

Anomaly-based Intrusion Detection System for the CERN Security Operations Center

Master Thesis

Author : Melchior Thambipillai^{†‡}
Industry Supervisor : Liviu Valsan[†]
Professor : Prof. Bryan Ford[‡]

[†]*CERN, Geneva, Switzerland* and [‡]*EPFL, Lausanne, Switzerland*

August 13, 2018

Abstract

The CERN Security Operations Center monitors the network traffic and the command executions on different machines at CERN. To detect potential attacks, intrusions and security incidents, it makes use of signature-based Intrusion Detection Systems (IDSs) like Bro and Snort. This kind of IDSs works well to detect attacks with well-known signatures or patterns, but it fails to detect zero-day attacks or unknown patterns. On the other hand, anomaly-based IDSs detect by learning patterns in the data and looking for anomalies. They often have a high false positive rate but they don't require any additional knowledge about the attacks and can detect intrusions with unknown signatures. This thesis presents the design, implementation and evaluation of the anomaly-based IDS we created for the CERN SOC to complement the existing signature-based IDSs. It works on unlabeled data and combines three different unsupervised machine learning algorithms to detect anomalies in logging data : Isolation Forest, K-Means and Local Outlier Factor. It is designed to work on very large amounts of data and is built on Big Data technologies like HDFS and Apache Spark.



1 Introduction

In this section, we will describe cyber-security at CERN and why an anomaly-based IDS would increase the security level. In section 3, we will describe the design of the system and in section 4 we will dive in the implementation details of the system. Then section 5 will provide some evaluation results of the system. After the conclusion in section 6, the section 7 will describe some possible future work.

1.1 Problem Statement

CERN is linked to a lot of other institutions for different communication and research purposes. The data produced by the experiments of the Large Hadron Collider (LHC) has to be processed by data centers located across the globe forming the Worldwide LHC Computing Grid (WLCG). A lot of people visit CERN, have time-limited contracts or interact with CERN resources from another institution. This results in a very large amount of networking data coming from a lot of different institutions and people across the world. The CERN machines are used by thousands of people for different purposes. Securing a system so large and so widely open to the world requires a Security Operations Center (SOC) to have eyes on the network traffic and computing resources to detect security incidents.

The CERN SOC described in more details in 1.3 uses signature-based Intrusion Detection Systems (IDSs) to raise alerts when a known attack is detected on the traffic. This kind of detection works well for attacks with well-known signatures but it fails to detect zero-day attacks and attacks occurring on a longer time period. Another kind of IDSs called anomaly-based try to detect anomalies in the data without requiring any previous knowledge. The SOC would benefit from such an IDS to complement the signature-based IDSs and increase the security level.

1.2 Objectives

The goal is to design, implement and integrate an anomaly-based IDS for the CERN SOC. It has to take the logs produced by the SOC as input, detect anomalies and output relevant information to the security analyst for further investigation. The IDS must be as generic as possible to be able to ingest any data source (network protocols logs, certificate logs, file logs, execution logs, etc) with different parameters to adapt the IDS to the current needs. Some techniques might work for some data sources but not others. So the IDS should provide different algorithms to detect anomalies and it should be easy to add new algorithms to the system in the future. Each of these algorithms will be an unsupervised machine learning algorithm since we don't have any labeled data.

The system will be triggered by the security analyst to try to detect anomalies on the logs produced during some time period. The system should be able to process the amount of data stored for a time period of at least a month. For longer durations, if the amount of data is too large, it should be possible to detect on subsets of the data individually. Creating a periodic job to trigger the system should be easily feasible but is not part of the project. As a consequence, real-time detection is not a feature of the system. It detects anomalies on a data set, not on single real-time log entries. Serving the model for real-time detection or

updating it in real-time is considered out of the scope of the project.

Like any anomaly-based IDS, it will detect anomalies not intrusions. Even if the final goal from the operational point of view is to detect intrusions or security incidents, anomaly-based IDSs make the hypothesis that intrusions are anomalies, meaning they are rare and different from the rest of the data, and apply different anomaly detection techniques to try to detect intrusions. This doesn't mean that all anomalies are intrusions. From the machine learning point of view, if the system detects a true anomaly which is not an intrusion, it is still a true positive.

In anomaly detection and in most of the machine learning problems, evaluation has two components : precision and recall. We have a dataset which contains normal data and anomalous data. Our system will retrieve a subset of this data set and classify each point in that subset as anomalous even though it might be wrong. "Precision" measures the amount of true anomalies that were correctly detected out of all the points that were detected. On the other hand, "recall" measures the number of detected anomalies out of all the anomalies contained in the data that should be detected. These two notions can be mathematically expressed in the following way :

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

where TP denotes the number of true positives, FP the number of false positive and FN the number of false negatives.

These two measurements are trade-offs balancing each other. If we detect every single point in the whole dataset as anomalies, we obtain a recall of 100% because we detected every anomaly contained in the data but the precision is very low. If we don't detect any point as an anomaly, we obtain 100% precision because in our empty set of results there is no mistake, but the recall is now zero. In different machine learning problems, depending on the situation, one might be more important than the other and the system can be adapted to favorize either precision or recall.

In our case, the objective is to put a very strong emphasis on precision. So the goal is to raise alerts which are almost all the time actual anomalies, we want to minimize the false positive rate and we don't try to detect every anomaly present in the data.

1.3 CERN Security Operations Center

The CERN Security Operations Center monitors, logs and investigates the data from different data sources like the network traffic and the command executions on CERN machines. This real-time data can reach several hundreds of gigabytes per day and is ingested by a stream processing engine and available for 30 days to be queried efficiently by Elastic Search. The data is then stored on HDFS for one year and available for long-term investigation. The SOC provides visualization tools to help the security analyst investigate the data.

At the different steps of the pipeline, different systems can automatically detect and prevent different kinds of attacks and intrusions. Signature-based IDSs like Bro and Snort try to

detect known signatures of attacks and the Malware Information Sharing Platform (MISP) checks for different Indicators of Compromise (IOC) contained in its database. The Bro IDS creates insights and aggregations of the real-time data which counts as another data source to be ingested by the SOC and persisted to HDFS.

The anomaly-based IDS that we want to create will take as input the different data sources persisted to HDFS in real-time. It will use months of data to learn patterns from this unlabeled data and try to detect anomalies in the data over the long term.

2 Related work

2.1 Types of Intrusion Detection Systems

IDSs can be either be host-based or network-based. Host-based IDSs are implemented in endpoints to detect different kind of malware and unexpected host behavior. On the other hand, network-based IDSs inspect the network traffic between hosts to detect different kind of attacks and intrusions attempts. This IDS is designed to be both network-based and host-based as it depends on the logs used as input, which can be either network traffic logs or execution logs.

Most Intrusion Detection Systems try to recognize known attacks. They are called signature-based IDSs and have the advantage to be easier to design and can provide very high precision for known attacks. The signatures for the attacks must be constantly maintained to include the newly known attacks. This kind of IDS will fail to recognize intrusions unknown to the database. On the other hand, anomaly-based IDSs try to recognize anomalies in the data, assuming that attacks and intrusions are somewhat abnormal. This approach has the advantage to be able to detect zero-days attacks or intrusion patterns not included in the intelligence database. They are significantly more difficult to design as the false positive rate can be very high. The IDSs already deployed in the CERN SOC, Bro and Snort, are signature-based and this IDS is anomaly-based to complement them.

Anomaly-based IDSs use different machine learning techniques to detect anomalies in the data. In some cases, there exists already a dataset of logs labeled with "normal" or "anomaly" tags. This allows to train a classifier on this dataset and use the model to classify anomalies in new logs. These techniques are called supervised learning. Unfortunately, in most cases there is no such dataset. In the case of CERN, the amount of data is way too large to consider a manual tagging of even a small fraction of the logs. We are then left with unsupervised learning: we need to create systems able to distinct normal traffic from anomalies in a dataset without any labeled data. The subsection 2.3 will describe different possible techniques.

2.2 Features

Like in any machine learning problem, we need to extract features from the data and pre-process them so that they can be used by anomaly detection algorithms.

2.2.1 Extraction

In network-based IDSs we distinguish three kinds of features : basic features, traffic features and content features.

Basic features focus on a single log entry (usually an IP datagram in the case of network-based IDSs) and use the information of the different headers (Version of the protocol, number of bytes, TCP flags, etc) as features. In our case, each network log entry is a connection so basic features for network-based IDSs also include duration of the connection, number of bytes sent/received, etc... Basic features for host-based IDSs are usually extracted from one-line command executions and can contain information like user ID, process ID, words of the command, etc...

Traffic features use aggregations of the logs for some time window. The logs can be grouped by the source entity (usually defined by the host name for network-based IDSs and user ID for host-based IDSs) to examine the total number of connections made by each user, the most common command executed, the most common server users connected to, the average duration of their connections, etc. Logs can also be grouped by destination entity to examine the activity of each server during the time window, or by protocol to look at the statistics of the different protocols like HTTP, SSH, SMTP, etc... In our case, the IDS supports aggregation for source and destination entities, but only aggregation based on the source was used for evaluation.

Content features look at the payload of packets. This is highly application-specific, even if some research explored a way to use the probability distribution of bytes sequences as features[1]. This requires the packets payloads to be in clear text, which is not the case for most of the network logs at CERN so we didn't use any content features.

2.2.2 Scaling

The values of the different features can belong to very different ranges, which can lead to some features having a lot more impact on the results compared to other features rendered meaningless. To avoid this issue, features need to be scaled in a uniform way. There are multiple ways to do it, we explored two different ones: rescaling the range and normalizing to unit length.

Rescaling the range maps all the values of each feature individually to values between 0.0 and 1.0, meaning that the minimum value will be mapped to 0.0 and the maximum to 1.0. Every value is mapped according to the following formula :

$$f(x_i) = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Another common way in machine learning to scale features is to normalize each row such that its vector length is 1.0. This can be achieved by dividing each feature value of the row

by the norm of the row:

$$f(r) = \frac{r}{\|r\|}$$

2.3 Unsupervised Anomaly Detection

Three different unsupervised anomaly detection algorithms were implemented. They are extensively described in the sections 3 and 4. The following algorithms are also unsupervised and have been considered. They could be added to the implementation of the IDS in some future work.

2.3.1 DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm. It divides the points in the dataset into categories : core points, density-reachable points and outliers. Core points are defined as points having more than some number of neighbors n within some distance ϵ . These neighbors are said to be directly reachable from the core point and we call a point density/reachable to the core if there exists a path of directly reachable points leading to the core. The algorithm assigns a cluster to each of these core points and their respective density-reachable points. Outliers are points that are not directly reachable to any other point and don't belong to any cluster.

This algorithm has the advantage to not require a predefined number of clusters as opposed to K-Means, but it requires to set n and ϵ . It defines the notion of noise which is very useful for anomaly detection : DBSCAN can be used to define outliers as anomalies as well as clusters with small sizes. It is very computationally expensive but there exists already a parallel version of DBSCAN on Spark which should perform better than the single-threaded original version on large data sets[2].

2.3.2 Principal Components Analysis

PCA is generally used to reduce the dimensionality of a dataset, either to make a computation less expensive or for visualization purposes. The data is projected onto its principal components space and can then be reconstructed to the original data space. It can be used for anomaly detection as well[3]: if only the top n principal components are used for both projection and reconstruction, the normal points will have a low reconstruction error since the top principal components account for the variance of normal points, on the other hand, outliers will have a high error since they won't be well reconstructed. While the concept is sound, PCA requires matrix decomposition to find the eigenvalues and then a lot a matrix multiplications which don't scale for very large datasets.

2.3.3 Unsupervised SVM

Support Vector Machines are initially a supervised learning technique to create a binary classifier. A training set is extracted from the data set and the algorithm tries to find a hyperplane that separates the data in a way that most of the points from one class are on

one side and the other points are on the other side of the hyperplane. In case of a linear SVM, this hyperplane is constructed from a linear combination of points in the training set called support vectors. An optimization technique, usually sub-gradient descent can be used to find the best hyperplane minimizing the classification errors. The rest of the data can then be classified using the hyperplane as a delimiter to determine the category of each point.

This algorithm can be adapted for unsupervised anomaly detection[4]: the algorithm tries to find a dense region where most of the data points lie and classify them as normal points. The hyperplane is constructed around this class and other points are considered anomalies.

2.3.4 Self Organizing Maps

SOMs, also called Kohonen maps, are neural networks well suited to analyze and visualize high-dimensional data in a 2D data space. The map (or grid) is constituted of a two-dimensional array of neurons which are the output layer. Each data point goes through an input neuron which is connected to all output neurons of the grid. Basically, each input vector will try to find the best matching weight vector, the neuron where the weight vector leads to is called the 'winning' neuron or the Best Matching Unit (BMU). The weights of the adjacent neurons are updated with a learning rule. After processing all data points, the map shows where most of the data points lie, clusters are formed within the map to show the denser regions.

This can be used for anomaly detection[5] as normal points are attracted to the most dense regions and points going to sparse regions are anomalies. The fact that the output representation is a two-dimensional grid with weights for different densities allows to visualize the repartition of the data between normal points and anomalies. The disadvantage is that the number of neurons in the grid affect the performance of the algorithm. For very large datasets, it requires a large grid and heavy computation time. However, the MapReduce paradigm can be used to take advantage of parallelization and reduce the computation time. It has been implemented and tested on Spark[6].

2.3.5 Deep Learning

One way to use deep learning for anomaly detection is to build deep neural networks to learn the normal behavior of users in the corporation so that we can compute how they deviate from their normal behavior to spot anomalies[7]. We build one deep neural network per user, it can be a LSTM (Long Short-Term Memory) neural network for example. The activity of users is aggregated in real-time and fed to its corresponding neural network. It will learn the normal behavior of the user from the different input vectors that it gets. This will allow the network to build predictions for the future input vectors. Once the network is trained, each input vector can be compared with the prediction. The more far away too each other the vectors are, the more anomalous the input vector is considered.

This approach has the advantage to build a different model for each user and detect patterns which are anomalous for one particular user but wouldn't be for other users. The downside is that we need to be able to identify the users which is not always possible. Another constraint is that we also need to have the normal behavior of the users to be able to train the neural

networks.

2.4 Ensemble techniques

The different features and algorithms give us different anomaly scores for each data point. There are many ways to try different algorithms and then combine the different scores obtained. These techniques are called ensemble techniques.

In this IDS, three different anomaly detection algorithms were implemented. One ensemble technique[8] that was implemented is to run each algorithm independently on the dataset and combine the different scores using an aggregation function like the mean or the maximum. To ensure that the scores of every algorithm have the same weight when applying the aggregation function, the technique requires that every algorithm assigns scores within the same range, usually between 0.0 and 1.0.

This combination technique could be also applied to other sets of models: we could run the same algorithm with different parameters or using different subsets of the features (feature bagging). The selection of subsets of features is usually done using supervised learning but there exists techniques for unsupervised feature selection especially for IDSs[9]. We could also use different subsets of the dataset, it has been shown[10] that combining different sub-sampling anomaly detectors can give better performance than a single detector on the complete data. Isolation Forest[11] (described in 3.3.1) is itself an example of this technique.

2.5 User Feedback

Even with unlabeled data, once unsupervised models are constructed, supervised models can be designed as well by using user feedback. The anomalies with the highest scores will be shown to the security analyst who must then decide for each anomaly whether it is a true or false positive. In the case of a true positive, an incident response procedure should start. This user evaluation of the results can be used by the system as a feedback to build a labeled training set. Once there is enough data in this set, a supervised model can be trained and applied to the new incoming data. The unsupervised and supervised models can be combined with ensemble techniques. It has been shown[3] that this technique can increase precision. The supervised model helps the system to learn more accurately what are anomalies and thus increases precision. The unsupervised models on the other hand, help the system to detect new unknown anomalies and keep a high recall.

3 Design

3.1 Overview

The pipeline of the entire system, from the logs to the analyzed results, has been split into three big steps that will be run separately :

- Feature extraction
- Anomaly detection
- Results inspection

Each of these steps is a separate command that the user can trigger manually. There is a fourth command available to the user to optimize the different anomaly detection algorithms but it is not strictly part of the pipeline.

In the feature extraction step, we read the logs from the database and convert every column to a double type according to the content of the column. We then aggregate these logs, usually per user, according to diverse aggregation functions and some specified time window. These aggregated logs consisting only of numbers are then scaled according to a machine learning technique and persisted to the database as the final features.

