Yale University
Department of Computer Science

Similarity Detection via Random Subsets for Cyber War Protection in Big Data using Hadoop Framework

Dafna Ackerman[1] Amir Averbuch[2] Avi Silberschatz[3]
Moshe Salhov[2]

YALEU/DCS/TR-1517
July 10, 2015

[1]School of Engineering, Tel Aviv University, Tel Aviv 69978, Israel
[2]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
[3]Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA

Abstract

The increasing volume of Internet traffic to be analyzed imposes new challenges for anomaly detection systems. These systems should efficiently analyze a large amount of data to discover anomalous fragments within a reasonable response time. In this work, we propose a method for anomaly detection such as intrusion attacks in networking data, based on a parallel similarity detection algorithm that uses the MapReduce methodology implemented on an Hadoop platform. The proposed system processes large amount of data on a commodity hardware. The experimental results on the 2009 DARPA database demonstrate that the proposed system scales very well when data sizes increase.

List of Abbreviations

SDA	Similarity Detection Algorithm
IDS	Intrusion Detection System
MDP	Multidimensional Data Point
EMDP	Embedded Multidimensional Data Point
HDFS	Hadoop Distributed File Systems
DoS	Denial of Service
DDoS	Distributed Denial of Service
DM	Diffusion Maps
HTCondor	High-Throughput Computing Condor
DNS	Domain Name System
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IP	Internet Protocol
ICMP	Internet Control Message Protocol
LLC	Logical Link Control
ARP	Address Resolution Protocol
RM	ResourceManager
NM	NodeManager
AM	ApplicationMaster

1 Introduction

In recent years, growing attention has been given to network security problems, in particular, network intrusion detection. This is due to the indisputable dependence on computer networks for personal, business and government use. Intrusion Detection System (IDS) has the potential to mitigate or prevent such attacks. Intrusion detection is the process of monitoring network traffic for unauthorized use, misuse, and abuse of computer systems by both system insiders and external penetrators [37].

The increasing volume of Internet traffic to be analyzed imposes new challenges to an IDS performance. IDSs should efficiently analyze a huge amount of data to discover anomalous fragments within a reasonable response time. Due to the complexity and the size of the data generated by large-scale systems, traditional IDSs, which were designed for small-scale network systems, cannot be directly applied to large-scale systems. In order to cope with the increase in data sizes, new distributed methods need to be developed in order to make the IDSs scalable in their operation [49].

In this paper, we propose an IDS, which applies a distributed similarity detection algorithm (SDA) that uses the MapReduce methodology implemented on an Hadoop platform. The proposed IDS processes large amount of data on a commodity hardware. The experimental results from a 7TB 2009 DARPA dataset demonstrate that the proposed system scales well with the increase of the data size.

This paper has the following structure: Section 2 discusses related work on detecting intrusion attacks in networking data. Section 3 describes different preliminaries of the mathematical background needed for the algorithm and the technical details of the frameworks used to implement it. Section 4 presents a detailed description of the SDA and how it is implemented. Section 5 presents the results from the application of the algorithm applied to the 2009 DARPA dataset. Finally, section 6 presents our conclusions and suggests options for future work.

2 Related Work

There are numerous approaches for detecting intrusion attacks in networking data and it has been extensively studied over the years [10, 19, 50]. Intrusion detection techniques are traditionally categorized into two methodologies: anomaly detection and signature detection [27]. General description of these methodologies is presented in Fig. 2.1.

(a) Anomaly Detection

(b) Signature Detection

Figure 2.1: Intrusion Detection Techniques

Anomaly detection based IDS monitors network traffic and compares it against an

established baseline. The baseline identifies what are the "normal" characteristics for that network. This IDS stores features of the user's typical behaviours into a database, then compares the user current behavior with those in the database. It treats any incoming traffic, which is significantly different than the normal profile, as anomalous [49]. It is capable of identifying attacks which were not previously detected but have a high false alarm rate for a legitimate activity [12]. This is due to the fact that the normal profile cannot provide a complete description for the behaviours of all the users in the system. Moreover, user's behaviour changes constantly.

There are numerous anomaly detection techniques, which make use of supervised or unsupervised methods, to detect abnormal behaviors in patterns. Parametric approaches assume that the normal data is generated by a parametric probability density function. This technique is often used in quality control domain [46]. Another category of parametric anomaly detection is known as "Regression Model Based" [11]. Non-parametric techniques include density estimation and clustering-based techniques. These techniques are known as non-parametric techniques since they do not rely on a specific parametric representation of the data but rely on the interrelations between data-points to separate between normal and abnormal behaviors in the data. Density estimation based anomaly detection methods, such as in [45, 55], rely on a reasonable assumption that normal trends can be defined and detected by the behaviors of most of the observations in the dataset. Clustering based anomaly detection methods, such as in [58, 40, 9], group together data-points based on a similarity measure where the anomalies (attacks) are the patterns in the smallest clusters. Two assumptions have to hold for anomaly detection method to be effective:

1. The number of normal patterns should be higher than the number of the anomalies.

2. The anomalies should be distinguishable from normal patterns.

Signature detection based IDS monitors network traffic and compares it against a database of signatures or attributes from previously known malicious threats. This method involves prior knowledge of anomalies patterns [23]. Its advantages are in its high detection speed, low false alarm rate and high true-positive rate for known anomalies. However, it fails in detecting new emerging threats and the attacks signature database has to be manually updated, which is time consuming and effectively creates a lag between a new threat discovery and its signature being applied by the IDS [51]. Signature detection based IDS that uses SNORT [41], which is an open source network intrusion and prevention, has been proposed in [30, 31, 39]. Other signature methods use algorithms to generate decision trees adaptive to the network traffic characteristics [28, 7].

