- Random Forest - With the number of selected features = 25 and tree depth = 7

	Accuracy	F-measure
With Class Weights	0.761	0.717
Without Class Weights	0.739	0.691

- XGBoost - With the number of selected features = 7 and tree depth = 7

	Accuracy	F-measure
With Class Weights	0.783	0.741
Without Class Weights	0.768	0.725

As we can see here, the best performance was produced by the XGBoost algorithm with an accuracy of 78% and an F-measure of 74%. Also, we notice that the addition of class weights does increase the predictability of the models but only slightly.

Even though with this parametarization we manage to yield significant accuracy percentages, upon further inspection of the classified results in the spark dataframe we noticed that the low-appearing classes (R2L or U2R) were very rarely correctly identified by the ML models. That means that a U2R or R2L attack will probably not be classified correctly. This is due to the low quantity of training samples corresponding to these classes in the NSL-KDD dataset.

Therefore, instead of providing predictors that would identify three out of the five classes and almost always misclassify the other two, we decided to resort to binary classification and provide a better overall predictability.

Binary Classification

Essentially, we merged all the attack classes into a single label called attack and noticed that it was easier for the models to identify these records as attacks instead of recognizing the specific attack class.

For the evaluation and performance comparison of the binary classifiers, we went with a different set of metrics of Apache Spark. The area under ROC and area under PR. This is because the accuracy is not a reliable measure when distinguishing between only two classes. For example, in a dataset where there are 70% positive values and 30% negative, a classifier that always predicts the positive value, would still have an 70% accuracy.

AUC - ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC represents the degree or measure of separability. Essentially, it depicts how capable is the model in distinguishing classes. It is a plot between the true positive rate and the false positive rate.

A precision-recall curve is a plot of the precision and the recall for different probability thresholds. A high area under the PR curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate.

The results generated with the optimal parameter values found are:

- Decision Tree - With selected features = 7 and tree depth = 7

Area under ROC	Area under PR
0.913	0.934

- Random Forest - With selected features = 10 and tree depth = 7

Area under ROC	Area under PR
0.970	0.974

- XGBoost - With selected features = 10 and tree depth = 7

Area under ROC	Area under PR
0.951	0.953

We can see here that with performing binary classification we manage to get significantly better results. Especially from the Random Forest model that yields 0.97 at both metrics (the closer to one at both values, the better the predictability).

3.4.2 Scalability

In order to assess the scalability of the system and investigate how well its processing capabilities respond to heavy load of incoming data, we utilized a script written in Python that streams the CSV testing file at an increasing rate as shown in the figures below. This procedure was followed for different cluster setups with a gradually increased factor of parallelism. Since the streaming rates are quite high and the script reaches the end of testing file quickly, we adjusted it so it reopens and sends the same file. In this way, we manage to have a continuous flow of data for an adequate testing time period.

Spark Cluster Setup

The experiments in this section were performed in an AWS cluster with an increasing number of Spark workers in each test. Each single-thread worker node had four cores and 1GB of RAM.

Experiments

At first, the configuration of the application used two worker nodes to run the experiment. This is translated as two Spark worker containers that were created as replicas.