In the anomaly detection step, we read the features from the database and apply different algorithms to detect anomalies in the features. An anomaly is rare and different from the rest of the data, we assume that intrusions are anomalies but not all anomalies are necessarily intrusions. These algorithms are then combined using some ensemble technique to produce some anomaly score for each row. The rows with a score above some threshold are persisted to the database.

In the results inspection step, we read the anomalies persisted in the previous step and reconstruct the corresponding original logs before the feature extraction step by fetching them again from the database. We then apply a set of specified rules to each set of logs to flag them with tags and comments specifying if they are true anomalies or false positives. We can then compute a precision measurement and persist the analyzed results to the database. This precision measurement is always part of the process because, as explained in 1.2, we focus solely on precision to evaluate the system and knowing the precision can help the security analyst to investigate the results. However, we implemented a way to obtain an approximate recall measurement by injecting our own anomalies.

We are now going to describe in more details these three steps of the pipeline.

3.2 Feature Extraction

The extraction of features can be seen as an ETL process (Extraction, Transformation and Loading) and so consists of different sub-steps:

- Reading and converting
- Defining source and destination entities
- Aggregation

- Scaling
- Statistics computation and storing

In order to read and convert every column to a numeric type, we need to define a schema of the data source. The logs are stored with a schema defining the different columns and their types. We define our own schema which defines which of the original columns will be considered for our system. Our schema defines two types of features:

- Parent features: Each of these features corresponds to one of the column in the original logs schema. it can be columns such as the source IP address or the executed command as a string.
- Child features: Each of these features is computed from one of the parent features according to some specified function. For example, it can be the hour of the day computed from a timestamp feature or the length of a string computed from a feature with a string type.


Our schema defines for each feature whether it is a parent or child feature and its 'type' so that the system knows how to convert it to a numeric type. This 'type' can be standard data type like int, double, string, boolean for which we define specific ways to convert to double type. The 'type' can also be an abstract data type defined by our system which has some other specific way to be converted to double type. This is used mostly for child features : the abstract data type defines how to compute the feature from the parent feature.

In addition to these parent and child features, two features need to be added : source and destination entities. The source entity is the user/machine/organization performing the action. For network logs, this is usually defined by the host name obtained by reverse DNS from the source IP address or the IP address itself if the DNS resolution failed. For host-based logs, it can be defined by the user id executing the command or the machine or some combination of multiple fields. The destination entity is the target of the action. For network logs we can do the similar reasoning using the destination IP address. For host-based logs, we can define it as the machine on which the command was executed. Since there are many ways to extract entities from the existing columns, we implemented several of them in the system and define the one to use with a parameter. This process will yield two additional features always named "scentity" and "dstentity".

The user then needs to specify according to which entity (source or destination) and which time window the data will be aggregated. The logs entries will be split by their timestamp according to some predefined window of time. For example if the whole data set contains the logs for a day and the time window is two hours, the logs are split into 12 different ranges. Within each range, the logs are grouped by either the source entities or the destination entities using different aggregation functions on each feature. To this end, our schema defines for each feature (in addition to its type and whether it is a parent or child feature) a list of aggregation functions to apply per source or destination entity within each time range. For example, if a feature defines the number of incoming bytes in a connection, we could use the minimum, maximum, mean and sum as aggregation functions. We will define later the different functions supported by the system. After the aggregation, the size of the data is normally reduced: all the logs made by a user during some time range are aggregated to a

single log but this log has more columns since we can define multiple aggregation functions for each feature. So the more logs within each time range and the bigger the time range, the greater the data reduction will be. If the time window is small and we have very few logs per user within each time window, the resulting size might even be larger than the original data size. The figure 1 depicts an example of how features can be aggregated within a time window.

srcip	loginattempts	version	dstport
1.2.3.4	1	2.2	22
1.2.3.4	1	2.2	22
1.2.3.4	2	2.1	22
5.6.7.8	4	1.9	2000
5.6.7.8	5	1.9	22



srcentity	sum(loginattempts)	mostcommon(version)	countdistinct(dstport)
1.2.3.4	4	2.2	1
5.6.7.8	9	1.9	2

Figure 1: Example of logs aggregation within a time window

Once the data is aggregated, we need to scale the values as explained in 2.2.2. We let the user define which technique to use to scale the features : either rescaling each column range between 0.0 and 1.0 or normalizing each row to unit length. This process is applied to all the features except two : the beginning time of the time range and the source or destination entity. So each row describes the activity of an entity during a time range using three main components:

- A beginning timestamp. Since the time window is a constant parameter, the beginning time is sufficient to know the time range of the row.
- An entity. All entities will either be source or destination entities.
- Aggregated features. All following columns of the row are scaled aggregated features describing the activity of the entity during the time range.

This constitutes the final features. Before we persist them, a final step is to extract some statistics (number of rows, minimum and maximum, mean and variance for each column, etc) that are also persisted. This can be useful for faster inquiries about a dataset or for some anomaly detection algorithms that require some of this information: instead of computing these statistics every time on the fly, this preprocessing step during feature extraction allows to read the information in constant time complexity.

This is the standard feature extraction process. In case the user wants to test the system with some recall measurement, there is an extra step : we inject fake intrusions in the logs in order to check how many are detected. We cannot determine how many anomalies there

are in the data since it is unlabeled, but we create our own set of anomalies and mix them with the real data. The system can later evaluate how many of the injected anomalies are detected. This gives us some recall measurement but it should be noted that this is not the real recall measurement.

We define an "intrusion kind" to be a way to inject intrusions in the logs, it defines which columns are affected and how. An intrusion is the set of logs resulting of the application of an intrusion kind. It has a begin and an end timestamps as well as a signature. If the user decides to evaluate recall, the system takes as input a list of intrusion kinds, each one along with the number of intrusions of that kind that should be injected. The system injects the intrusions at random timestamps in the already in-memory read logs (we don't affect the original database) and the process continues in the same way, the final features contain real and injected data. The intrusions computed at run time are also persisted along with their signatures.

3.3 Anomaly Detection Algorithms

In this subsection we will describe the three anomaly detection algorithms that we used. Their concrete implementation will be described in 4.4. Each one of them will produce an anomaly score between 0.0 and 1.0, points with higher scores being more anomalous. Their respective scores are then combined into a single final anomaly score for each data point. The user defines a threshold: points with anomaly scores higher than this threshold are considered anomalies. The user also defines the number of top anomalies he wants to analyze.

3.3.1 Isolation Forest

Isolation Forest[11] is an anomaly detection algorithm trying to isolate outliers with decision trees called isolation trees. It is very similar to Random Forest which is a classification algorithm. The trees are constructed from a training set by splitting the data d times according to some feature values, where d is the depth of the tree. At each splitting, every tree selects randomly a feature and some splitting value in the range of that feature, the two splits are then used as input data for two next recursive splits until one of the following termination conditions occurs :

- The current level of the tree has reached the limit d .
- There is only a single point left in the input data.
- All points in the data have the same values.

In Random Forest, these trees would be applied to the rest of the data to classify the points. In Isolation Forest, we only care about the length of the path of each point in the tree : anomalies are supposed to be rare and far from the other points, thus recursively splitting the data space is supposed to reach a single-point region with much fewer splits for an anomaly than for normal points. Figure 2 shows an example of this principle. Consequently, the anomalies tend to have a shorter path length than other points, this path length is then compared to the average path length in order to compute an anomaly score.

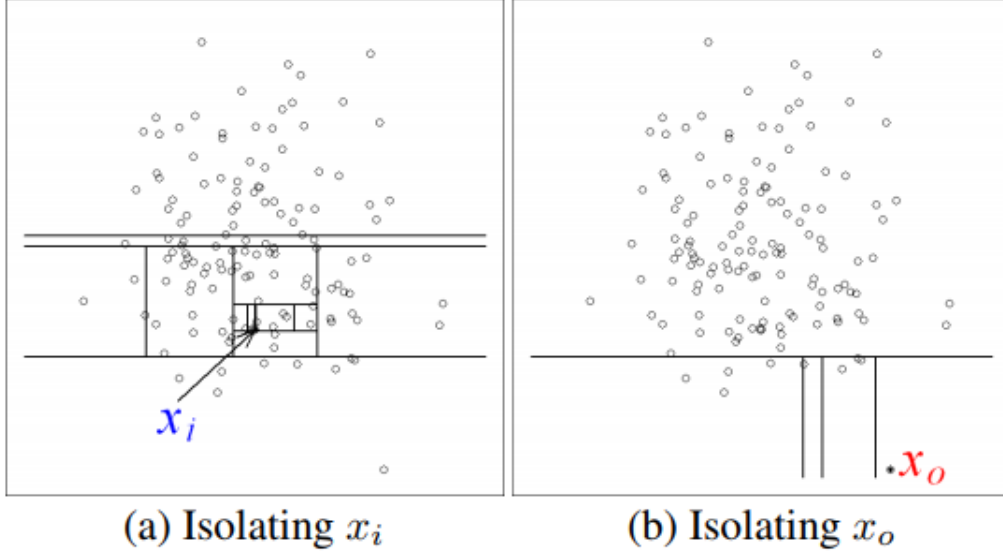


Figure 2: Illustration[11] of how isolating x_i (a) takes more splits than isolating x_o (b).

Isolation Forest has been shown to work better using sub-sampling of 128-256 data points for each tree which results in depth of 7-8 levels. Initially, the algorithm constructs a fixed number of trees. Each one takes a random sample from the data. At each split, every tree selects randomly a feature and a value within the range of that feature. In our case, since we have normalized features, the range is always between 0.0 and 1.0.

Once the trees are computed, the whole dataset is passed through all the trees. Every tree returns a path length for every data point. The different lengths are then averaged to compute a forest-wide path length for each data point. Given a dataset of n points, the average path length of any binary search tree can be computed as follows :

$$c(n) = 2H(n-1) - (2(n-1)/n)$$

where $H(x)$ is the harmonic number of x . It can be estimated by : $\ln(x) + 0.5772156649$. This gives us a way to compare the computed path length over all trees $E(h(x))$ of each point x with the average path length and output some score $s(x, n)$ between 0 and 1:

$$s(x, n) = 2^{-\frac{E(h(x))}{c(n)}}$$

The closer the score is to 1, the more anomalous the data point is considered. The points with scores higher than some threshold are classified as anomalies and kept for the next stages of the pipeline.

So Isolation Forest has only 2 parameters : the number of trees and the numbers of samples to draw from the dataset for each tree construction. Better results are expected with higher number of trees until some point where adding trees doesn't seem to increase precision or recall. The algorithm seems to work better with small samples, the evaluation part of the

original paper suggests optimal results using 100 trees and samples of 256 data points. Our evaluation described in section 5.2 shows similar optimal values.

This algorithm was chosen because of two main reasons : thanks to sub-sampling, the training of the model is very efficient even when dealing with huge data sets like ours and it works using very different techniques than other anomaly detection algorithms so it can be a good complement to clustering for example. It also doesn't suffer from the curse of dimensionality so it can be used in high-dimensional spaces.

3.3.2 K-Means

k-Means is a commonly used clustering algorithm : given a dataset and a predefined number of clusters k , it tries to assign a cluster to every data point in a way that the distance to the center of the cluster is minimum. More precisely, k cluster center points are computed and define the different clusters. Every point is assigned to the cluster of the closest cluster center and the distance between every data point and its cluster center should be as small as possible.

This optimization problem is NP-hard, but K-Means doesn't try to find the most optimal solution : it iterates to update the cluster centers and converges towards a local optimum, but there is no guarantee that this is the global optimum since this is a non-convex problem. Initially the cluster centers are chosen randomly, even if there exist techniques like K-Means++[12] that can choose good initial centers to decrease the number of iterations to converge and to minimize the risks of picking initial values that would lead to a bad local optimum. We compute the distances of every data point to every cluster center to find the minimum distance and assign each point to its cluster. Once we know the cluster of each point, we recompute the center of each cluster by averaging the positions of all the data points inside each cluster. We now have new cluster centers and reassign each data point to the cluster with the closest center. We iterate this process until it converges, meaning that the difference of cluster centers between 2 iterations becomes lower than some error rate ϵ . Computing all these euclidean distances between points at every iteration can become very expensive as the size of the data set becomes very large. So usually, only a subset of the data is used for training the algorithm and computing the optimized cluster centers. Then we assign clusters to the rest of the data.

A limitation of K-Means clustering is the obligation to know the number of clusters beforehand. If a small labeled validation set is available, a solution would be to apply the algorithm with different values of k , observe which gives the best accuracy on the validation set and use this value to run the algorithm on the whole data set. In our case, no labeled set is at our disposal, but the third phase of the pipeline described in subsection 3.4 gives us a way to partially evaluate the precision of the results on a data set and so to pick the best value for k .

Another possible solution to choose the right number of clusters is to use the so-called 'elbow method'. When the number of clusters increases, we decrease the variance within each cluster : the points are grouped closer to each other within smaller clusters. We define the Sum of Squared Errors (SSE) of a given cluster K of size n and center \bar{K} with the following formula:

$$SSE(K) = \sum_{x \in K} (x - \bar{K})^2$$

For the whole data set X constituted of k clusters we can then define the Within-Set Sum of Squared Errors (WSSSE) :

$$WSSSE(X) = \frac{1}{k} \sum_{i=1}^k SSE(K_i)$$

The WSSSE is not proportional to the number of clusters : on a chart like the one shown in figure 3 as an example, we see that it starts by decreasing very fast when we increase the number of clusters. At some point the rate becomes much flatter and the augmentation of the number of clusters doesn't help much to decrease the WSSSE anymore. The number of clusters at that particular point is supposed to be optimal. The angle of the curve at that point looks like an elbow, hence the name of the technique. It is an approximation technique and the point is not always that easy to spot and the neighboring number of clusters should also be considered as potential candidates for the optimal number of clusters. This is a visualization technique, data scientists observe the chart and pick k accordingly. But we wanted a way to use that technique without the intervention of a human such that K-means can be computed in a single run. The rate at which the WSSSE decreases can be computed by considering the difference between iterations. At every iteration i , we can compute if the rate of the WSSSE descent has increased or decreased and by how much between the last two updates :

$$r(i) = \frac{WSSSE_{i-1} - WSSSE_i}{WSSSE_{i-2} - WSSSE_{i-1}}$$

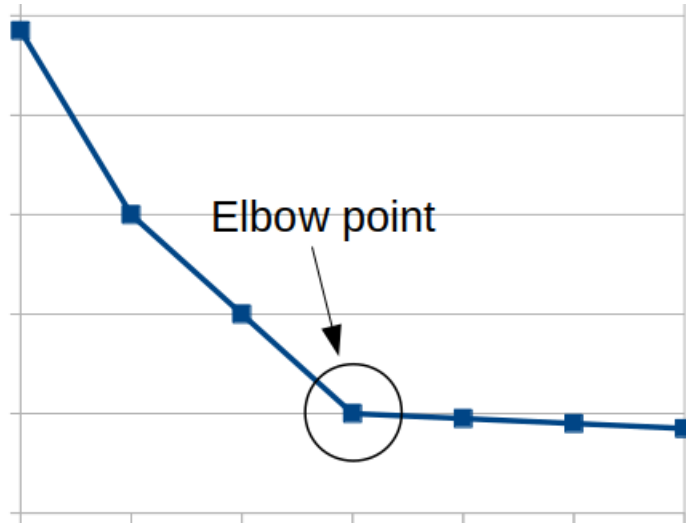


Figure 3: Visualization of the elbow technique

According to the elbow method, this rate should suddenly decrease and fall below some threshold rate. At that point the number of clusters should be optimal. The predefined

number of clusters parameter has now been replaced by three parameters but they can stay the same for any data set :

- The minimum number of clusters from which to start the WSSSE descent.
- The maximum number of clusters until which we compute the WSSSE.
- The ratio of the elbow point. When the rate falls below that ratio, we stop the computation.

So far we have a way to split the data in a number of clusters which can have many applications among which anomaly detection : smaller clusters tend to contain points that are more anomalous. The idea is to assign the cluster size to every point instead of the cluster itself. Then some mapping has to convert cluster sizes to anomaly scores such that small clusters have scores close to 1.0 and large clusters have scores close to 0.0.