The results in [56] show, for the first time, that it is possible to efficiently obtain meaningful statistical information from an immense raw traffic data through MapReduce methodology implemented on Hadoop. As a result, a growing number of IDSs adjusted to large-scale networks by their implementation on Hadoop platform have been proposed [26, 33, 8]. Two algorithms for detecting DDoS attacks using MapReduce methodology are given in [53, 54]:

1. Counter based algorithm: A method that counts the total traffic volume or the number of web page requests. This method relies on three key parameters:

(a) Time interval: the time duration in which the packets are analyzed.

(b) Threshold: frequency of requests.

(c) Unbalance ratio: the anomaly ratio of responses per page requested between a specific client and a server.

The number of requests from a specific client to a specific URL within the same time duration, is counted using a masked timestamp. The map function filters non-HTTP GET packets and generates key values of the server IP address, masked timestamp and client IP address. The reduce function aggregates the number of URL requests, number of page requests and the total server responses between a client and a server. Finally, values per server are aggregated by the algorithm. When the threshold is crossed and the unbalance ratio is higher than the normal ratio, the clients are marked as attackers. The key advantages of this algorithm is its low complexity that could be easily converted to a MapReduce implementation. However, the authors indicate that the threshold value is a key factor in the algorithm, they do not offer any further information on how to determine its value.

2. Access pattern based algorithm: This method is based on a pattern that differentiates the normal traffic from the DDoS attacks. This method requires two MapReduce jobs: the first job gets the access sequence to the web page between a client and a web server that computes the spending time and the bytes count for each URL request. The second job finds suspicious hosts by comparing between the access sequence and the spending time among clients trying to access the same server. This method has a high computational complexity and ad-hoc queries are delayed due to the FIFO scheduling [43].

An IDS, which is based on a SNORT alert log using Hadoop, is proposed in [13]. This method contains 3 components: Snorts, Chukwa, which is a package that collects logs in various computers and stores them in HDFS, and Hadoop. Each Snort monitors a working server and sends alert logs to Chukwa agent as an adaptor. Chukwa agents send logs to a Chukwa collector, which writes logs into a single sink file. Periodically, the file is closed and the next sink file is created. The map function parse the sink files for warning messages using special expressions. Then, the reduce function calculates the count of each warning type.

An unstructured log analysis technique for anomalies detection implemented on Hadoop platform is described in [20]. This technique is based on an algorithm that converts free form text messages in log files to log keys without heavily relying on application specific knowledge. The log keys correspond to the log-print statements in the source code which can provide clues of system execution behavior. After converting log messages to log keys, a Finite State Automaton (FSA) is learned from the training log sequences to present the normal work flow for each system component. At the same time, a performance measurement model is learned to characterize the normal execution performance based on the timing information of the log messages. With these learned models, the application can automatically detect anomalies in new input log files.

3 Preliminaries

This section describes different preliminaries of the mathematical background used in the algorithm and the technical details of the frameworks used to implement it. It describes Hadoop and MapReduce, HTCondor, different intrusion attacks and a diffusion based method that reduces the number of extracted parameters by embedding the input data into a lower dimension space. These methodologies and information are needed for the implementation of the SDA.

3.1 Hadoop and MapReduce

Hadoop [3] is a popular Apache open source platform. It is a Java-based implementation that provides tools for parallel processing of vast amounts of data using the MapReduce paradigm [36]. It can be used to process immense amounts of data in a distributed manner on large clusters in a reliable and fault-tolerant fashion [34].

MapReduce is a programming model for processing large datasets. It was first suggested by Google in 2004 [18]. MapReduce splits a computational task into two steps such that the whole process is automatically parallelized and executed on a large cluster of computational elements. The user only need to specify map and reduce functions. All mappers and reducers are independent from each other, thus, the application can run in parallel on different blocks of the input data. A mapper maps the input data, which is in the form of key/value pairs, to a list of intermediate key/value pairs by the map method denoted by $map(key1, value1) \rightarrow list < key2, value2 >$. The mapped intermediate records do not have to be of the same type as the input records. A given input pair may map to zero or to multiple output pairs. The reducer reduces a set of intermediate values that share a common key to a smaller set of values by a reduce function, which is denoted by $reduce(key2, list < value2 >) \rightarrow list < value3 >$. The reducer has three primary phases:

1. Sort: The framework groups the inputs to each reducer by a key since different mappers may output the same key.

2. Shuffle: The reducer copies the relevant sorted output from each mapper using HTTP across the network. The shuffle and sort phases occur simultaneously, i.e., while outputs are being fetched they are merged.

3. Reduce: The reduce function is called for each (key, <values>) pair in the grouped inputs. The output of each reducer is written to Hadoop Distributed File System (HDFS) and it is not re-sorted.

Data flow through MapReduce architecture is depicted in Fig. 3.1.