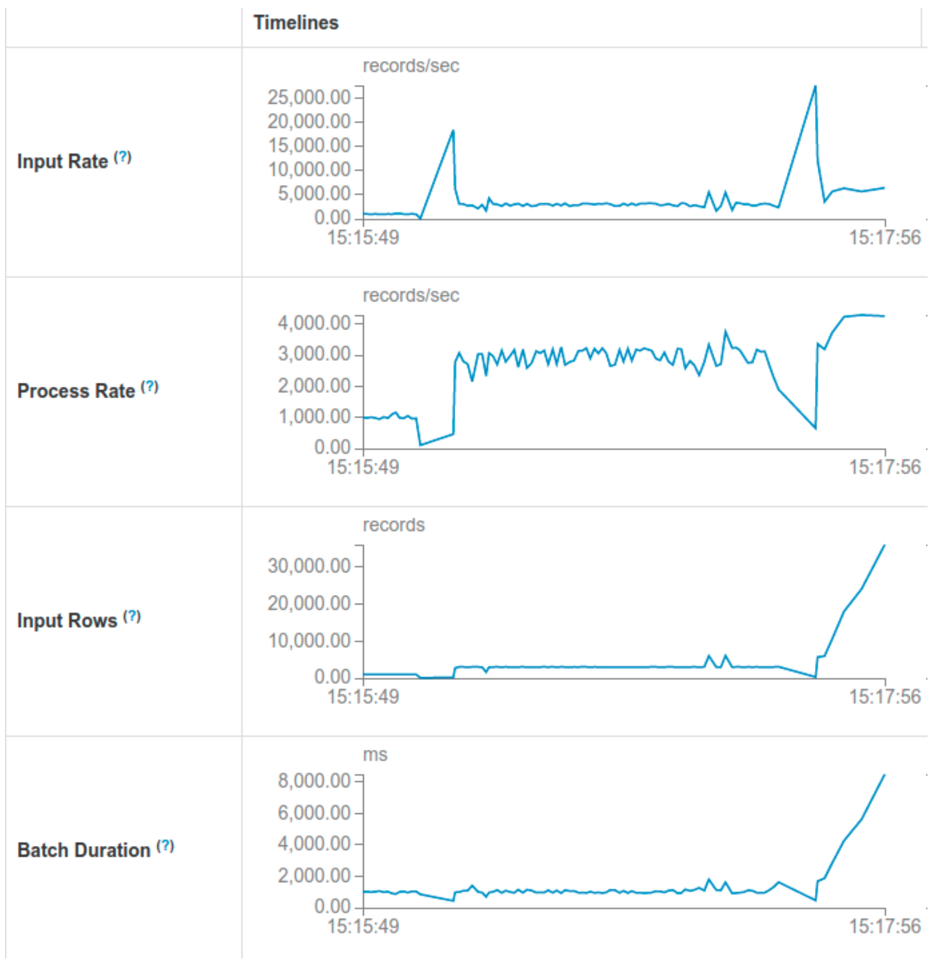


Figure 3.9: Spark UI 2 workers

As depicted in figure 3.9, we gradually increased the input rate to assess the processing efficiency of this setup. At an input rate of 1000 records/sec, the process rate was able to meet the demand while maintaining a batch duration of one second. The same happens at the 3000 records/sec with no increase of the batch duration which means that the cluster was able to handle this load without problems. Though, with a further increase of the input rate, we notice that the batch duration start increasing dramatically. This state shows that we reached the limit of its processing power. The maximum process rate that it was able to reach is around 4000 records/sec.

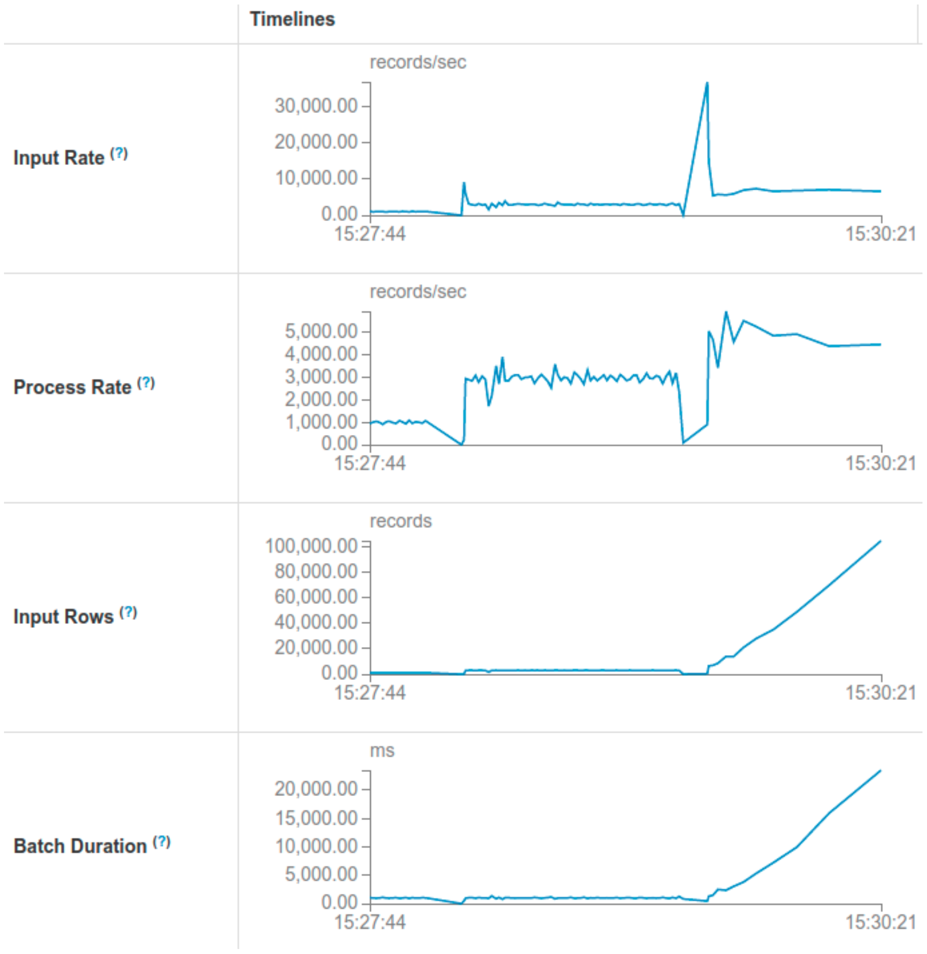


Figure 3.10: Spark UI 4 workers

In figure 3.10 that shows the performance of the Spark cluster with four workers, we can see that it follows a similar pattern. The application can handle the increasing input rate with a steady batch duration but after a certain point, the batch duration skyrockets. At this point, the maximum processing rate reached with 4 workers is around 4800-5000 records/sec.