We cannot simply rescale the cluster sizes such that the minimum has a score of 1.0 because there might be no small clusters, so normal points in clusters of normal sizes would be considered top anomalies. Similarly, if only the points in the cluster with maximum size have a score of 0.0, the points in the cluster with a slightly smaller size will have a much higher score even though they are not anomalies at all. So we need to have a lower and an upper bounds out of which the scores remain the same : points within clusters that have sizes smaller than the lower bound will all automatically have a score of 1.0, points within clusters that have sizes larger than the upper bound will all have a score of 0.0. Between the 2 bounds, we can rescale linearly the cluster sizes to scores between 1.0 and 0.0.

This algorithm was chosen mostly because it has been shown[13][14][15] to be well suited for anomaly detection. It can require quite heavy computation but a distributed implementation can efficiently process a very large data set as we will see in subsection 4.4.

3.3.3 Local Outlier Factor

LOF[16] is an anomaly detection algorithm using density-based clustering. It has a lot of similarities with DBSCAN described in 2.3.1 but it is especially designed for anomaly detection. It tries to define and compute the local density in different neighborhoods of the data space. Points with low local density compared to their neighbors are considered anomalies. The algorithm starts by computing the k-Nearest-Neighbor (kNN) of each data point. We then need a distance measurement between points, but to get more stable results, the Euclidean distance is not strictly used: All points within the core of a point A (meaning they are its kNN) have the kth nearest neighbor distance from A . This leads us to the following asymmetric distance computation of a point A from another point B called the reachability-distance (rd):

$$rd(A, B) = \max(kd(B), d(A, B))$$

where $kd(B)$ is the distance between B and its kth nearest neighbor and $d(A, B)$ is the Euclidean distance between A and B . Basically, the distance between any point and B is the Euclidean distance but it has to be at least the distance between B and its kth nearest neighbor. Figure 4 shows an example of this concept.

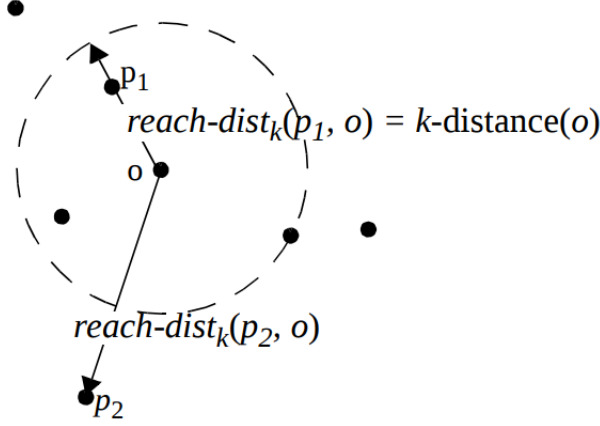


Figure 4: Illustration[16] of the concept of reachability distance in LOF.

We can then compute for each point the average reachability distance from each of its kNN. The greater this average distance is, the lower is the density of this point in this neighborhood. If the distance is low, it means the point is close to its neighbors and thus has a high local density. The local reachability density (lrd) is computed in the following way:

$$lrd(A) = \frac{1}{\frac{\sum_{B \in N_k(A)} rd(A, B)}{k}} = \frac{k}{\sum_{B \in N_k(A)} rd(A, B)}$$

The neighborhood itself can be very sparse or very dense. So we are not interested in the absolute density of the point but rather how dense it is compared to the densities of its neighbors. This comparison yields the final local outlier factor (LOF):

$$LOF(A) = \frac{\sum_{B \in N_k(A)} lrd(B)}{k \cdot lrd(A)}$$

The greater the LOF of a point, the more anomalous it is. The issue is that the LOF works on an open scale but we need scores between 0.0 and 1.0. One possible solution is to find the maximum LOF score computed m and rescale to a range between 0.0 and 1.0 by dividing each LOF score by m . This automatically classifies the point with the maximum score as a top anomaly of score 1.0, this could lead to the same issues as described in 3.3.2. So another solution is to explicitly set as a parameter a maximum score m above which all LOF scores will be mapped to 1.0. This depends on the data set and requires at least some approximate knowledge of the distribution of the LOF scores in the data.

LOF has been chosen mostly because multiple studies[4][17] show it gives very good results for network-based IDSs. As a density-based clustering algorithm, it is also a good complement to K-Means which is a distance-based clustering algorithm. LOF suffers from a similar disadvantage as K-Means : the number of nearest neighbors to consider has to be

defined beforehand. We didn't try to come up with some optimization technique to pick a good value for k , it is a parameter of the algorithm that needs to be set. The first step of LOF, the kNN computation, is another disadvantage since it is very expensive. However, we found a way to implement an approximate distributed solution that will be described in 4.4.3.

3.3.4 Ensemble Techniques

Every detector assigns anomaly scores between 0.0 and 1.0 so we can combine their scores in a fair way since all the detectors scores have the same weight. We decided to use symmetric and associative binary operators to combine detectors by pairs of 2. This allows to build a final score for a set of detectors such that the score is the same no matter in what order we consider the detectors. These functions must also produce output scores between 0.0 and 1.0. Usually the mean is used to combine scores, but the maximum can also provide good results[8]. These are the two functions that were implemented but the system is designed to support any operator that matches the following conditions :

- binary : the operator must take 2 scores as input.
- closure : the output must also be a score between 0.0 and 1.0. Along with the previous condition, this can be formalized this way : $op : [0, 1] \times [0, 1] \mapsto [0, 1]$
- commutativity : $op(s_1, s_2) = op(s_2, s_1)$.
- associativity : $op(s_1, op(s_2, s_3)) = op(op(s_1, s_2), s_3)$.

Once the final score is computed, the data can be sorted according to the final anomaly score and the top n anomalies are persisted and will be shown to the security analyst where n is a user-specified parameter.

3.4 Results Inspection

As described in 3.2, each one of the persisted anomalies describes the activity of an entity during some time range with scaled features. The anomaly detection step adds an anomaly score as a last column. So the security analyst cannot directly investigate by looking at the anomaly which consists solely of scaled features numbers that make sense only for a machine learning system and not a human. Therefore, we need to reconstruct the original logs of that entity during that time range so that the security analyst can determine whether these logs constitute a security incident or not.

For each anomaly, we determine the end timestamp of the time range from the begin timestamp and the duration of every time range. In the feature extraction step, we computed the entity from one or multiple columns of the original logs schema, we will now reverse the process: from the entity value we compute the values of the different columns involved in the entity. For example if the entities are computed from the IP addresses and the port numbers separated by a colon ("145.0.32.1:80"), we extract the two values "145.0.32.1" and "80" along with the names of the columns to which they belong. We now have a way to query again the original database to fetch the logs that match exactly the different attributes

making the entity and falling within some time range. We can thus obtain for each anomaly the set of corresponding original logs that the security analyst must inspect. Figure 5 shows an example of how we can reconstruct the corresponding logs from the anomaly, the end timestamp and the logs database.

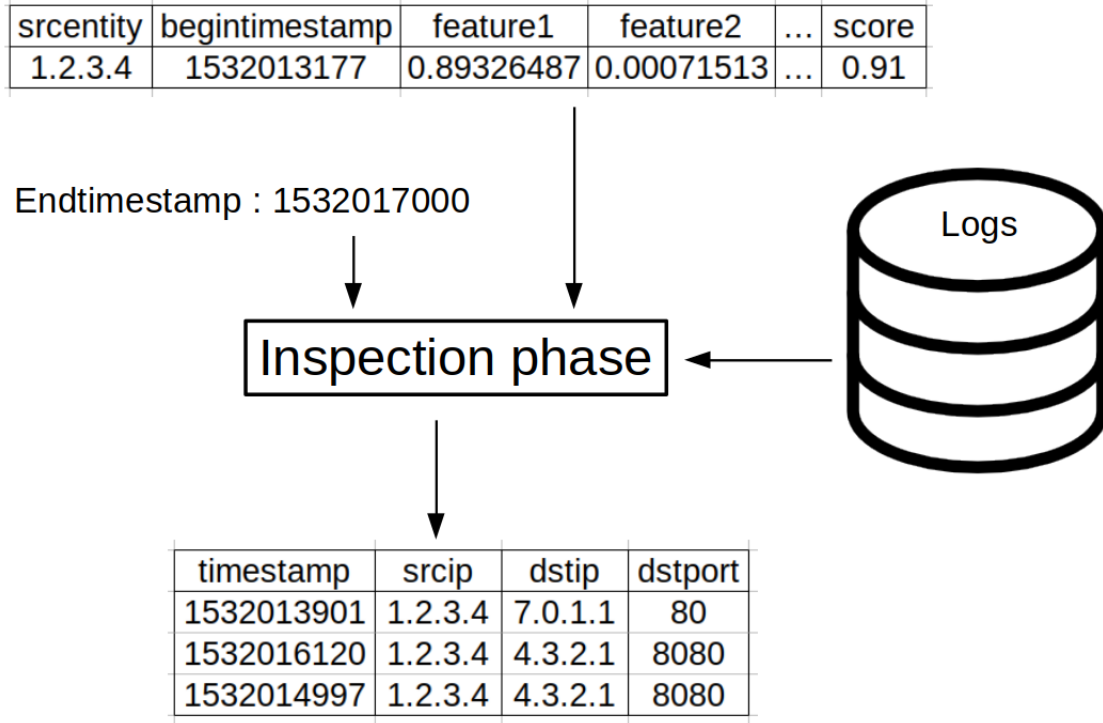


Figure 5: Illustration of the logs reconstruction process

If we retrieve 100 anomalies and each of them contains 10'000 logs, going through all of them is a very long and tedious task for a human security analyst. In addition, the anomaly detection algorithms need to be tuned. Usually the algorithm is tested against a labeled validation set to compute precision and recall for each parameter. In our case we don't have any labeled data, so since this work cannot conceivably be entirely done by a human, the system must provide a way to automate as much as possible this process. For each data source, there must be a set of rules that can determine if a log or a set of logs constitute an anomaly. Each rule can be of two different types:

- Simple rule: it analyzes each log entry of the anomaly independently and checks if it matches some predicate.
- Complex rule: it analyzes all log entries of the anomaly with an aggregation function and checks if the result of the aggregation matches some predicate.

This predicate reads one or multiple column value(s) and compares it with a predefined value using a boolean function. For example, a rule can check if the destination port of a SSH connection log entry is 22 or not. Another example would be a rule computing the sum of

failed login attempts across all log entries and check whether it is below some threshold or not.

This set of rules is configurable and every parameter of every rule can be changed. For each anomaly, the system applies every rule to the corresponding reconstructed set of logs. The simple rules are applied for every log entry of the anomaly and the complex rules are applied to the whole set of log entries. Each rule defines a comment that describes why a particular log or set of logs was/were flagged as anomalies. The original log schema is extended with two additional columns of string type: "anomalytag" and "comment". The "anomalytag" field is always empty except for the first row where it can take two values: "yes" to denote that the anomaly is a true positive according to the rules or "?" to denote that the system was unable to determine if the anomaly is a true or false positive because no rule matched it. When a simple rule finds a log entry that matches, it adds its comment in the "comment" field and updates the first "anomalytag" field with a "yes" value if it doesn't already have that value. When a complex rule finds that the set of logs match, it appends its comment to the "comment" field of the first rows and also updates the first "anomalytag" field with a "yes" value if it doesn't already have that value.

The precision can then be computed as the number of anomalies with a "yes" value in the "anomalytag" field of the first row out of the total number of anomalies detected. It should be noted that if an anomaly is not flagged by any rule, it doesn't necessarily mean it is a false positive. The rules set is not exhaustive and in such cases the security analyst must make decisions on every anomaly that was not flagged to obtain the true precision measurement, the precision computed only from the rules set is only a lower bound.

The precision measurement is printed on standard output and the reconstructed logs annotated by the rules are persisted to the database. The security analyst can now read these logs assisted with the comments created automatically by the rules to better understand why some logs are considered anomalies. The security analyst can perform further investigation and incident response accordingly.

In the case where the user decided to compute recall as well, the system will evaluate how many intrusions were detected. The different anomalies initially retrieved after the second step described in 3.3 contain each one field for the entity. The intrusions computed in the features extraction step all have distinguishable entity values following a pattern: "dummy*i*" where *i* is a number identifying uniquely each intrusion. By looking at the entity value, the system can separate anomalies from the real data and the ones from the injected data. The process of logs reconstruction is done in the same way for the two types of anomalies. For the injected anomalies, we compute the signature of the logs and compare with the signatures of the intrusions computed in the features extraction step. The number of signature matches determines the number of injected intrusions that were detected. This number out of the total number of intrusions that were injected gives us some recall measurement.

4 Implementation

In this section, we will describe in more details how the system was implemented, using which technologies and how the user can interact with it. The entire project is open-source and the code is available online[18].

4.1 Technologies and Resources used

In section 3, we talked about the database storing the logs in an abstract way. Concretely, the database is not a standard SQL relational database but structured files on HDFS[19]. HDFS (Hadoop Distributed File System) allows to store very large files on a distributed cluster. We used the CERN analytix cluster, its characteristics are described in 5.1. The files are replicated to provide fault-tolerance and HDFS provides scalable read and write access to these files. Since the logs data can reach more than 100 GB per day for only one data source, a standard relational database would not have been possible. The logs for each data source are stored within a separate folder which contains a meta-file for the schema. The files themselves are structured to correspond to the logs schema, they are stored and partitioned in sub-folders according to the day of the logs. For example if a data source stores the logs in folder `ds`, the logs for May 1st 2018 will be stored in `ds/year=2018/month=05/day=01`. This partition scheme allows faster search through the data like a normal indexing technique.

Originally the files were stored in JSON[20] format. This format is not compressed at all and is row-store. This works for small amounts of data and is well suited for OLTP queries. But in our case, we have very large data sets and the algorithms perform mostly OLAP queries on the different columns. So to compress the data and allow faster analytical queries, we switched to a column-store format : Apache Parquet[21]. The transformation is done using Apache Kite[22] which can convert JSON files to Parquet files if provided with the schema. It can also automatically partition the logs into the different sub-folders according to their date.

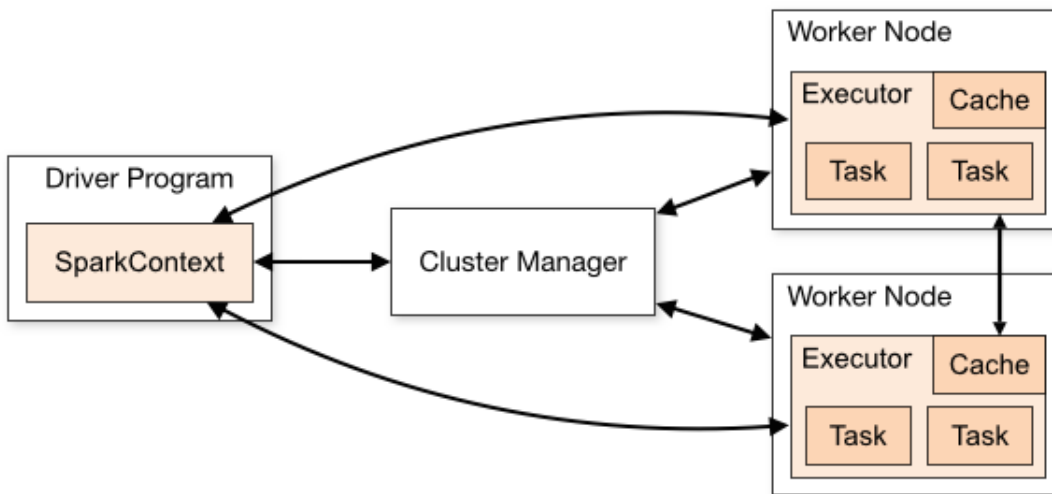


Figure 6: Overview of the architecture of Spark clusters

The whole processing of the data stored on HDFS is done using Apache Spark[23], which is a distributed in-memory processing engine for large data sets. The data is split in small parts called partitions that are distributed across a cluster of nodes that will process each of these partitions and exchange data. These nodes are called the executors. They are controlled by one or more driver node(s) from which the jobs are launched. The driver node dispatches tasks and code among the executors and awaits the results back from the executors. Figure 6 shows an overview of this distributed architecture. We implemented the IDS in Scala[24] since it is faster than other languages in Spark because it is the original language Spark was built in. We also made use of an extension of Scala called Scalaz[25]. It provides a core library for extra functional programming features like sum types and monads. Since we are implementing machine learning algorithms, we also made use of the standard Spark machine learning library : Apache Spark MLlib[26]. Although it turned out that most of the anomaly detection techniques were either not implemented or not efficient enough.