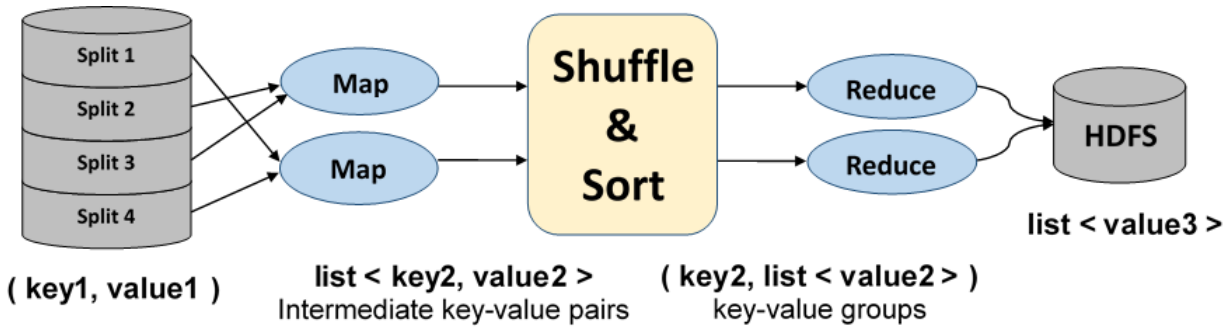


Figure 3.1: Data flow through MapReduce architecture

In some cases, there is a significant repetition in the intermediate keys produced by each map task and the user specified reduce function is commutative and associative. In this case, the user can specify an optional combiner function that does a partial merge of the intermediate data before it is sent to the reducer over the network [17]. The combiner is a "mini-reduce" process that is executed only on data generated by one mapper. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file on HDFS. The output of a combiner function is written to an intermediate file that is sent to a reduce task. A partial combining significantly speeds up certain classes of MapReduce operations.

HDFS is a distributed file system designed to run on commodity hardware in a fault-tolerance manner [29]. Files in a Hadoop cluster are typically gigabytes to terabytes in size. HDFS divides each file into smaller blocks and distributes copies of these blocks throughout the cluster across different machines. This way, the map and reduce functions are executed on smaller subsets of large datasets. This provides the scalability that is needed for big data processing. HDFS uses a master/slave architecture where a master node consists of a single NameNode that manages the file system metadata and the slave nodes are one or more DataNodes that store the actual data. To ensure high availability, there are often an active NameNode and a standby NameNode.

Hadoop consists of a framework for job scheduling and cluster resource management, called YARN. YARN can run applications that do not follow the MapReduce model. It permits simultaneous execution of a variety of programming models, including graph processing, iterative processing, machine learning, and general cluster computing. YARN consists of a global ResourceManager (RM), NodeManager (NM) and a per-application ApplicationMaster (AM). The RM manages and allocates all cluster resources among all applications in the system. The NM manages and enforce node resources allocations. The AM manages each application life-cycle and task scheduling. Every node in the cluster is considered to be composed of multiple containers where a container is an available resource on a single node. A container is supervised by the NM and scheduled by the AM. One MapReduce task runs on a set of containers, as each map or reduce function runs on one container.

When a client request the execution of an application, the RM negotiates the necessary resources and launches an AM to represent the submitted application. Using a resource-request protocol, the AM negotiates resource containers for the application at each node. Upon execution of the application, the AM monitors the containers until completion.

When the application completes, the AM unregisters its containers with the RM, thus completing the cycle [2].
A description of YARN architecture is presented in Fig. 3.2.

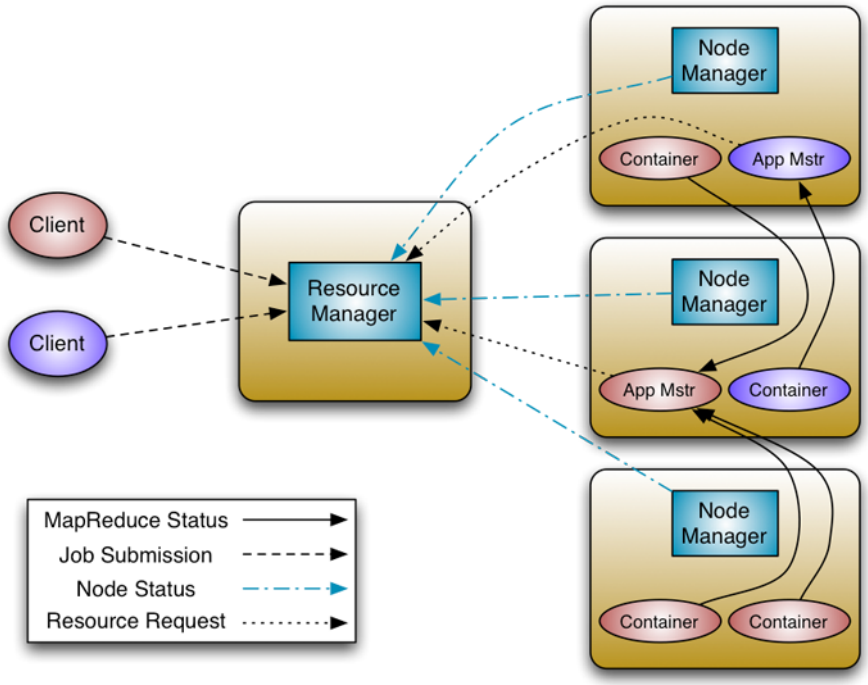


Figure 3.2: YARN Architecture [1]

3.2 High-Throughput Computing Condor (HTCondor)

HTCondor [5] is an open-source of High-Throughput Computing (HTC) software framework for distributed batch jobs. It can be used to manage workload on a dedicated cluster of computers and/or harness non-dedicated resources, such as idle desktop computers, under a distributed ownership. Thus, it can efficiently harness the use of all available computing resources [48]. Like other batch systems, HTCondor provides a job management and scheduling mechanism, priority scheme and resource monitoring and management [47]. When a user submits a job to HTCondor, it finds an available machine on the network to run the job on that machine. HTCondor has the capability to detect that a machine running a HTCondor job is no longer available. It can checkpoint the job and migrate this job to a different idle machine. Then, it continues the job on a new machine from precisely where it left off.