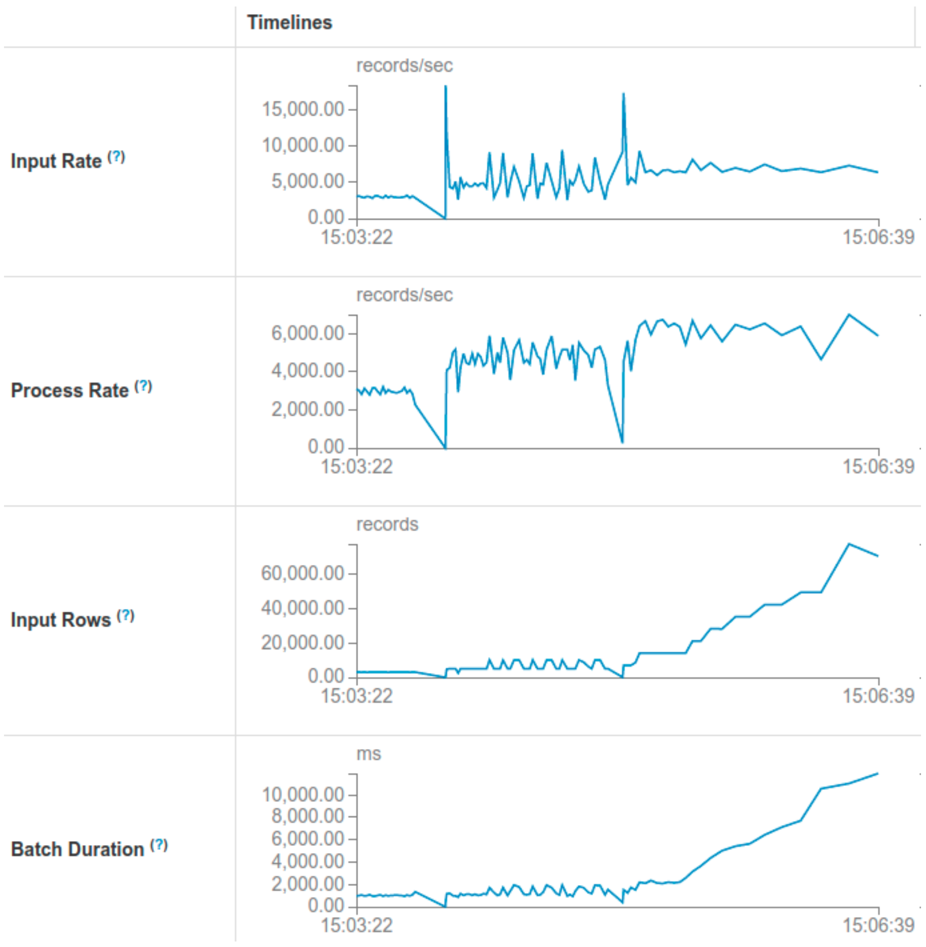


Figure 3.11: Spark UI 6 workers

Finally, in the last experiment with a cluster of six workers, the application manages to reach a maximum processing rate of around 6000 records/sec at the threshold where the batch duration starts escalating.

Number of workers	Maximum processing rate
2	around 4000 records/sec
4	4800-5000 records/sec
6	around 6000 records/sec

Table 3.1: Summarized results

By observing the summarized results of the aforementioned experiments in the table 3.1 we can conclude that the processing capacity of the designed intrusion detection system does indeed scale with further resources that are added into the topology in the form of Docker containers on new nodes.

Chapter 4

Conclusion and Future work

4.1 Conclusion

In this work, we discussed the need for a fast, real-time intrusion detection system to handle evolutive network traffic and provide classifications to protect a network as well as live insights on the nature of connections through a visualization panel. Our proposed system uses Apache Spark Structured Streaming to process and detect anomalies in real-time. The end product is a distributed, scalable, fault tolerant and easily deployed system that was tested in AWS to showcase its performance within a distributed cloud infrastructure. We used the NSL-KDD dataset to evaluate it against cyber-threats and managed to yield classifiers with a 97% accuracy. In addition, the proposed system is designed to be deployable on any environment regardless of the underlying infrastructure. The only requirement is the installation of Docker.