For installation and deployment, we used RPM Package Manager[27]. It allows to package binaries and define dependencies and installation procedures. We used Koji[28] to distribute this package to users from a server available inside CERN. We will discuss it in more details in 4.7.

4.2 Architecture

The system provides a command line interface with four commands triggered manually by the user. Each one is a keyword to be placed after the main installed command named `cert-anomaly-ids`:

- `extract` : Extract the features from the logs according to the schema, scales them and stores them as described in 3.2.
- `detect` : Applies anomaly detection algorithms on the features, combines the scores and stores the detected anomalies as described in 3.3.
- `inspect` : Inspects the anomalies to reconstruct the original logs and determine if they are true positives by using a rules set as described in 3.4.
- `optimize` : Finds the best parameters for a given detector. This command isn't strictly part of the pipeline. It should be used to tune a detector before applying it.

Each of these commands takes parameters that are defined in `conf/application.conf` and most of them can be overridden with command line flags. `conf/application_loader.conf` defines the path and file name of the configuration (`conf/application.conf` by default). The user can override it with his own `.conf` file as long as it defines every parameter like in the default `conf/application.conf`. Parameters that can be set directly in the command line as well have a consistent name and can be set by placing `--param <value>` or `--param=<value>` after the command keyword for a parameter named `param`. Some parameters have a single letter flag that can be used as well with `-p <value>` or `-p=<value>`. We will go through every parameter in the following subsections.

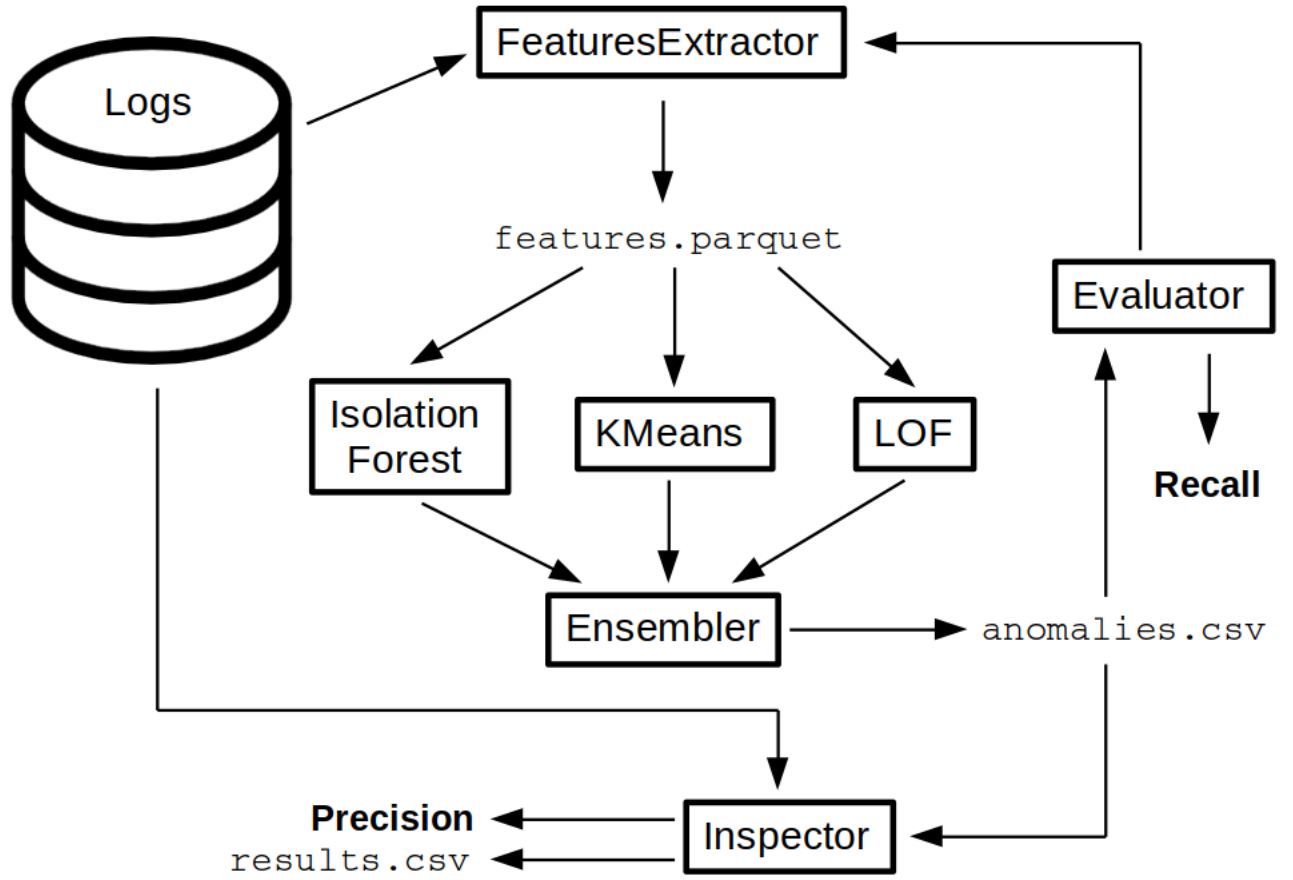


Figure 7: Architecture of the IDS and its components

The system is organized as described in Figure 7 with seven main components which correspond to the most important classes in the code:

- **FeaturesExtractor:** This class provides functions to read the logs stored as Parquet files in HDFS and treat them as DataFrames in Spark. It also creates the different features from the columns, the source and destination entities, aggregates according to the time window, scales the features and persists them to HDFS as Parquet files.
- **IsolationForest:** This class extends the abstract Detector class and as such provides an implementation of the 'detect' abstract method to append anomaly scores to every entry in an input DataFrame using the Isolation Forest algorithm.
- **KMeansDetector:** It also extends the abstract Detector class but this time uses the K-Means algorithm to detect anomalies.
- **LOFDetector:** This class extends the same abstract Detector class and provides an implementation of the LOF algorithm.

- **Ensembler:** This class provides functions to combine the anomaly scores computed by different instances of the Detector class using different ensemble techniques. It persists the final anomalies with their scores to a CSV file.
- **Evaluator:** This class is responsible for the recall evaluation : it creates and injects the intrusions during the features extraction phase, and it evaluates how many intrusions were detected in the results inspection phase.
- **Inspector:** This class provides functions to inspect the anomalies persisted to a CSV file during the detection phase and create results CSV files. It loads the anomalies, reconstructs the original logs for each anomaly by fetching the Parquet files again from HDFS and applies the different rules to compute the precision measurement and add automatically add comments to the log entries. The final annotated logs are persisted to CSV files.

4.3 Feature Extraction

In this subsection, we will describe in details how feature extraction was implemented, the different steps of the processing and all the parameters than can be tuned by the user to extract features differently.

4.3.1 Basic Features

The extraction of features starts by parsing the features from the schema defined as a JSON file to create a list of instances of the Feature class. Each Feature object has a name and optionally its parent's name. These names must match the names of the columns in the logs schema from which we will extract the values. Every Feature also has a function to parse all the values of the corresponding column in the extracted DataFrame to numerical values in Double data type. For columns storing values with types like Boolean, Integer or Long, the conversion to Double is pretty straightforward. For the String type the idea is to assign a Double to each different String. Mllib provides a class called StringIndexer: it goes through the data and for each String it looks up in a table if there is already a Double assigned to it, if yes it uses that Double to convert the String, otherwise it increments some Double counter and creates a mapping between the new String and the new Double value. Evaluation showed that maintaining this in-memory lookup table and the use of RDDs[29] instead of DataFrames[30] results in poor performance when dealing with huge datasets. So we decided to implement our own version of StringIndexer: HashStringIndexer. The idea is to hash each String using some hashing function that provides a good distribution to minimize the risk of collisions even with very large datasets. This has two advantages compared to the StringIndexer from the standard library : it does not require additional space like a lookup table and since the processing of each String is independent, it can be processed in parallel for each partition without any data transfer between the nodes. Regarding the hashing function itself, cryptographic hashing functions would provide a good distribution but their computation is expensive and the resulting hash being usually between 128 and 256 bits, it could not fit in a single 32 bits Double value. We used the djb2[31] hashing function which was especially designed for hashing strings. We also implemented parsing functions

to extract the day or hour as a Double from the Long timestamp or the length of a String as a Double. The complete list is provided in 4.3.4. In addition to the parsing function describing how to convert to Double, each Feature also has a list of aggregation functions to be used within each time range and that are described in more details in 4.3.2.

Once we have the list of Features, the FeaturesExtractor knows which columns of the logs it should read from HDFS. Then it will create the source and destination entities using an EntityExtractor. Each instance of this class has a list of required columns and a function to create two new columns from these required columns : "srcentity" and "dstentity". At the same time it also adds two new Features instances for these two new columns. We define later the different EntityExtractors that were implemented.

If the user wants to compute the recall measurement, we inject intrusions at this stage using the Evaluator class. We parse a schema as a JSON file that defines IntrusionKinds. Each instance of IntrusionKind defines a function to inject fake logs into the extracted DataFrame, meaning that we do not affect the original logs in HDFS. The function returns the new DataFrame and an Intrusion. The Intrusion is the result of the injection by an IntrusionKind and contains several informational fields such as its IntrusionKind, the name of the fake source entity of the Intrusion, the beginning and end timestamps, and a signature which is computed from the fake logs. The signature computation is done by an Accumulator. Accumulators are Spark objects capable of accumulating an initialized value using a binary operator applied at each row. The standard library provides Accumulators to sum Integer or Double types: the counter is initialized and each partition on each executor node can increment the counter concurrently and the driver node can request the counter value. Building on this mechanism, we extended the Accumulator class to create an Accumulator-Sign class. It maintains a sorted list of rows hashed with SHA256. The final signature is computed by hashing again the concatenation of all the hashes. The Intrusions are then serialized on the driver node disk such that they can be read again during the results inspection phase to compute the recall measurement. Finally, we can apply the parsing function of every Feature to obtain a DataFrame with columns that all have a Double data type. The resulting features are called raw basic features.

4.3.2 Traffic Features

The next step is to aggregate the values within each time range. To this end, we compute the minimum and maximum timestamps values across all logs and use the parameter for the duration of each interval to know the beginning and end timestamps of each interval. We can then split the DataFrame into n DataFrames according to the timestamp value of each log, where n is the number of time intervals. We can create a list of aggregation functions across all columns from the aggregation functions of each Feature instance. For values that were initially already numerical types, we generally use a list of common aggregation functions like the mean, sum, maximum and minimum. For values that were initially Strings, we can use the number of distinct values as an aggregation function. We can also determine the most common value encountered as an aggregation function. This last one was the only one not provided by the Spark standard library. In Spark, you can define your own UDAF (User Defined Aggregate Function) by extending the UserDefinedAggregateFunction class, this is how we implemented the MostCommonValueUDAF class which uses a frequency map

to determine the most common value in a set. Then for each DataFrame we group by either the source entity column or the destination entity column and aggregate using all the aggregation functions. Finally we merged the n aggregated DataFrames back into a single DataFrame containing the intermediate features known as raw traffic features.

4.3.3 Features Scaling

The last step is to scale the features so that the different machine learning algorithms can process them better and give better results that are not biased towards a specific feature. As described in 2.2.2, we have two ways of doing that : either we rescale every feature range between 0 and 1 or we normalize each row such that the sum is 1. In the first case, we need to compute the minimum and maximum values for each column and then apply the rescaling formula to every value in the column. The computation of the minimum and maximum values are expensive in Spark and the overall computation requires two passes on every column of the entire DataFrame so this technique is not recommended for performance issues. In the second case, each row can be processed individually : we compute the norm of the row and divide each value by the norm. Since the computation of each row is independent, this process can be parallelized by partitions and requires a single pass on each row which makes it much faster since no data transfer between nodes is required. For these reasons, we used this second technique in every evaluation of the system. The final scaled features are then written as Parquet files back to HDFS. We also compute some statistics on the final features as explained in 3.2 and persist them to another Parquet file in HDFS.

4.3.4 Parameters

We are now going to describe the formal parameters involved in the features extraction phase. The following parameters are global parameters meaning they affect the whole pipeline and not only the feature extraction phase:

- `--loglevel` : Defines the log level as a string according to the log4j levels[32].
- `--featuresschema` : Defines the path and name of the JSON file of the schema that describes the different columns, their types, aggregation functions, etc... Appendices A and B show two examples of such features schemas. A schema has 2 fields: `name`, which simply defines the name of the schema, and `features` which is an array of features where each feature element has the following structure :

```
1 {  
2   "name" : "featurename",  
3   "parent" : "parentfeature"|null,  
4   "type" : "Datatype",  
5   "doc" : "Description of the feature",  
6   "aggs" : ["agg1", "agg2", "agg3", ...]  
7 }
```

- **name**: Features can be either 'top' features that are columns in the original logs schema (in which case this field must match one of the columns' names), or they can be computed from another feature in the schema.
- **parent**: In the case the feature is computed from another feature, this field defines the name of that parent feature. In the case the feature is a 'top' feature this field is null.
- **type**: Describes the data type of the feature so that we know how to parse it to Double. Currently supported values are : ["Boolean", "Int", "String", "Long", "Host", "Day", "Hour", "Head", "Length", "PpidBinary"]. The "Host" type should be used when the column defines a FQDN, "Day" and "Hour" should be used when the column contains UNIX timestamps in milliseconds as Longs. The "Head" type extracts the first word in a string made of words separated by spaces, this word is then converted to Double. The "Length" type extracts the length of a string. The "PpidBinary" resolves the name of the binary of the parent process identifier (PPID) in host-based data sources.
- **doc**: Relevant documentation for the feature.
- **aggs**: List of aggregation functions to use to extract traffic features in the extract phase. Possible values are : ["mostcommon", "countdistinct", "mean", "sum", "max", "min"].
- **--extractor**: Defines the name of the entity extractor to use (how to define the source and destination entities based on the fields). Every entity extractor needs the original logs schema to contain some specific columns in order to be used. They are specified by the "Requires" list for each of the following already implemented entity extractors :
 - **hostsOnly**: Use only the hostname given by reverse DNS, NOT_RESOLVED if it couldn't be resolved. Requires: ["srchost", "dsthost"]
 - **ipOnly**: Use only IP addresses. Requires: ["srcip", "dstip"]
 - **hostsWithIpFallback**: Use hostname and fall back to the IP address if it couldn't be resolved. Requires: ["srchost", "dsthost", "srcip", "dstip"]
 - **hostsWithCountryFallback**: Use hostname and fallback to the country from GeoIP if it couldn't be resolved. Requires: ["srchost", "dsthost", "srcip_country", "dstip_country"]
 - **hostsWithOrgFallback**: Use hostname and fall back to the ISP if it couldn't be resolved. Requires: ["srchost", "dsthost", "srcip_org", "dstip_org"]
 - **uidOnlyAndHost**: In the case of host-based data sources, use the user ID of the process as source entity and the hostname of the machine as destination entity. Requires: ["uid", "host"]
 - **uid0AndHost**: If the user ID is 0 (root) it is not a reference to the responsible user. If the user ID is between 1 and 999, it is a local user ID and not a global LDAP[33] user ID. So if the user is 0 we define the source as a combination user ID, PPID and hostname of the machine. If the user ID is between 1 and 999 we use the

user ID and the hostname of the machine. In other cases, we use only the user ID. The destination entity is still only the hostname of the machine. Requires: ["uid", "ppid", "host"]

- `--interval`: Defines the size of the time window for the aggregation and the logs reconstruction. It uses the Scala duration notation[34] so "60 min", "1 day" or "8 hours" are all valid parameter values.
- `--trafficmode`: This parameter defines whether aggregation is performed per source or destination entity. It can take 2 values : "src" or "dst".
- `--help`: Prints all the parameters, their usage and a small description in the console.