3.3 Intrusion Attacks

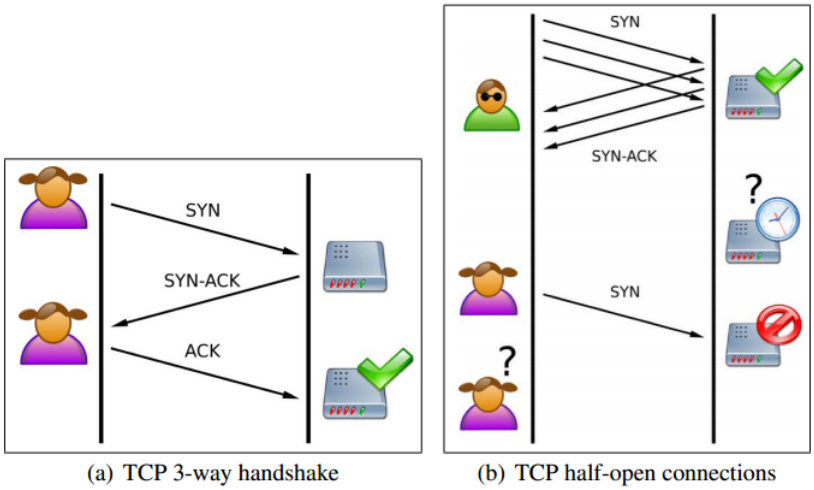
In this section, we describe a few common intrusion attacks, which are present in the 2009 DARPA dataset, and explain their significance.

1. **Denial of Service Attacks (DoS)**
   A DoS attack is an attack in which the attacker causes the resources of CPU and memory to be too busy or too full to handle other legitimate requests [57].

Therefore, it can deny legitimate users access to a machine.
In a DoS attack, one computer and one Internet connection are used to flood a server with packets with the aim of overloading the targeted bandwidth and resources of the victim.
A DDoS attack uses multiple machines and Internet connections to attack a specific machine. It is often distributed globally using a botnet. A botnet is a number of Internet connected machines that, although their owners are unaware of it, have been set up to forward transmissions, which include spam or viruses, to other machines. [16]. Therefore, DDoS attack is hard to deflect because there is no single attacker to defend from since the targeted resource is flooded with requests from hundreds or even thousands of different sources.
There are many varieties of DoS attacks [38], such as:

Mail Bomb: The attacker sends ample mail messages to a server, thus overflowing its mail queue that may cause a system failure.

SYN Flood: Occurs during the three-way handshake that marks the onset of a TCP/IP connection. In the three-way handshake, a client sends a SYN packet to a server to request a new connection. In response, the server sends a SYN-ACK packet back to the client and places the connection request in a queue of a finite size. As a final step, the client acknowledges the SYN-ACK packet. When an attack takes place, the attacker sends an abundance of TCP SYN packets to the victim forcing it to open a considerable number of TCP connections and to respond to them. Then, the attacker does not execute the final step of the three-way handshake. Thus, leaving the victim paralyzed since he is incapable to accept new incoming connections since its queue is full of half-open TCP connections [43]. Figure 3.3(a) presents a diagram of a normal TCP 3-way handshake. Figure 3.3(b) presents a diagram of a SYN flood attack where numerous half-open TCP connections deny service from legitimate users.



(a) TCP 3-way handshake (b) TCP half-open connections

Figure 3.3: SYN flood attack diagram

2. **Spambot**
   A spambot is an automated computer program designed to send spam. Spambots create fake accounts or collect e-mail addresses in order to build mailing lists for sending unsolicited e-mails. A spambot can gather e-mail addresses from web sites, newsgroups, chat-room conversations, etc.

3. **Domain Name System (DNS) redirection**
   DNS redirection is the practice of subverting DNS queries. This can be achieved by malware that overrides the TCP/IP configuration of a victim computer to point at a rogue DNS server under the control of the attacker, or through modifying the behaviour of a trusted DNS server so that it does not comply with Internet standards. These modifications may be made for malicious purposes such as phishing, blocking access to selected domains as a form of censorship, or for self-serving purposes by Internet service providers or public/router-based online DNS server providers to direct users' web traffic to the attacker own web servers where advertisements can be served, statistics collected, etc [25].

4. **Phishing**
   Phishing is the illegal attempt to acquire sensitive information, such as usernames, passwords, credit card details, etc, for malicious reasons by masquerading as a trustworthy entity in an electronic communication such as emails.