4.2 Future work

As a future possible improvement, utilizing the newly introduced feature of Spark Structured Streaming: continuous streaming could yield better results with regards to the speed of data processing. Even though it is still an experimental feature, it is claimed to enable end-to-end data processing of one millisecond. Also, a useful feature would be the ability to configure the scalability level of the application without affecting the already running Spark job.

In conjuction with the proposed system, a secondary application that further recognizes the exact type of attack and provide more insights could be a useful addition. Alternatively, a threat prevention tool could start to take action against the incoming threat when alerted. There are several ways to take preventive measures such as sending an alarm to the administrator, blocking traffic from specific source addresses, or changing firewall configurations.

Bibliography

[1] *Cisco Cybersecurity Reports*. 2018. URL: https://www.cisco.com/c/en/us/products/security/cybersecurity-reports.html.

[2] Govind P. Gupta and Manish Kulariya. “A Framework for Fast and Efficient Cyber Security Network Intrusion Detection Using Apache Spark”. In: *Procedia Computer Science* 93 (2016). Proceedings of the 6th International Conference on Advances in Computing and Communications, pp. 824–831. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.07.238. URL: https://www.sciencedirect.com/science/article/pii/S1877050916314806.

[3] Mounir Hafsa and Farah Jemili. “Comparative Study between Big Data Analysis Techniques in Intrusion Detection”. In: *Big Data and Cognitive Computing* 3.1 (2019). ISSN: 2504-2289. DOI: 10.3390/bdcc3010001. URL: https://www.mdpi.com/2504-2289/3/1/1.

[4] *Apache Spark*. URL: https://spark.apache.org/.

[5] *Apache Flink*. URL: https://flink.apache.org/.

[6] *Apache Storm*. URL: https://storm.apache.org/.

[7] Mahbod Tavallaee et al. “A detailed analysis of the KDD CUP 99 data set”. In: *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*. 2009, pp. 1–6. DOI: 10.1109/CISDA.2009.5356528.

[8] *Apache Kafka*. URL: https://kafka.apache.org/.

[9] Elasticsearch. *Distributed search and analytics engine based on Apache Lucene*. URL: https://www.elastic.co/.

[10] Docker. *An open platform for developing, shipping, and running applications*. URL: https://www.docker.com/.

[11] L. Dhanabal and S. Shantharajah. "A Study on NSL-KDD Dataset for Intrusion Detection System Based on Classification Algorithms". In: 2015.

[12] The UCI KDD Archive. *KDD Cup 1999 Data*. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[13] Eman Shaikh et al. "Apache Spark: A Big Data Processing Engine". In: *2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM)*. 2019, pp. 1–6. DOI: 10.1109/MENACOMM46666.2019.8988541.

[14] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 2.

[15] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, 1383–1394. ISBN: 9781450327589. DOI: 10.1145/2723372.2742797. URL: https://doi.org/10.1145/2723372.2742797.

[16] Michael Armbrust et al. "Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 601–613. ISBN: 9781450347037. DOI: 10.1145/3183713.3190664. URL: https://doi.org/10.1145/3183713.3190664.

[17] Xiangrui Meng et al. “MLlib: Machine Learning in Apache Spark”. In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), 1235–1241. ISSN: 1532-4435.

[18] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos M. B. Duarte. “A Performance Comparison of Open-Source Stream Processing Platforms”. In: *2016 IEEE Global Communications Conference (GLOBECOM)*. 2016, pp. 1–6. DOI: 10.1109/GLOCOM.2016.7841533.

[19] Gerry Saporito. “A Deeper Dive into the NSL-KDD Data Set”. In: *Towards data science* (2019). URL: https://towardsdatascience.com/a-deeper-dive-into-the-nsl-kdd-data-set-15c753364657.

[20] *What is a denial of service attack?* URL: https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos.

[21] Mouhammd Alkasassbeh and Mohammad Almseidin. “Machine Learning Methods for Network Intrusion Detection”. In: *CoRR* abs/1809.02610 (2018). arXiv: 1809.02610. URL: http://arxiv.org/abs/1809.02610.

[22] R. Staudemeyer and C. Omlin. “Extracting salient features for network intrusion detection using machine learning methods”. In: *South Afr. Comput. J.* 52 (2014), pp. 82–96.

[23] *NSL-KDD dataset.* URL: https://www.unb.ca/cic/datasets/nsl.html.

[24] Khalil El-Khatib. “Impact of Feature Reduction on the Efficiency of Wireless Intrusion Detection Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 21.8 (2010), pp. 1143–1149. DOI: 10.1109/TPDS.2009.142.