The following parameters can be placed after the `cert-anomaly-ids extract` command:

- `-l, --logspath`: Input path for the logs. The logs must be stored in parquet format. The only requirement regarding columns is to have a timestamp column. The entity extractors might require additional columns. The path accepts wild cards or any other format specific to the file system used. So for example in HDFS the following are all valid: `todaylogs.parquet`, `januarylogs/*`, `logs/year=2018/*/*/*`, `logs/year=2018/month=0{2,3}/*/*`.
- `-s, --scalemode`: Defines how the features are scaled in order to be used by machine learning algorithms. Currently it accepts two modes:
 - `unit`: Every row is normalized such that the sum of all the fields is 1.0.
 - `rescale`: Every feature column is rescaled such that the range is [0.0, 1.0].
- `-f, --featuresfile`: Defines the path and name of the file where to write the features to. The extension is .parquet and is added automatically to the path. So `featuresfile =myfeatures` will create `myfeatures.parquet`.
- `-s, --featuresstatsfile`: Defines the path and name of the file where to write the statistics of the features to, according to the describe function of DataFrame. These numbers can be useful for some detectors.
- `-r, --recall`: Must be set to true if we are testing some model and want to inject intrusions to compute recall in the inspect step, false otherwise.
- `-i, --intrusions`: Defines the path and name of the JSON file that describes the intrusions, the number of instances for each type of intrusion. Appendix D contains an example of such an intrusion schema. This parameter has no effect if `recall=false`. The intrusions schema file has two fields : `name`, which simply defines the name of the intrusions schema, and `intrusions` which is an array of intrusions kinds where each intrusion kind element has the following structure:

```

1 {
2   "name" : "intrusionkindname",
3   "doc" : "Description of the intrusion kind",
4   "requiredcolumns" : ["col1", "col2", ...],
5   "number" : 3
6 }

```

- `name`: The name of the intrusion kind which must be already implemented with that name. For now there is only one implemented intrusion kind: `"tooManyAuthAttempts"`, which creates some connections of a fake source host, each one with a very high number of login attempts.
 - `doc`: Relevant documentation for the intrusion kind.
 - `requiredcolumns`: List of columns of the original logs schema that must be present because the intrusion kind will use these fields.
 - `number`: Defines how many intrusions of that kind should be injected in the logs.
- `-d, --intrusionsdir`: Defines the path of the directory where to persist the computed intrusions and their corresponding fake logs. This parameter has no effect if `recall=false`.

4.4 Detection

Every anomaly detection algorithm has to be implemented in a class extending the abstract Detector class: it provides a single abstract method to detect anomalies on a DataFrame of features. It takes the DataFrame as input and appends a new column of Double with the anomaly scores for each row. The DataFrame is filtered such that only the rows with scores higher than some fixed threshold are kept.

4.4.1 Isolation Forest

The standard MLib library doesn't provide any implementation of Isolation Forest and the only open source implementation[35] cannot handle large data sets efficiently. It has to count the number of rows at run time and relies on RDDs for the data processing. It could not have been easily improved and integrated into our system so we implemented our own Isolation Forest from scratch.

The IsolationForest class extends the Detector class and implements the Isolation Forest algorithm. As explained in 3.3.1, we need to compute the average path length in every tree from the number of samples in the dataset. Counting the number of elements in a DataFrame can be very expensive when dealing with very large data sets, especially if we have to count again every time we try the algorithm with different parameters to tune the precision measurement. This is when the statistics that we computed in the features extraction phase are useful: the IsolationForest detector reads the number of samples from the statistics file that was persisted to HDFS.

IsolationForest starts by creating a number of isolation trees. A parameter can set the

number of trees to create. Each tree is implemented by the `IsolationTree` trait which has two concrete subclasses `ExNode` and `InNode` to represent leaf and intermediates nodes in the tree. Each `InNode` has two `IsolationTree` children and the root node represents the whole tree. The two classes provide implementation of a function to compute the path lengths of a subset of rows in a recursive way. Every tree is built using a different training set which is a subset of the dataset. The user can choose how many subsamples are taken from the dataset for each training set. At each level of the tree, we randomly pick a feature and a value to split the current set into two different parts and create the children nodes according to the algorithm. As the training sets are small (normally 256 samples), we can collect them entirely in the driver node and compute the trees entirely in the driver node. We use multi-threading to compute trees in parallel to accelerate the process.

Once the trees are built, we can broadcast them across all the executor nodes of the cluster and apply their path length computation function on every row of the data. This can be parallelized at the partition level. According to the algorithm described in 3.3.1, we should average the different path lengths computed by the different trees for a same row and compute the final anomaly score according to the scoring formula. But we can avoid a lot of unnecessary computation since we only want to retain the rows above the anomaly threshold t . So we can reverse the scoring function to determine the threshold below which we consider rows with average path $E(p)$ anomalies :

$$s = 2^{-\frac{E(p)}{c}} \geq t \Leftrightarrow E(p) \leq -c \cdot \log_2 t$$

Instead of computing the average path across all trees for each row, we can compute only the sum of paths and update our formula to know the sum of paths threshold given that $E(p) = \frac{\sum_{i=1}^n p_i}{n}$ where p_i is the path length of tree i and n is the number of trees :

$$E(p) \leq -c \cdot \log_2 t \Leftrightarrow \sum_{i=1}^n p_i \leq -nc \cdot \log_2 t$$

This gives us a way to filter rows according to their sum of path lengths across all trees compared to the fixed threshold $-nc \cdot \log_2 t$. Then we can compute the final scores only for the rows we are interested and return the new `DataFrame`.

4.4.2 K-Means

The `MLlib` library provides an efficient implementation of the K-Means algorithm described in 3.3.2. we wrapped the `KMeans` class from the library in a `KMeansDetector` class. The `KMeans` class expects a single feature column of `Vector of Double` type and a number of clusters to train the model using a subset of the data and provide clusters. Then we can apply the model to the rest of the data to determine to which cluster each point belongs. So we start by assembling our columns into a single column of vectors using the `VectorAssembler` tool from `MLlib`. Then we determine the number of clusters either from an explicit parameter value or from the elbow technique described in 3.3.2. We take a fraction of the data as train set and run the K-Means algorithm. Once the model is computed, we extract the sizes of the different clusters and map sizes to anomaly scores using the lower and upper bounds as explained in 3.3.2. Finally we filter the `DataFrame` to keep only rows with a score higher than the threshold and return it.

4.4.3 Local Outlier Factor

The MLlib library doesn't provide an implementation of LOF. There exists an open source implementation[36], however it has several issues : it doesn't actually implement LOF but what is called simplified-LOF (when we use the k -distance instead of the reachability distance as explained in 3.3.3) and this simplified technique is supposed to produce worse results. This implementation also suffers from performance issues since it relies on RDDs, doesn't take advantage of parallelization at the partition level and uses expensive operations like "groupBy" and "reduceBy". For these reasons, we decided to build our own implementation of LOF.

The LOF algorithm starts by computing the k nearest neighbors for each point in the data set. The most common way to achieve that is to iterate over every point p in the data and compute the distances to every other point in the data from p and maintain the k points nearest to p . This approach has a quadratic time complexity with respect to the number of points and in high-dimensional spaces computing the euclidean distances can itself be expensive. When dealing with very large data sets, this approach cannot be used and it is preferable to use other techniques like locality-sensitive hashing (LSH)[37]. The k nearest neighbors of any point are all within the same neighborhood, so if we could split the data into different subsets according to their position in the data space, we could do the intuitive search with quadratic time complexity on a much smaller and local dataset. As opposed to cryptographic hash functions, locality-sensitive hash functions try to preserve the original distances between points, meaning that close points will obtain the same hash or very close hashes and points far from each other will obtain very different hashes. So using such a function, we can assign a hash to every point in the dataset that will represent the neighborhood of the points. We can then partition the points by their hash so that each partition contains points lying in the same local area. The computation of the k nearest neighbors can then be performed independently in parallel in each partition.

There exists different ways to implement a LSH function[38]. For nearest neighbor search, a family of LSH functions is usually used instead of a single LSH function. This allows points of the dataset to fall in different potential neighborhoods. The local search for the nearest neighbors is performed in every potential neighborhood of the point and the results are then merged across different neighborhoods. However, since we want to have a completely independent processing in each partition, this scheme cannot work in our case so we built our own LSH function.

If we simply sum up all components of rows, rows that are close in the data space will have similar sums since their components have similar values. But vectors that are very far from each other can also have very close sums if they have the same components values but in a different order. So if we assign coefficients to every component and ensure that these coefficients are co-prime, the resulting sums will still be very similar for vectors close to each other, but vectors far from each other are less likely to have a similar sum even if they have similar components in different orders because these same components values will be multiplied by different coefficients. We can repeat this process with different coefficients and average the results to minimize the probability of having far vectors with similar hash values. Formally the hash function for a row r in a d dimensional space works as follows :

- Create a set A of k d -dimensional vectors with randomly chosen prime components.
- For every $a_i \in A$, compute the dot product with r : $s_i = a_i \cdot r = \sum_{j=1}^d a_{ij} \cdot r_j$.
- Average the computed sums to produce the final hash : $h = \frac{1}{k} \sum_{i=1}^k s_i$

We now have a hash as a single Double value for every row and we want to re-partition the data such that rows with similar hashes fall within the same partition. To achieve that we need to map close hashes to a same hash value that can act as a key for rows to be in the same partition. We can round down the last x digits of every hash, this way close hashes will have the same hash value to be grouped by together. For example, if row $r1$ has a hash of 17456.6767761509 and row $r2$ has a hash of 17456.6767764731, we can round down the last 4 digits and they will both have the hash value 17456.676776. If row $r3$ has a hash value of 17456.6767747012, its truncated hash value will be 17456.676774 so $r1$ and $r2$ will be in the same partition but $r3$ will be in a different one.

This number of digits x can be used as a parameter to choose the trade-off between correctness and performance. When x is large, the number of distinct hash values is small and as a consequence so is the number of partitions. Rows are grouped together in a few large partitions so there is a high probability that the k nearest neighbors of any row are within the same partition so we will produce correct results. On the other hand, large partitions affect performance since it doesn't distribute well the load across the executors of the cluster. When x is small, the number of distinct hashes is large so the rows are put in small groups that have very close hash values and the number of partitions is very large. This implies that there is a high probability that the k nearest neighbors of many rows are split between different partitions, so the resulting k nearest neighbors of the local search within a partition might not be correct for many points. But the local search will be fast since the number of rows in the partition is small and the overall computation will be efficient since we take advantage of distributing the data across the executors and process it in parallel.

We can now implement k -nearest neighbor search within a partition. We create a two-dimensional distance matrix between the rows of the partition. Then for each row we iterate over the other rows and maintain the k rows with the minimum distance to the fixed row according to the distance matrix. For each new encountered row, we check if the distance is smaller than the maximum distance found in the current k nearest neighbors. If it is the case, we can replace the neighbor with the largest distance with the new encountered row. We could maintain the nearest neighbors seen so far in a list of size k but finding the maximum element would take $O(k)$ time. We instead use a max-heap, provided by the PriorityQueue class, to provide access to the maximum in $O(1)$ time and remove it in $O(\log(k))$ time.

The rest of the LOF algorithm can also take place within the same partition and produce anomaly scores on an open scale. Then the LOFDetector class can map the scores obtained by the LOF implementation on a range between 0 and 1 as discussed in 3.3.3, either by using a predefined max score to will be mapped to 1, or computing the maximum obtained score at run time and map it to 1.

4.4.4 Ensemble Techniques

Since every implemented detector extends the Detector class, the Ensembler can use polymorphism to provide functions to combine detectors without knowing their concrete implementation. Each detector applies its algorithm on the same DataFrame to add a score column. Then the Ensembler combines the scores column two by two into a single score column using either the mean or the maximum binary operators. Since these operators have the properties described in 3.3.4, we can repeat this process and obtain a single column score for the DataFrame no matter the number of detectors and in which order they are applied. These final scores are then sorted in descending order and we take the n top rows of the DataFrame. These rows are the final anomalies and they are stored in a CSV file. Every line in the CSV file is an anomaly with the following fields: the source or destination entity, the timestamp at which the anomaly began, the different scaled features values and finally the anomaly score.

4.4.5 Parameters

The following parameters apply to the detection phase of the pipeline so they can be placed after the `cert-anomaly-ids detect` command and are not specific to any detector:

- `-f, --featuresfile` : Defines the path and name of the file where to read the features from. The extension is `.parquet` and is added automatically to the path. So `featuresfile=myfeatures` will try to read from `myfeatures.parquet`. The same parameter is defined for features creation in extract but in `conf/application.conf` it is a single parameter.
- `-s, --featuresstatsfile` : Defines the path and name of the file where to read the statistics of the features from. Some detectors use these informations such as the number of rows or the min and max of each columns to speed up the processing. The same parameter is defined for features creation in extract but in `conf/application.conf` it is a single parameter.
- `-d, --detectors` : Defines the detectors to be used for anomaly detection in the following format : `d1,d2,d3,...` where `d1`, `d2` and `d3` are the names of the detectors. Isolation Forest has the name `iforest`, K-Means `kmeans` and LOF `lof`.
- `-t, --threshold` : Defines the threshold score value between 0.0 and 1.0 above which aggregated rows are considered anomalies.
- `-n, --nbtopanomalies` : Defines how many anomalies should be persisted, starting from the anomaly with the highest score and going down in the sorted scores of the detected anomalies.
- `-a, --anomaliesfile` : Defines the path and name of the file where to write the anomalies to. The extension is `.csv` and is added automatically to the path. So `anomaliesfile=myanomalies` will create `myanomalies.csv`.

- `-e, --ensemblemode`: Defines the ensemble technique used to combine the scores of the different detectors. The value can be either `mean` or `max`.

The following parameters concern the IsolationForest detector. They cannot be set directly via the command line, they can only be changed in the configuration file :

- `nbtrees`: The number of IsolationTrees to use.
- `nbsamples`: The number of samples to use to build an IsolationTree.

These are the parameters for the K-Means detector and are also modifiable only through the configuration file:

- `trainratio`: The percentage value between 0.0 and 1.0 of the data to be used as a training set.
- `minnbk`: The minimum number of clusters when using the elbow technique.
- `maxnbk`: The maximum number of clusters when using the elbow technique.
- `elbowratio`: Defines the ratio between 0.0 and 1.0 at which the WSSE should decrease 'fast', when the ratio hits a value below this threshold, the current number of clusters is used.
- `nbk`: The number of clusters to use in case we want to set it explicitly, should be -1 if we want to use the elbow technique instead.
- `lowbound`: Defines the cluster size below which points are considered anomalies with 1.0 score.
- `upbound`: Defines the cluster size above which points are considered normal with 0.0 score.

The LOF detector has the following parameters changeable only through the configuration file as well:

- `k`: The number of nearest neighbors to consider.
- `hashnbdigits`: The number of digits after the decimal dot to keep for each hash. A low number means a low number of different hashes so fewer and larger partitions which means slower computations but more likely to have the kNN inside the partition. A large number means a large number of different hashes so more and smaller partitions which means faster computation inside each partition but more likely to have the kNN across different partitions.
- `hashnbvectors`: number of vectors to create for the dot products with each row.
- `maxscore`: LOF score that will be mapped to 1.0. Should be -1 if this score should be computed as the maximum score.

4.5 Inspection

Explain how the logs are reconstructed from the anomalies, how they are compared to intrusions to compute recall and how the different rules are applied to compute precision. Describe all the different parameters that can be changed.

The Inspector class implements the reconstruction of the logs from the anomalies. If the user wants to evaluate recall, we call the Evaluator class to inspect the intrusion detected. Finally, the Inspector applies rules to the logs to evaluate precision and help the investigation of the security analyst. We are going to see these three different steps in details as well as the parameters that can be changed regarding the inspection phase.

4.5.1 Logs Reconstruction

The Inspector reads the logs from the same location in HDFS that was read by the FeaturesExtractor in the extraction phase. It also reads the anomalies CSV file created by the detection phase. For each anomaly, it extracts the source or destination entity and the beginning timestamp. We want to retrieve all original logs from HDFS corresponding to that anomaly. Since we know the value of the interval parameter for the time window used in the extraction phase, we can compute the end timestamp t_e from the beginning timestamp t_b and the interval duration d : $t_e = t_b + d$.

As seen in the extraction phase, the source or destination entity might be constituted by more than one column. For example, if we use the hostname and fall back to the IP address in case the reverse DNS resolution was not possible, in the same "srcentity" column, we have both hostnames and IP addresses. When we want to retrieve the logs corresponding to a particular entity, we need to know on which original column to select the entity value. In our example, we either select on the "srcip" column or "srchost" column. This is why every EntityExtractor provides also a reversing function: for any value of the source or destination entity, it returns the SQL clause to fetch the entity from the original columns. In our previous example, the EntityExtractor uses pattern matching to determine if the source value "v" is a hostname or an IP address. In the first case, it will return "srchost=v" and in the second case, it will return "srcip=v". We could have a clause with multiple equality statements, for example if the destination entity is made of the IP address and the port number separated by a colon, for a value like "192.168.200.1:80", the EntityExtractor should return "srcip=192.168.200.1 AND srreport=80".