3.4 Diffusion Maps

Diffusion Maps (DM) is a dimensionality reduction algorithm introduced in [42, 14, 32]. Dimensionality reduction is the process of reducing the number of features in each Multidimensional Data Point (MDP). It can be used to extract latent features from raw and noisy features. It computes a family of embeddings of a dataset into an Euclidean space that is often low-dimensional whose coordinates can be computed from the eigenvectors and eigenvalues of a diffusion operator on the data. The Euclidean distance between data-points in the embedded space is equal to the diffusion distances between probability distributions centered at these data-points. Different from linear dimensionality reduction methods such as principal component analysis (PCA) [52] and multi-dimensional scaling (MDS) [24], DM is a nonlinear dimensionality reduction technique that focused on discovering the underlying manifold where the data resides. The shape of the low-dimensional manifold is unknown a priori. By integrating local similarities at different scales, DM provides a global description of the dataset. Compared with other methods, the DM algorithm is robust to noise perturbation and is computationally efficient.
The basic algorithm framework of DM has the following components:

1. **Kernel construction**
   For a dataset X of M MDPs $\{x_i\}_{i=1}^{M}$ where each $x_i \in \mathbb{R}^n$, a Gaussian kernel matrix also called a similarity matrix, is constructed by $k(x_i, x_j) = exp\left(-\frac{\|x_i - x_j\|}{2\varepsilon}\right)$ where ε is the scale (radius) of the Gaussian kernel.
   Setting the scale value has a great effect on the performance of the DM algorithm [22]. A small scale intensifies the notion of locality, however, it may result in a poorly connected graph. On the other hand, a large scale guarantees graph connectivity

but makes the kernel insensitive to variations in the distances. In practice, the scale is often determined as the empirical standard deviation of the dataset, however, there exist analytic methods for setting the scale [15].
The kernel represents some notion of affinity or similarity between MDPs of X as it describes the relationship between pairs of points. Different kernels, other than the Gaussian kernel, are also possible.

2. **Normalize the kernel matrix**
A diagonal normalization matrix $D_{ii} = \sum_i k(x_i, x_i)$ is used for kernel normalization to obtain the transition matrix $P = D^{-1}K$.

3. **Singular Value Decomposition (SVD) is applied to the transition matrix P**
SVD which is applied to P matrix, yields a complete sequence of left and right eigenvectors $\{\Phi_j, \Psi_j\}_{j=0}^{n-1}$ and eigenvalues $\{\lambda_j\}_{j=0}^{n-1}$. The eigenvalues are in a descending order such that $1 = \lambda_0 > |\lambda_1| \geq |\lambda_2| \geq \ldots$. The corresponding right eigenvector is $\Psi_0 = 1$.

4. **Embedding**
The δ largest right eigenvalues and the δ corresponding eigenvectors of the P matrix are used to obtain the embedded space by

$$\Psi_t(x_i) \triangleq [\lambda_1^t \psi_1, \ldots, \lambda_\delta^t \psi_\delta]^T \tag{3.1}$$

where $1 \leq \delta \leq M - 1$ is the new subspace dimensionality and $t > 0$ is the time steps. Each component in $\Psi_t(x_i)$ is termed a "diffusion coordinate".

4 The Similarity Detection Algorithm

4.1 Preprocessing: Features Extraction

The input dataset X to the SDA is a matrix of size $N \times n$. Each row in the matrix X corresponds to a features vector, which is a MDP, with n extracted features. The matrix is formed after a preprocessing phase of features extraction from raw pcap files. This way, a features based data is created.

For the DARPA 2009 intrusion dataset (section 5.1), each non-overlapping time interval is mapped into an MDP. For this purpose, a time interval-oriented traffic analyzer is defined: The traffic analyzer handles only ICMP and IP protocols. For each newly arrived packet, the analyzer parses its header and collects several values from the protocol. At every predetermined time interval (e.g., one second/minute), the analyzer summarizes the values collected during this time interval and saves them as an MDP. The following features were gathered via time aggregation and computed for every predefined time interval:

1. Number of TCP/IP packets;
2. Number of UDP/IP packets;
3. Number of ICMP packets;

4. Number of packets which are not TCP, UDP or ICMP;
5. Number of TCP packets with TCP flag “syn” ON;
6. Number of TCP packets with TCP flag “ack” ON;
7. Number of TCP packets with TCP flag “cwr” ON;
8. Number of TCP packets with TCP flag “ecn” ON;
9. Number of TCP packets with TCP flag “fin” ON;
10. Number of TCP packets with TCP flag “ns” ON;
11. Number of TCP packets with TCP flag “push” ON;
12. Number of TCP packets with TCP flag “res” ON;
13. Number of TCP packets with TCP flag “reset” ON;
14. Number of TCP packets with TCP flag “urg” ON;
15. Number of TCP packets with destination port 80 (HTTP);
16. Number of UDP packets with destination port 53 (DNS);
17. Number of TCP packets with source port 0;
18. Number of data TCP packets which where retransmitted (indication of slow application performance and packet loss);
19. Number of control TCP packets (packets without a payload);
20. Number of data TCP packets (packets with a payload);
21. Number of data TCP bytes (the bytes count of all the payloads);
22. Number of TCP connections (sessions);
23. Number of completed TCP connections;
24. Ratio between the number of TCP packets with reset flag ON and the number of TCP packets with syn flag ON (computed feature);
25. Ratio between the number of TCP packets with syn-ack flags and the number of TCP packets with syn flag (computed feature).

This preprocessing phase was performed by the application of the packet analyzer Tshark (terminal-based version of Wireshark) [6, 44] and by the HTCondor framework [5]. The process of extracting features from each pcap file was performed by Tshark and the distributed execution was done by HTCondor. The parsing of each pcap file was issued as a separate job by HTCondor. After the parsing of a pcap file is completed, the output file is copied to HDFS.