So for an entity e and a beginning timestamp t_b , we are able to construct a SQL statement with the following form:

```
"SELECT col1,col2,...,coln FROM logs WHERE timestamp  $\geq$   $t_b$  AND timestamp  $\leq$   $t_e$  AND  $C(e)$ "
```

where $t_e = t_b + d$ is the end timestamp and $C(e)$ is the clause returned by the reversing function of the EntityExtractor in use.

Using such SQL statements, we can fetch the original logs for every anomaly. We add a new column "id" to label the logs according to the anomaly they belong to. We also add two columns "anomalytag" and "comments" of String type that are initially empty. They will

be used in the rule-based precision evaluation. The original logs are returned as a list of DataFrames.

4.5.2 Recall evaluation

If the user sets the parameter for recall evaluation, we will split the anomalies between the real anomalies found in the original dataset and those corresponding to injected intrusions. We can easily differentiate them by looking at the source or destination entity : in the case of a fake anomaly, the entity is named with "dummy*i*" where *i* is the identifier of the intrusion created in the extraction phase. The real anomalies are treated as explained above. We reconstruct the logs of the fake intrusions by looking at the fake logs created in the extraction phase. Those logs were persisted in a different location of HDFS to distinguish them from the real logs. The logs reconstruction process is the same as the one for the real anomalies. They are also returned as a list of DataFrames and they are then passed to the Evaluator. It loads the Intrusions computed in the first phase and for each DataFrame, the Evaluator computes the signature of the logs as described in 4.3.1 and looks for an Intrusion with the same signature that wasn't already detected by another signature. It then marks the Intrusion as detected and increments the number of Intrusion detected. Finally, the Evaluator is able to output how many Intrusions were detected out of the total number of Intrusions injected, which gives our recall measurement.

4.5.3 Precision evaluation

Once we have the reconstructed logs of the real anomalies as a list of DataFrames, we will process each DataFrame individually in order to compute a precision measurement. We apply a set of rules on the logs and implement three classes to represent the two different types of rules described in 3.4. The abstract Rule class defines a way to analyze logs of an anomaly and determines if they match some condition that implies that the detected anomaly is a true positive. The Rule class has 2 subclasses : SimpleRule and ComplexRule. A SimpleRule is a Rule that considers each log entry of the anomaly individually, if at least one of them matches the condition, the anomaly is considered a true positive. A ComplexRule is a Rule that considers all logs of the anomaly to determine if they match the condition. It has an accumulator to compute a value across all logs and then evaluate that resulting value with the condition. We will describe later the different rules that were implemented.

Every Rule has a set of required columns to be able to operate. We first check that the DataFrame has all these required columns and then broadcast the rules to every node of the cluster. We can then apply the rules on every partition. For every SimpleRule, since the processing is independent between rows, we can process in parallel the rows in every partition. For every row, the SimpleRule checks if some columns matches some condition in which case it sets the comment field with some rule-specific fixed String. For every ComplexRule, we cannot process each partition individually since the results depends on all the logs across the different partitions, so we use an accumulator and decide at the end if the value of the accumulator matches the condition of the ComplexRule. At the same time for every Rule, we accumulate a boolean value to determine if at least one row was flagged by the Rule. If it is the case, the anomaly tag field of the first row is set to "yes" and to "?" otherwise to

express that we cannot determine based only on the rules if the detected anomaly is a true positive or not.

Once we applied the rules on every DataFrame, we can check the anomaly tag field of the first row of every DataFrame. If it is set to "yes", we increment the number of detected number that are true positives according to the rules. We can then easily compute the precision by dividing by the total number of anomalies detected. This precision measurement is printed to the standard output for the user information. The logs of every anomaly along with the comments created by the rules are then persisted to CSV files. We create a separate results folder for every anomaly which contains one or more CSV file(s) depending on the number of logs corresponding to that anomaly to avoid huge CSV files that are slow to open and read.

4.5.4 Parameters

The following parameters apply to the inspection phase and can be placed after the inspection command (`cert-anomaly-ids inspect`):

- `-a, --anomaliesfile` : Defines the path and name of the file where to read the anomalies from. The extension is `.csv` and is added automatically to the path. So `anomaliesfile=myanomalies` will try to read `myanomalies.csv`. The same parameter is defined for anomalies detection in detect but in `conf/application.conf` it is a single parameter.
- `-r, --rules` : Defines the path and name of the JSON file that describes the inspection rules and their parameters. Appendix C contains an example of such a rules schema. A rules file has two fields: `name`, which simply defines the name of the rules schema, and `rules` which is an array of rules where each rule element has the following structure :

```
1 {  
2   "name" : "rulename",  
3   "params" : ["param1", "param2", ...],  
4   "text" : "comment text"  
5 }
```

- `name` : The name of the rule which must be already implemented with that name. See below a list of already implemented rules.
- `params` : Defines a list of rule-specific string parameters for the rule. They might be filenames or integers that must be parsed in the application or a specific text to pattern match on, etc...
- `text` : In case the rule finds a match, defines the text to add in the comments column to describe why the log was flagged.

The following rules are already implemented, the string in quotes represents the name of the rule:

- `"ssh_auth_attempts"`: Checks if the number of attempts in a SSH connection is higher than some value defined by the first parameter in `params`.
- `"ssh_dstport"`: Checks if the destination port in a SSH connection is not 22. No parameters are needed.
- `"ssh_version"`: Checks if the version number in a SSH connection is less than the version number described by the first parameter.
- `"ssh_srcip"`: Checks if the source IP address belongs to a list of known malicious IP addresses. The first parameter should contain the file path and name where these addresses are stored, one per line.
- `"ssh_dsthost"`: Checks if the destination hostname is very uncommon by looking in a file of most common destination hosts. The first parameter should contain the file path and name where these hosts are stored, one per line in the following format `hostname|||count` where `count` is the number of times this `hostname` was seen in some log dataset used to compute these statistics.
- `"ssh_client"`: Checks if the client software used for the SSH connection is very uncommon by looking in a file of most common client softwares. The first parameter should contain the file path and name where these clients are stored, one per line in the following format `client ||| count` where `count` is the number of times this `client` was seen in some log dataset used to compute these statistics.
- `"ssh_server"`: Similar to `"ssh_client"` but this time for server software.
- `"ssh_cipher"`: Checks if the cipher algorithm used for the SSH connection is very uncommon by looking in a file of most common ciphers. The first parameter should contain the file path and name where these ciphers are stored, one per line in the following format `cipher ||| count` where `count` is the number of times this `cipher` was seen in some log dataset used to compute these statistics.
- `"ssh_total_auth_attempts"`: Checks if the number of attempts across all SSH connections in the anomaly is higher than some value defined by the first parameter in `params`.

Note that the statistics files for `"ssh_srcip"`, `"ssh_dsthost"`, `"ssh_client"`, `"ssh_server"` and `"ssh_cipher"` are not provided in the code repository as they contain sensitive and environment-dependant data.

- `-i, --inspectionfiles`: Defines the path and name of the files where to write the anomalies to. If the amount of logs per anomaly is not too large, there will be one file per anomaly, otherwise some anomaly logs might be split in different files. The extension is `.csv` and is added automatically to the path. So `inspectionfiles=myinspection` will create files like `myinspection-part0001.csv`.
- `-d, --intrusionsdir`: Defines the path of the folder where to read the injected intrusions and their logs from in case we want to compute recall. This parameter has no effect if `recall=false`. The same parameter is defined for intrusions injection in extract but in `conf/application.conf` it is a single parameter.

4.6 Optimization

The optimization command isn't really part of the pipeline. It takes a detector name as input and finds the best parameters for that detector on some specified dataset by doing a grid search on predefined ranges of parameters. The `Optimizer` class takes a list of Detector as input and a corresponding list of values for a same parameter. The `Optimizer` applies each Detector on the same data and outputs the precision measurement for each parameter value. The companion object of the `Optimizer` class provides functions to generate different kinds of lists of Detectors to be used as input for `Optimizer` instances. We describe below which generators functions were implemented. For each Detector, we can construct one or more `Optimizer` instance(s) by using the different functions to generate lists of Detectors. The user can choose which Detector to optimize but cannot configure how the optimization is done.

The `cert-anomaly-ids optimize` command accepts the following parameters:

- `-d, --detectoropt` : Detector to optimize. So far it can take 3 values:
 - `iforest` : Tries different number of trees by steps of 10 between 10 and 200. Also tries 128, 256, 512 and 1024 samples.
 - `kmeans` : Tries different number of clusters between 5 and 35.
 - `lof` : Tries different number of nearest neighbors between 4 and 10.
- `-f, --featuresfile` : Defines the path and name of the file where to read the features from. The extension is `.parquet` and is added automatically to the path. So `featuresfile =myfeatures` will try to read from `myfeatures.parquet`. The same parameter is defined for other commands but in `conf/application.conf` it is a single parameter.
- `-s, --featuresstatsfile` : Defines the path and name of the file where to read the statistics of the features from. The same parameter is defined for other commands but in `conf/application.conf` it is a single parameter.
- `-t, --threshold` : Threshold between 0.0 and 1.0 above which logs are considered as anomalies.
- `-n, --nbtopanomalies` : Number of top anomalies to evaluate.
- `-r, --rules` : Defines the path and name of the JSON file that describes the inspection rules and their parameters.

4.7 Deployment

The whole program can be packaged as a single `.jar` file using `sbt assembly`. We choose to install every component of the program on Linux machines in a `/opt/cert-anomaly-ids` folder. We can then run the `.jar` file using the `spark-submit` command. To install the `cert-anomaly-ids` command, we create a script of the same name calling `spark-submit` with the `.jar` file. We also create an installation script to create an environment variable pointing

to our installation and add it to the `PATH` environment variable. This installation script can be placed in `/etc/profile.d` so that it is executed at startup.

Using RPM, we can automate this process by providing a specification file. It defines the different sources to package together (the `.jar` file, the binary executable and the installation script), the dependencies to install and the commands to run to install the program on the machine. All this information can be packaged into a `.rpm` file. Then we can use Koji which is an RPM-based building system to distribute the `.rpm` file. CERN has a Koji server that can host RPMs and distribute it to clients.

5 Evaluation

5.1 Settings

We used the analytix cluster for evaluation as well. It has the following characteristics:

- Total RAM : 7.39 TB
- Total number of virtual cores : 1296
- Total number of nodes : 46
- HDFS version : 2.6.0-cdh5.7.6

We used mainly one data source for testing and evaluation purposes: Bro-SSH. It contains one log entry per SSH connection that was made or attempted from CERN to the Internet or vice-versa. Each log entry contains details of the SSH connection such as the source and destination hostnames, IP addresses and ports, the protocol version, the number of login attempts, the time, etc... The complete schema can be found in the Bro documentation[39]. We used the month of April 2018 to evaluate our system for Bro-SSH, it has a total size of 673.5 MB for 21'449'178 records. We will call this dataset D_B .

Another data source was used only for testing purposes : the execution logs (ExecLogs). This data source contains significantly more data: between 65 and 75 GB per day. It was used to verify the system could scale given to a much larger data set, but creating correct features for this schema, tuning the parameters of the algorithms and evaluating the results would have taken too much time. So the evaluation of the results were done only for Bro-SSH. However we still evaluated the time performance of the system for ExecLogs without evaluating the results using the data of May 1st 2018 which has a total size of 65.5 GB for 1'156'594'641 records. We will call this dataset D_E .

These data sources are real unlabeled logs from CERN users activities. We cannot use an external labeled validation set to evaluate the results since the data would not reflect the real data from CERN. So we cannot strictly evaluate precision and recall, we used the techniques described in the Design section to compute some precision and recall measurements.

5.2 Performance

In this subsection, we will evaluate the time and space performance of the different phases of the pipeline. Each value is a mean of five independent computations.

5.2.1 Features Extraction

We evaluated the time taken for features extraction for the two datasets using three different time windows as well as the space taken in HDFS by the resulting final features. The features schema for D_B contains 19 different features and can be found in appendix A. The features schema for D_E contains 20 features and can be found in appendix B.

Time window	Bro-SSH	ExecLogs
1 hour	338	388
4 hours	230	103
24 hours	117	112

Table 1: Time to extract features in minutes

As expected, the time taken decreases when the time window increases. The basic features extraction step is not affected by the time window parameter but the aggregation step is. The larger the time window is, the smaller the number of intervals to aggregate is. The resulting dataset is also smaller when the time window is large so the features scaling step has a smaller input and is thus faster.

We also see that the time taken is not correlated with the size of the dataset. This is because the cluster uses dynamic allocation: the number of executors assigned to the Spark job depends on the size of the dataset and the processing power required. So the difference between a small and a large dataset might not be significant because the large dataset gets a lot more computing power.

Time window	Bro-SSH	ExecLogs
1 hour	801.2 MB (118%)	6.7 GB (10%)
4 hours	236.4 MB (35%)	6.3 GB (9%)
24 hours	58.5 MB (9%)	4.6 GB (7%)

Table 2: Storage Size of the features (% compared to original size)

The space taken by the features decreases also when the time window increases because we aggregate over a larger number of rows for each entity to replace them by a single row. The space taken for the features of D_B is larger than the initial dataset when we use a time window of one hour. Even with aggregation this is possible when the number of rows per hour and entity is very small and when the number of columns increases. This is indeed the case for D_B because we consider most of the columns and for each one create multiple columns according to the different aggregation functions.

5.2.2 Anomaly Detection

We used D_B to evaluate the performance of the three anomaly detection algorithms using different combinations of values for the different parameters of each algorithm. The dynamic allocation in the cluster makes the trials with different parameters settings run in a similar amount of time since the more computationally expensive settings get more resources allocated.

Number of trees / subsamples	Time
50 / 128	7.66
100 / 128	5.99
100 / 256	7.13
200 / 256	6.45

Table 3: Time to detect with Isolation Forest in minutes

Theoretically, in Isolation Forest the number of samples used to build each tree is supposed to increase the time taken to build the trees but table 3 suggests that the training step is not significant compared to the time taken to apply the trees on the whole dataset. The number of trees should affect the time taken to process the data but the parallel execution seems to make the computation time insensitive to the number of trees used, at least when it is between 50 and 200.

Number of clusters / Training ratio	Time
10 / 0.5	10.87
10 / 1.0	9.71
20 / 0.5	11.85
20 / 1.0	10.75
30 / 0.5	11.30
30 / 1.0	9.86

Table 4: Time to detect with K-Means in minutes

The K-Means cluster assignment of points, as shown in table 4, in the test set seems to be more computationally expensive than the training of the clusters centers because the algorithm takes less than when we consider the whole dataset as a training set instead of just half of the dataset. The numbers of clusters at this scale doesn't really affect the computation time.

Number of nearest neighbors	Time
3	6.85
5	7.31
8	6.85
10	8.32

Table 5: Time to detect with Local Outlier Factor in minutes

The LOF computation time shown in table 5 seems correlated with the number of nearest neighbors. This is because the processing of every data point loops over the neighbors at the different steps of the algorithm and is thus more expensive when it needs to consider a large set of neighbors.

5.2.3 Results Inspection

We now measure the time taken to reconstruct the logs and apply the different rules on the detected anomalies with different number of anomalies.

Number of anomalies	Time
1	1.16
10	2.60
20	6.74
50	18.17
100	42.12

Table 6: Time to inspect the anomalies in minutes

As expected, the time taken increases when the number of anomalies to inspect is larger because the anomalies are processed in an independent way. We execute one SQL statement per anomaly to retrieve the corresponding logs and we apply the rules on each set of logs.

5.3 Precision

The CERT is more concerned by false positives rather than false negatives. Getting a lot of false positives might cause what is sometimes called 'alarm fatigue': when the security analyst sees so many alerts raised which shouldn't have been raised for most of them, he/she ceases to analyze the alerts carefully and the system might become useless. On the other hand, given the size of the CERN network traffic and the variation of activity among its users, we can never expect to flag every anomaly. Letting slide some anomalies is fine as long as we flag most of the top anomalies. This is why we focused on precision and not recall. For each of the algorithms, we evaluated precision on the top 200 anomalies detected on D_B using the rules described in appendix C and different values for the parameters of the algorithms.