The preprocessing phase transforms the pcap files into a suitable input format to Hadoop. Each line in the output files from the preprocessing phase represents an MDP in the matrix X. Each line is divided into a key and a value parts by a <tab> character. The key is the time-stamp of the MDP (metadata that describes the MDP) and the value is the extracted features for a specific time interval. The delimiter between features is the character ','. An example of such a row in matrix X is:
11/4/2009 7:49 1687,30,0,0,64,1655,0,0,64,0,134,0,0,0,695,15,0,0,772,915,1266740,32,32,0.0228,0.9798.
A simplified configuration of the preprocessing phase is depicted in Fig. 4.1.

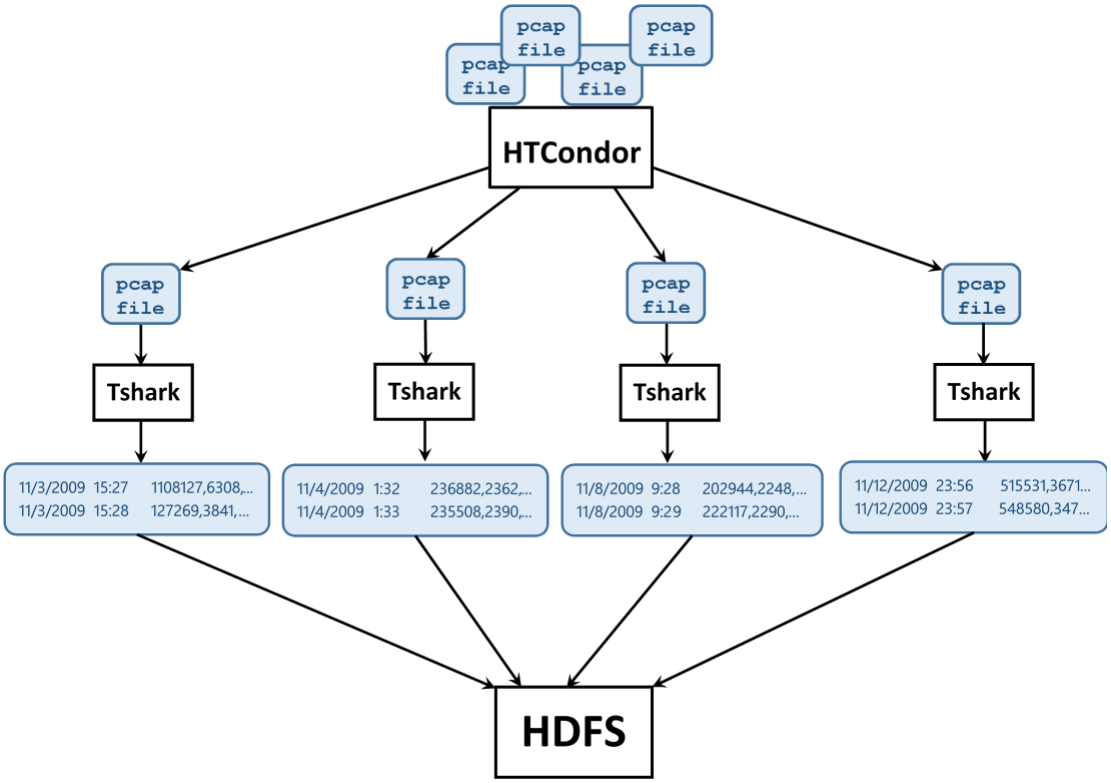


Figure 4.1: Preprocessing phase

4.2 Algorithm Description

The SDA finds similarities to a given initial designated MDP. Each MDP, which is similar to the designated MDP that was detected in a previous iteration of the algorithm, becomes a new designated MDP that leads to a detection of other similar MDPs to the newly designated MDP. This process is repeated until no other MDP is found or a predefined maximum number of iterations has been reached.

The input to the algorithm is a dataset X of size $N \times n$ and an initial designated MDP $y \in \mathbb{R}^n$, where N is the number of MDPs and n is the number of extracted features. If y does not belong to X then it is added to X and there will be $N+1$ MDPs in X. We assume that y belongs to X and the size of the matrix is $N \times n$.

Our objective is to find a set $\xi \subset X$ of MDPs for all $x_i \in X, i = 1, \ldots, N$, which are similar to $y \in \mathbb{R}^n$. Similarity is defined next.

We commence this process by randomly selecting a number k, such that

$0 < minFeatures \leq k \leq maxFeatures < n$. Then, a new subset dataset $\overline{X}_k = \{\overline{x}_1, \ldots, \overline{x}_N\}$, where $\overline{x}_i \in \mathbb{R}^k$ is constructed. In other words, the set of MDPs where $x_i \in X, i = 1, \ldots, N$, is transformed such that it has only k features instead of n features. The same transformation is applied to $y \in \mathbb{R}^n$ as well. It is denoted by $\overline{y}_k$ that also has the same k features instead of n features as $\overline{X}_k$. This type of transformation is performed repeatedly $numPermutations$ for random selections of k, where $numPermutations$ is the number of permutations for random selections of k for each initial designated MDP. The main assumption when using a feature selection technique is that the data contains redundancy, irrelevant or modified features. When the number of features is reduced, the data dimension also reduces and the new dataset is thus more compact [35].

Once the random feature selection is completed, nearest neighbours procedure is applied. A ball is defined around $\overline{y}_k$ such that

$$B_{\mu_1}(\overline{y}_k) \triangleq \{\overline{x}_i \in \overline{X}_k | \|\overline{y}_k - \overline{x}_i\| \leq \mu_1\}, i = 1, \ldots, N \tag{4.1}$$

where μ_1 defines the maximum distance allowed between $\overline{y}_k$ and all other MDPs in $\overline{X}_k$. The ball contains MDPs from $\overline{X}_k$ that are μ_1-close to $\overline{y}_k$. The ball in Eq. [**?**] determines the relation among MDPs.