5.3.1 Isolation Forest

We measure the precision of the Isolation Forest for different values of the two parameters (number of trees and number of samples). The default value for each parameter when iterating over the other parameter is the one recommended in the original Isolation Forest paper : 100 trees and 256 samples.

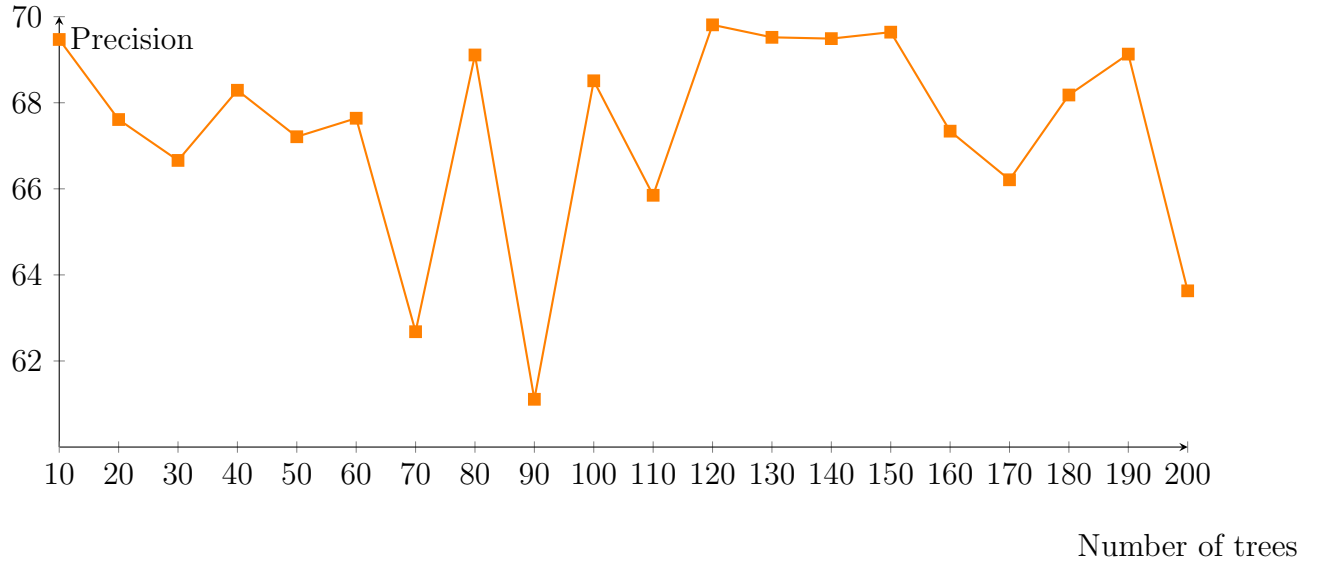


Figure 8: Precision for different number of trees and 256 samples.

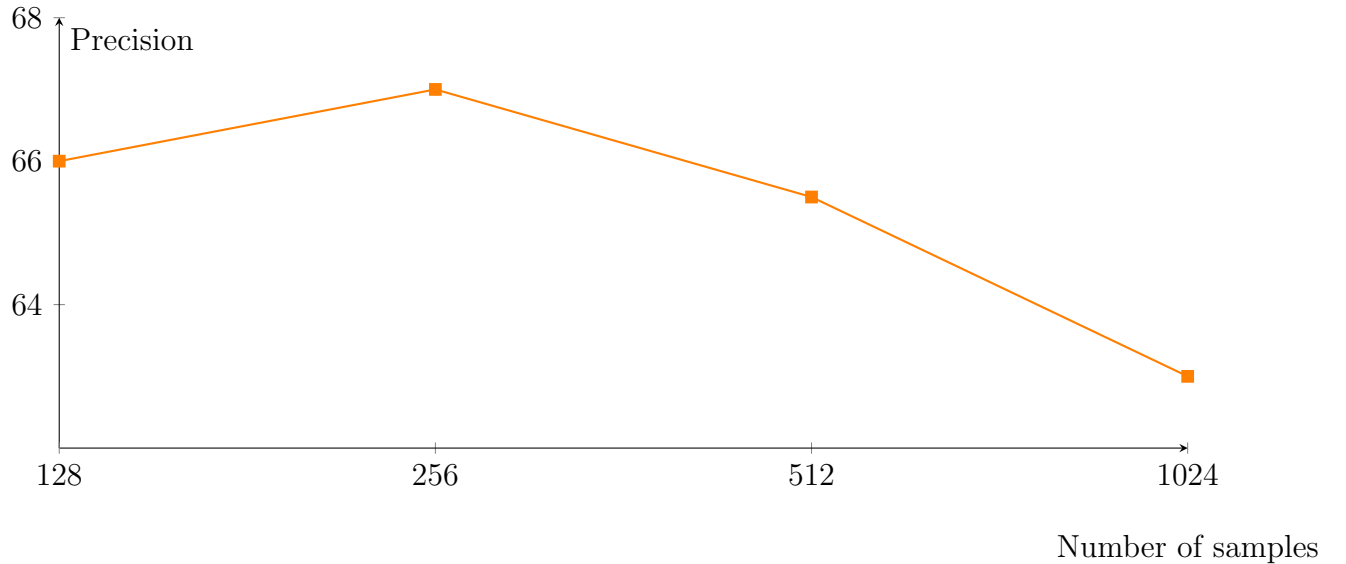


Figure 9: Precision for different number of samples and 100 trees.

The optimal number of samples in our evaluation seems to match the one recommended in the Isolation Forest paper. However, the precision is the highest when using 120 trees instead of 100 trees.

5.3.2 K-Means

We now measure the precision of the K-Means algorithm when trying different number of clusters. The lower and upper bounds of the cluster sizes for the score computation stay fixed to 10 and 200'000.

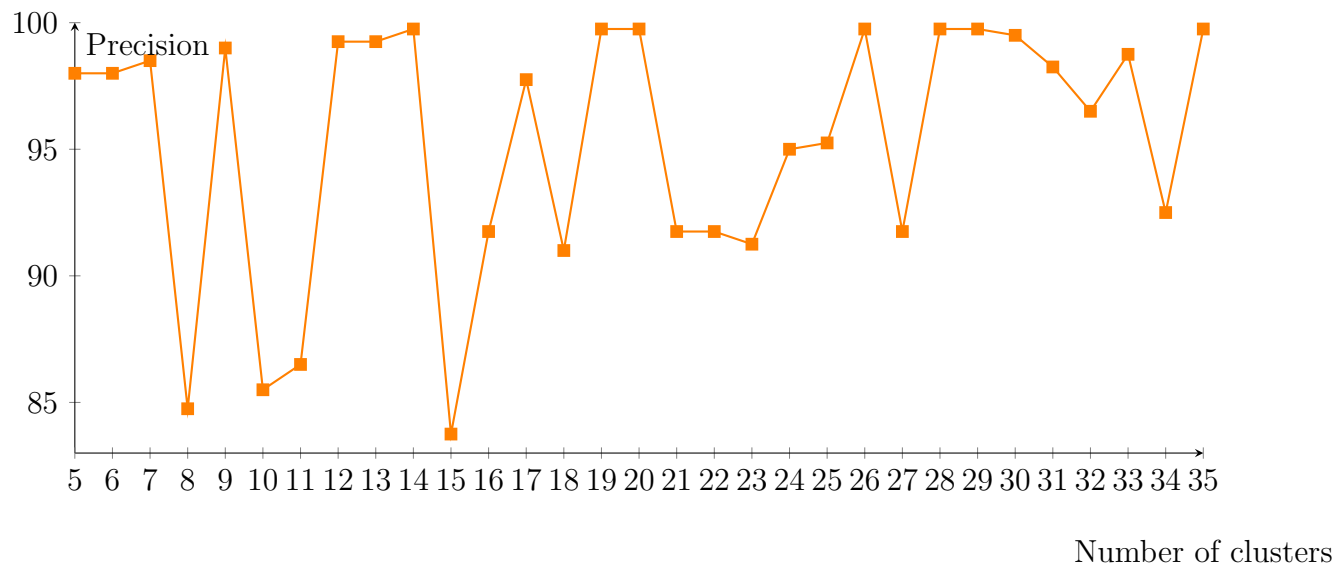


Figure 10: Precision for different number of clusters.

Since choosing the best number of clusters in K-Means is a non-convex optimization problem, it is not surprising that we don't see any clear global optimum. We chose 20 clusters as our preferred number of clusters since it is one of the best values for precision and is small enough to provide good time performance.

5.3.3 Local Outlier Factor

We try now to optimize the precision of the LOF algorithm using different values for the number of nearest neighbors to consider for each point.

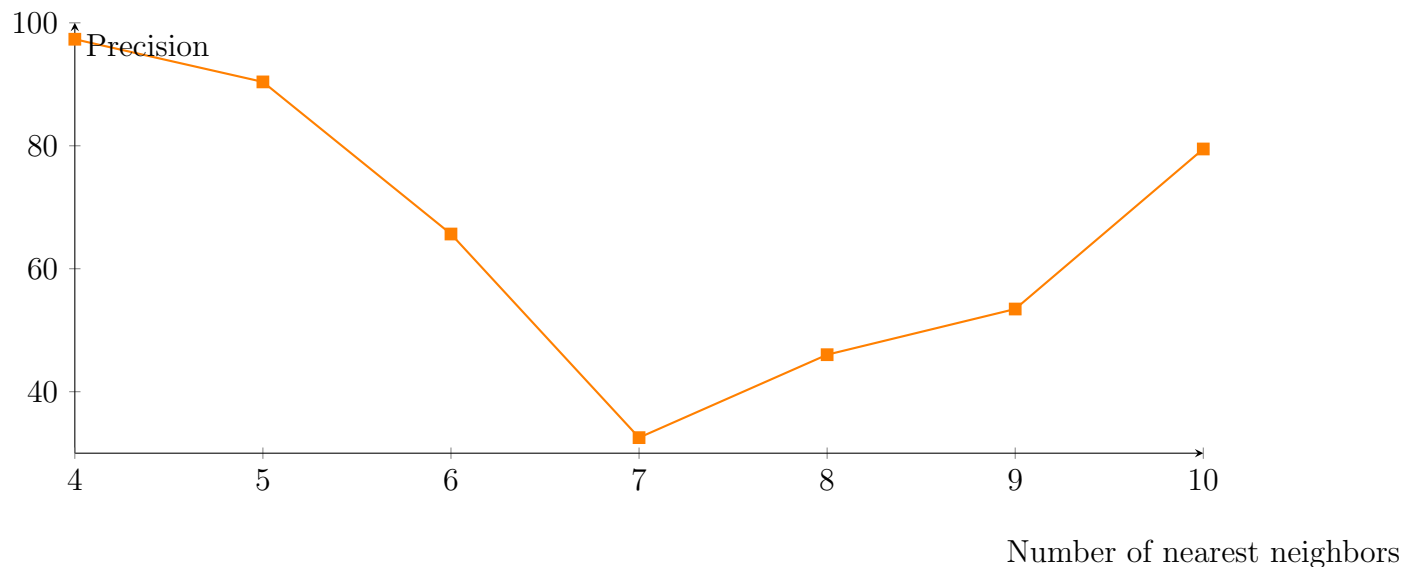


Figure 11: Precision for different number of nearest neighbors.

The algorithm seems to perform the best when considering four neighbors for each data point so that is the one we used as our preferred parameter value. It is also small enough to be computationally efficient.

5.3.4 Ensemble Techniques

Now that we have chosen optimal values for the parameters of our three different algorithms, we try all possible combinations of them using the two different ensemble techniques and compare the precision obtained for the top 100 anomalies.

Detectors	Combinator	Precision
iforest+kmeans	max	100%
iforest+kmeans	mean	86%
kmeans+lof	max	95%
kmeans+lof	mean	100%
iforest+lof	max	91%
iforest+lof	mean	55%
iforest+kmeans+lof	max	100%
iforest+kmeans+lof	mean	88%

Table 7: Precision for different combinations of detectors.

Overall, the ensemble technique based on the maximum seems to provide better results than the technique based on the mean. When taking the maximum operator, the top 100 anomalies contains the anomalies with the highest scores across the different algorithms, meaning they are the ones that the algorithms are the most confident classifying them as true anomalies. Since K-Means alone provides higher precision than Isolation Forest and LOF, the combinations involving K-Means seem to provide better results.

5.3.5 Anomalies detected on CERN SSH traffic

On the D_B dataset, we did a manual investigation of some of the results. The comments added automatically by the rules helped us to spot more easily the cause of each anomaly. We found mostly two different types of anomalies on this data source during the month of April 2018:

- Brute force attacks on SSH servers : 29 different IP addresses were identified. An online investigation of some reporting websites[40][41] showed that most of these IP addresses were already reported to be trying the same attacks on other enterprise networks.
- SSH tunneling from the CERN network to SSH servers outside CERN running on a port different than 22 : the IP addresses of 9 different SSH servers were identified. We cannot say that these are security incidents but running on a port other than 22 is definitely an anomaly. We also often spotted a tenth SSH server running on a non-standard port but this time inside CERN and it was already commonly known to be running on another port for some reason.

We probably could have spotted even more of these on the month of April 2018 by investigating more of the results files and by running the IDS on the same data more times.

6 Conclusion

This anomaly-based IDS shows as a proof of concept that we can detect intrusions and security incidents using anomaly detection algorithms. Several anomalies on our evaluation data from the CERN SOC were detected. Evaluations of the three algorithms that were implemented already existed to show how they could detect anomalies on research datasets. But we used only real up-to-date unlabeled data from the CERN SOC. We showed how it is possible to provide some recall and precision evaluation even without labeled test sets by injecting fake intrusions and by using sets of rules. It is crucial to be able to evaluate machine learning algorithms for intrusion detection without requiring labeled data because in most cases there is no data set corresponding to the real logs produced by the enterprise SOC. Using research data sets that are often outdated might not provide accurate evaluation and labeling real data by hand can be a huge effort.

When most research in anomaly detection is done on datasets of a reasonable size, this IDS based on Big Data technologies like Spark and HDFS addresses the issues of running these algorithms on much larger data sets. It shows that it is possible for anomaly detection algorithms to handle very large amounts of logs and still detect anomalies accurately.

The IDS allows the security analyst to tune different parameters for each algorithm and to use different data sources, features, intrusions or rules. as such, the IDS is intended to be very flexible to correspond to the environment and data of the users. It is important to be able to adapt the IDS because if some parameters work well on a particular dataset, it doesn't necessarily mean that they will still work well on another dataset. As an open source project, it was also built to allow users to easily contribute by implementing new rules or ensemble techniques, etc... This way the IDS could handle more data sources and provide better results using more algorithms and more features.

The increasing number of cyber-attacks makes it important for SOCs to not rely solely on signature-based IDSs to protect the enterprise network and machines. This anomaly-based IDS can complement the signature-based IDSs of the CERN SOC and increase the security.

7 Future Work

7.1 Detection Scheduling

At this stage, the user has to manually run the different commands of the pipeline. This process can detect anomalies on some dataset, for example a month of data for some network protocol. For the next month, the user will have to manually run the commands again. Some future work could implement some scheduled triggering of the IDS scans on a regular basis. This could be done using the distributed scheduler Nomad[42] which is already in use at CERN. The system requires some large amount of data for the algorithms to learn and detect anomalies and the time to gather this amount of data might be much larger than the desired interval between two scans. For example, we could train the system on two months

of data and run it every week. This would mean that the algorithms need to learn again most of a dataset that they have already explored. Updating a machine learning model with new data without re-training it from scratch is still a very new topic of research but some future work might explore this area.

7.2 Evaluation

We evaluated our three implemented algorithms using different parameters values but we didn't have time to try all possible combinations of the different ranges. We also didn't have time to run the algorithms on different subsets of features to see how the different features affect the results. There are also a lot of other data sources on which some future work could evaluate the different algorithms. We used the Execution Logs as a proof of concept that the system could handle more data but we didn't evaluate its precision on that data source and didn't have time to create interesting features for this data source which is very different from the Bro-SSH data source. Some future work could perform a more detailed evaluation of the system.

7.3 User Interface

So far, the user interacts with the system through the command line. He/She must manage where and how the Parquet and CSV files are stored. It would be much more convenient to implement a Web service that uses the CLI in the background, manages the storage of the different files and provides an intuitive user interface to launch scans and visualize results on a web page instead of opening CSV files manually. This would also eliminate the need to install the system on every user machine or the need for the user to SSH into a machine with the system installed. If some future work would implement such a service, users would only require a Web browser.

7.4 Contributions

The system has been designed to facilitate future additional implementations of some functionalities. The user can choose aggregation functions for features, ensemble techniques, rules, etc. But there is also the possibility to add new ones. Here are instructions for some possible contributions:

- Create a new aggregate function for features: in `features/Feature.scala`, in the companion object, the user can implement a new function of type `String => Column` where the `String` is the column name. It is possible to use already predefined aggregation functions from `org.apache.spark.sql.functions._` or to build a custom function by extending the class `UserDefinedAggregateFunction`. Once the user has this function, he can add a case in the `getAggFunction` method in `features/FeaturesParser.scala` using a name of his choice. This name will be string to specify in the `"aggs"` array in the JSON file of the features schema in order to use the function.

- Create a new entity extractor: in `features/EntityExtractor.scala`, in the companion object, the user can create a new `EntityExtractor` and add it to the list `extractors`. The `name` field of your extractor will be the string to specify in `conf/application.conf` or with the `--extractor` flag in order to use it.
- Create a new kind of intrusion / anomaly: in `evaluation/IntrusionKind.scala`, in the object the user can create a new function of type `String => IntrusionKind` where the input string is the description of the intrusion kind. Then add it to the list `allKinds`. The `name` field of your intrusion kind will be the string to use as name in the JSON file specified by `intrusions` in `conf/application.conf` or by the `--intrusions` flag.
- Create a new anomaly detection algorithm: the main class of the implementation must extend the `detection.Detector` class. Then in the object in `detection/Detector.scala`, the user can add a case to return a new instance of the `Detector` in the method `getDetector`. The string of the pattern match should be the name of the `Detector` that will be specified in the `detectors` field in `conf/application.conf` or in the `--detectors` flag.
- Create a new inspection rule: in `inspection/RulesParser.scala`, the user can add a case to return a new `Rule` in the method `getRule`. The string of the pattern match should be the `Rule`'s name to use in the JSON file specified by `rules` in `conf/application.conf` or by the `--rules` flag. The parameters of the rule are defined in `params` as `Strings`, if the user needs to parse them to some other data type like `Int`, proper error handling is in order like in the already implemented rules that involve parsing. `Rule` is abstract so the user must implement either a `SimpleRule` (over single log entry) or a `ComplexRule` (over all logs of the anomaly). In case of `SimpleRule`, the generic method `makeSimpleRule` defined in the companion object should be used.

References

- [1] Davide Ariu, Roberto Tronci, and Giorgio Giacinto. "HMMPayl: an Intrusion Detection System based on Hidden Markov Models" (). DOI: <https://www.sciencedirect.com/science/article/pii/S0167404811000022>.
- [2] Guangchun Luo et al. "A Parallel DBSCAN Algorithm Based On Spark" (). DOI: <https://ieeexplore.ieee.org/document/7723739?reload=true>.
- [3] Kalyan Veeramachaneni and Ignacio Araldo. "AI2 : Training a big data machine to defend" (). DOI: https://people.csail.mit.edu/kalyan/AI2_Paper.pdf.
- [4] Paul Dokas et al. "Data Mining for Network Intrusion Detection" (). DOI: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.331.6701&rep=rep1&type=pdf>.
- [5] Vivek A. Patole, Mr. V. K. Pachghare, and Dr. Parag Kulkarni. "Self Organizing Maps to Build Intrusion Detection System" (). DOI: <https://pdfs.semanticscholar.org/72ff/1517f2aa80cea440351bfec63e4d2f24e40b.pdf>.

- [6] Tugdual Sarazin, Hanane Azzag, and Mustapha Lebbah. “SOM Clustering using Spark-MapReduce” (). DOI: <https://ieeexplore.ieee.org/document/6969583/>.
- [7] Aaron Tuor et al. “Deep Learning for Unsupervised Insider Threat Detection in Structured Cybersecurity Data Streams” (). DOI: <https://arxiv.org/abs/1710.00811>.
- [8] Arthur Zimek, Ricardo J. G. B. Campello, and Jorg Sander. “Ensembles for Unsupervised Outlier Detection: Challenges and Research Questions” (). DOI: http://www.kdd.org/exploration_files/V15-01-02-Zimek.pdf.
- [9] Mohammed A. Ambusaidi, Xiangjian He, and Priyadarsi Nanda. “Unsupervised Feature Selection Method for Intrusion Detection System” (). DOI: <https://ieeexplore.ieee.org/document/7345295/>.
- [10] Arthur Zimek et al. “Subsampling for Efficient and Effective Unsupervised Outlier Detection Ensembles” (). DOI: <http://www.dbs.ifi.lmu.de/~zimek/publications/KDD2013/subsampling-outlier-ensemble.pdf>.
- [11] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. “Isolation Forest” (). DOI: <https://cs.nju.edu.cn/zhoush/zhoush.files/publication/icdm08b.pdf>.
- [12] David Arthur and Sergei Vassilvitskii. “k-means++: The Advantages of Careful Seeding” (). DOI: <http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf>.
- [13] Pedro Casas Hernandez, Johan Mazel, and Philippe Owezarski. “Unsupervised Network Intrusion Detection Systems: Detecting the Unknown without Knowledge” (). DOI: <https://hal.archives-ouvertes.fr/hal-00736278/document>.
- [14] Stefano Zanero and Giuseppe Serazzi. “Unsupervised Learning Algorithms for Intrusion Detection” (). DOI: <https://ieeexplore.ieee.org/document/4575276/>.
- [15] Leonid Portnoy. “Intrusion detection with unlabeled data using clustering” (). DOI: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.796>.
- [16] Markus M. Breunig et al. “LOF: Identifying Density-Based Local Outliers” (). DOI: <http://www.dbs.ifi.lmu.de/Publikationen/Papers/LOF.pdf>.
- [17] VARUN CHANDOLA, ARINDAM BANERJEE, and VIPIN KUMAR. “Anomaly Detection : A Survey” (). DOI: https://www.vs.inf.ethz.ch/edu/HS2011/CPS/papers/chandola09_anomaly-detection-survey.pdf.
- [18] *cert-anomaly-ids*. URL: <https://github.com/mthambipillai/cert-anomaly-ids>.
- [19] *Apache Hadoop HDFS*. URL: <https://hortonworks.com/apache/hdfs/>.
- [20] *JSON*. URL: <https://www.json.org/>.
- [21] *Apache Parquet*. URL: <https://parquet.apache.org/>.
- [22] *Apache Kite*. URL: <http://kitesdk.org/docs/current/>.
- [23] *Apache Spark*. URL: <https://spark.apache.org/>.
- [24] *Scala*. URL: <https://www.scala-lang.org/>.
- [25] *Scalaz*. URL: <https://github.com/scalaz/scalaz>.
- [26] *Apache Spark MLlib*. URL: <https://spark.apache.org/mllib/>.

- [27] *RPM Package Manager*. URL: <http://rpm.org/>.
- [28] *Koji*. URL: <https://pagure.io/koji>.
- [29] *RDD*. URL: <https://spark.apache.org/docs/1.5.1/api/java/org/apache/spark/rdd/RDD.html>.
- [30] *DataFrame*. URL: <https://spark.apache.org/docs/1.5.1/api/java/org/apache/spark/sql/DataFrame.html>.
- [31] *djb2*. URL: <http://www.cse.yorku.ca/~oz/hash.html>.
- [32] *log4j*. URL: <https://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Level.html>.
- [33] *LDAP*. URL: https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol.
- [34] *Scala duration notation*. URL: [https://www.scala-lang.org/api/current/scala/concurrent/duration/package\\$\\$DurationInt.html](https://www.scala-lang.org/api/current/scala/concurrent/duration/package$$DurationInt.html).
- [35] *Isolation Forest Spark*. URL: <https://github.com/titicaca/spark-iforest>.
- [36] *LOF Spark*. URL: <https://github.com/hibayesian/spark-lof>.
- [37] *Locality-sensitive hashing*. URL: https://en.wikipedia.org/wiki/Locality-sensitive_hashing.
- [38] Mayur Datar et al. “Locality-Sensitive Hashing Scheme Based on p-Stable Distributions” (). DOI: <https://www.mlpack.org/papers/lsh.pdf>.
- [39] *Bro-SSH*. URL: <https://www.bro.org/sphinx/scripts/base/protocols/ssh/main.bro.html#type-SSH::Info>.
- [40] *Virus Total*. URL: <https://www.virustotal.com/#/home/upload>.
- [41] *Abuse IP DB*. URL: <https://www.abuseipdb.com/>.
- [42] *Nomad*. URL: <https://www.nomadproject.io/>.

A Bro-SSH Features Schema

```
1 {
2   "name" : "BroSSH",
3   "features" : [ {
4     "name" : "auth_attempts",
5     "parent" : null,
6     "type" : "Int",
7     "doc" : "The number of authentication attempts we observed.
8     There is always at least one, since some servers might
9     support no authentication at all. It is important to note
10    that not all of these are failures, since some servers
11    require two-factor auth (e.g. password AND pubkey)",
12    "aggs" : ["sum", "max", "min", "mean"]
13  }, {
14    "name" : "auth_success",
15    "parent" : null,
16    "type" : "Boolean",
17    "doc" : "Authentication result (T=success, F=failure, unset
18    =unknown)",
19    "aggs" : ["mostcommon", "mean"]
20  }, {
21    "name" : "cipher_alg",
22    "parent" : null,
23    "type" : "String",
24    "doc" : "The encryption algorithm in use",
25    "aggs" : ["mostcommon", "countdistinct"]
26  }, {
27    "name" : "client",
28    "parent" : null,
29    "type" : "String",
30    "doc" : "The client version string",
31    "aggs" : ["mostcommon"]
32  }, {
33    "name" : "server",
34    "parent" : null,
35    "type" : "String",
36    "doc" : "The server version string",
37    "aggs" : ["mostcommon"]
38  }, {
39    "name" : "version",
40    "parent" : null,
41    "type" : "Int",
42    "doc" : "SSH major version (1 or 2)",
```

```

38     "aggs" : ["mostcommon"]
39 }, {
40     "name" : "direction",
41     "parent" : null,
42     "type" : "String",
43     "doc" : "Direction of the connection. If the client was a
        local host logging into an external host, this would be
        OUTBOUND. INBOUND would be set for the opposite situation.",
44     "aggs" : ["mostcommon", "mean"]
45 }, {
46     "name" : "dsthost",
47     "parent" : null,
48     "type" : "Host",
49     "doc" : "The responder hostname (reverse dns of IP address)
        .",
50     "aggs" : ["countdistinct"]
51 }, {
52     "name" : "dstip",
53     "parent" : null,
54     "type" : "String",
55     "doc" : "The responder IP address.",
56     "aggs" : ["countdistinct"]
57 }, {
58     "name" : "dstport",
59     "parent" : null,
60     "type" : "Int",
61     "doc" : "The responder port number.",
62     "aggs" : ["mostcommon", "countdistinct"]
63 }, {
64     "name" : "srchost",
65     "parent" : null,
66     "type" : "Host",
67     "doc" : "The originator hostname (reverse dns of IP address)
        ).",
68     "aggs" : ["countdistinct"]
69 }, {
70     "name" : "srcip",
71     "parent" : null,
72     "type" : "String",
73     "doc" : "The originator IP address.",
74     "aggs" : ["countdistinct"]
75 }, {
76     "name" : "timestamp",
77     "parent" : null,
78     "type" : "Long",

```

```

79     "doc" : "Time when the SSH connection began.",
80     "aggs" : []
81   }
82 }

```

B Execution Logs Features Schema

```

1 {
2   "name" : "ExecLogs",
3   "features" : [ {
4     "name" : "euid",
5     "parent" : null,
6     "type" : "Int",
7     "doc" : "Effective UNIX user ID.",
8     "aggs" : ["countdistinct"]
9   }, {
10    "name" : "pid",
11    "parent" : null,
12    "type" : "Int",
13    "doc" : "Process ID.",
14    "aggs" : ["countdistinct"]
15  }, {
16    "name" : "toplevel_hostgroup",
17    "parent" : null,
18    "type" : "String",
19    "doc" : "Top level host group.",
20    "aggs" : ["mostcommon", "countdistinct"]
21  }, {
22    "name" : "argv",
23    "parent" : null,
24    "type" : "String",
25    "doc" : "Executed command.",
26    "aggs" : ["mostcommon", "countdistinct"]
27  }, {
28    "name" : "sid",
29    "parent" : null,
30    "type" : "Int",
31    "doc" : "Security identifier.",
32    "aggs" : []
33  }, {
34    "name" : "ppid",
35    "parent" : null,
36    "type" : "Int",

```

```

37     "doc" : "Parent process ID.",
38     "aggs" : ["countdistinct"]
39 }, {
40     "name" : "uid",
41     "parent" : null,
42     "type" : "Int",
43     "doc" : "User ID.",
44     "aggs" : ["mostcommon", "countdistinct"]
45 }, {
46     "name" : "egid",
47     "parent" : null,
48     "type" : "Int",
49     "doc" : "Effective UNIX group ID.",
50     "aggs" : ["mostcommon", "countdistinct"]
51 }, {
52     "name" : "environment",
53     "parent" : null,
54     "type" : "String",
55     "doc" : "Environment of the host.",
56     "aggs" : ["mostcommon"]
57 }, {
58     "name" : "file",
59     "parent" : null,
60     "type" : "String",
61     "doc" : "Executable file that was executed.",
62     "aggs" : ["mostcommon", "countdistinct"]
63 }, {
64     "name" : "deleted",
65     "parent" : null,
66     "type" : "Boolean",
67     "doc" : "Whether file was deleted.",
68     "aggs" : ["mostcommon"]
69 }, {
70     "name" : "host",
71     "parent" : null,
72     "type" : "String",
73     "doc" : "Host name of the machine.",
74     "aggs" : ["mostcommon", "countdistinct"]
75 }, {
76     "name" : "hostgroup",
77     "parent" : null,
78     "type" : "String",
79     "doc" : "Host group.",
80     "aggs" : ["mostcommon", "countdistinct"]
81 }, {

```

```

82     "name" : "timestamp",
83     "parent" : null,
84     "type" : "Long",
85     "doc" : "Time of the execution.",
86     "aggs" : []
87 }, {
88     "name" : "lastboot",
89     "parent" : null,
90     "type" : "Double",
91     "doc" : "Time since system boot.",
92     "aggs" : ["max"]
93 } ]
94 }

```

C Bro-SSH Rules

```

1 {
2     "name" : "BroSSHrules",
3     "rules" : [ {
4         "name" : "ssh_auth_attempts",
5         "params" : ["4"],
6         "text" : "high/unknown nb attempts"
7     }, {
8         "name" : "ssh_dstport",
9         "params" : [],
10        "text" : "dst port not 22"
11    }, {
12        "name" : "ssh_version",
13        "params" : ["2"],
14        "text" : "version less than 2.x"
15    }, {
16        "name" : "ssh_srcip",
17        "params" : ["$CERT_ANOMALY_IDS_HOME/stats/knownips.txt"],
18        "text" : "known malicious ip"
19    }, {
20        "name" : "ssh_dsthost",
21        "params" : ["$CERT_ANOMALY_IDS_HOME/stats/dsthoststats.txt"]
22    }, {
23        "name" : "ssh_client",
24        "params" : ["$CERT_ANOMALY_IDS_HOME/stats/clientstats.txt"]
25    },

```



```

26     "text" : "unusual client"
27 }, {
28     "name" : "ssh_server",
29     "params" : ["$CERT_ANOMALY_IDS_HOME/stats/serverstats.txt"]
30     ,
31     "text" : "unusual server"
32 }, {
33     "name" : "ssh_cipher",
34     "params" : ["$CERT_ANOMALY_IDS_HOME/stats/cipher_algstats.
35     txt"],
36     "text" : "unusual cipher"
37 }, {
38     "name" : "ssh_total_auth_attempts",
39     "params" : ["30"],
40     "text" : "high total nb of attempts"
41 }
42 }

```

D Bro-SSH Intrusions

```

1 {
2     "name" : "BroSSHintrusions",
3     "intrusions" : [ {
4         "name" : "tooManyAuthAttempts",
5         "doc" : "The src entity tried to authenticate an unusual
6         number of times to the same dst",
7         "requiredcolumns" : ["auth_attempts", "srchost"],
8         "number" : 3
9     }
10 ]
11 }

```