Next, DM is applied to embed the MDPs that are contained in the ball $B_{\mu_1}(\overline{y}_k)$ (Eq. 4.1), as well as embedding $\overline{y}_k$ into the same lower dimension space. The output of the DM process is a matrix that is defined by Eq. 3.1, where its columns form a basis for the low-dimension space. We denote this matrix by $\overline{X}_\delta$ and denote $\overline{y}_k$ in the lower-dimensional space by $\overline{y}_\delta$. The MDPs in the lower dimensional space are denoted by Embedded Multidimensional Data Points (EMDP). Next, a nearest neighbours procedure is applied to $\overline{y}_\delta$. Thus, we find the nearest neighbours EMDPs to $\overline{y}_\delta$ from all $x_i \in \overline{X}_\delta, i = 1, \ldots, N$, where $x_i \in \mathbb{R}^\delta$. The set of nearest neighbours in the lower-dimensional space is defined by:

$$B_{\mu_2}(\overline{y}_\delta) \triangleq \{x_i \in \overline{X}_\delta | \|\overline{y}_\delta - x_i\| \leq \mu_2\}, i = 1, \ldots, N. \tag{4.2}$$

Each EMDP, which is μ_2-close to $\overline{y}_\delta$, is then added to the set of $nearestMDPs$.

This process is repeated multiple times with different random selections of k from the source of n features. The MDPs, which present in most iterations of the randomly chosen k, are described as similar to y. The number of times, which a specific MDP is expected to appear until it is described as similar, is denoted by the threshold Ω. Each MDP, which was described as similar, is added to the set of the designated MDPs denoted by ξ. Every similar MDP then becomes a new designated MDP and a source for a new search to detect other MDPs. If there are no related MDPs found, i.e. $\xi = \emptyset$, then the search process ends. As mentioned before, this process acts as a crawler due to the fact that we begin this process from a given initial designated MDP from which similar MDPs are found. These similar MDPs are then used for the discovery of additional similar MDPs and the process continues. Thus, one MDP leads the discovery of many other similar MDPs.

The parameters of the algorithm should be set beforehand. The parameters are:

$minFeatures$**:** Minimum number of features in each permutation, $0 < minFeatures \leq k$.

$maxFeatures$**:** Maximum number of features in each permutation, $k \leq maxFeatures < n$.

μ_1**:** Maximum distance allowed between $\overline{y}_k$ and all other MDPs in $\overline{X}_k$.

μ_2**:** Maximum distance allowed between $\overline{y}_\delta$ and all other EMDPs in $\overline{X}_\delta$.

Ω**:** Threshold of the number of occurrences for each MDP until it is described as similar.

$maxIterations$**:** Maximum number of iterations the algorithm can perform. Each iteration means the process of finding similar MDPs to a specific initial designated MDP.

Figure 4.2 describes the flow-chart of the SDA described in this section.

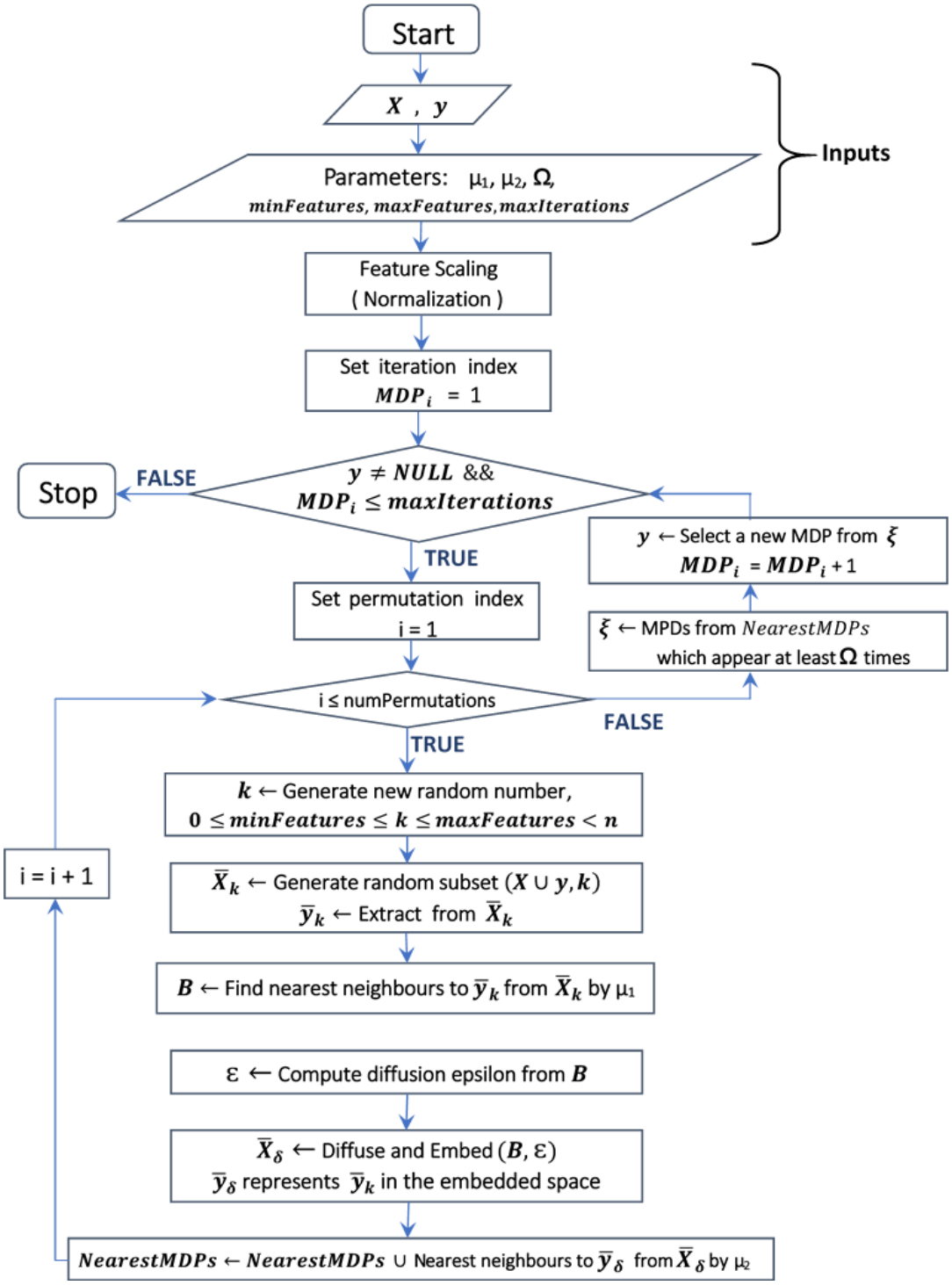


Figure 4.2: Algorithm flow chart

Next we describe in details each step in Fig. 4.2.

Input

The extracted features from the input data are arranged in a matrix X as described in section 4.1. $y \in X$ is the initial designated MDP where we are searching for all its similar MDPs in X. It describes a known attack.

Feature scaling (normalization)

The data is organized in matrix X of size $N \times n$, where N is the number of MDP samples and n is the number of extracted features. The data in each column of X is normalized. This step scales the range of independent features to a common numerical scale. It re-scales the features to be in the range of $[0, 1]$ in the following manner:

$$x' = \frac{x - min(x)}{max(x) - min(x)}. \tag{4.3}$$

Each column $i = 1, \ldots, n$ in X, which corresponds to a specific feature along the n rows of X, is normalized independently from the other columns as described in Algorithm 1.

Algorithm 1: Feature Scaling (Normalization)

Input: $N \times n$ matrix X with MDPs as its rows.
Output: Matrix $\overline{X}$ of size $N \times n$ where each feature (column) is normalized.

for $j = 1$ **to** n **do**
  $max \leftarrow 0$ # Initialize for each feature
  $min \leftarrow \infty$ # Initialize for each feature
  for $i = 1$ **to** N **do**
    if $X_{i,j} > max$ **then**
      $max \leftarrow x_{i,j}$ # Find max value of feature j
    end if
    if $X_{i,j} < min$ **then**
      $min \leftarrow x_{i,j}$ # Find min value of feature j
    end if
  end for
  for $i = 1$ **to** N **do**
    $\overline{X_{i,j}} \leftarrow \frac{X_{i,j} - min}{max - min}$ # Normalize $X_{i,j}$
  end for
end for
return $\overline{X}$

Generating a random subset

After randomly selecting a number k, such that $0 < minFeatures \leq k \leq maxFeatures < n$, we create a new dataset with only k features instead of n features. A pseudo-code for this phase is depicted in Algorithm 2.

Algorithm 2: Generating a Random Subset

Inputs:
$N \times n$ matrix X,
Length k of each row in the output subset.
Output: $N \times k$ matrix $\overline{X}$.

$\overline{X} \leftarrow \emptyset$
$N \leftarrow$ Number of rows in X
$n \leftarrow$ Number of columns in X
if $k < 1$ **or** $n \leq k$ **then**
    return $\overline{X}$
end if
$v \leftarrow$ A row vector containing k unique integers selected randomly from 1 to n inclusive.
for $i = 1$ **to** N **do**
    $x_i \leftarrow$ Get only the columns of $X_{i,1:n}$ that appear in v.
    $\overline{X} \leftarrow \overline{X} \cup x_i$
end for
return $\overline{X}$

Nearest neighbours

Finding nearest-neighbours MDPs to y.

A pseudo-code for this phase is depicted in Algorithm 3.

Algorithm 3: Nearest Neighbours

Inputs:
$N \times n$ matrix X,
y - Reference MDP which is compared to the other $N - 1$ MDPs of X,
μ - Maximum allowed distance from y
Output: Matrix $\overline{X}$ of all MDPs which are $\mu-$close to y.

$\overline{X} \leftarrow \emptyset$
for $i = 1$ **to** N **do**
    $x_i \leftarrow X_{i,1:n}$ # Row i of X
    $r \leftarrow \|x_i - y\|, x_i \in \mathbb{R}^n$
    if $r \leq \mu$ **then**
        $\overline{X} \leftarrow \overline{X} \cup x_i$
    end if
end for
return $\overline{X}$

Finding MDPs that appear at least Ω times

Detect the MDPs, which are the nearest-neighbour to y in the lower-dimensional space, for at least Ω permutations. A pseudo-code for this phase is depicted in Algorithm 4.

Algorithm 4: Finding MDPs that appear at least Ω Times

Inputs:
$N \times n$ matrix X
Ω - Threshold for the occurrences of each MDP in the neighbourhood of y.
Output: Matrix $\overline{X}$ of MDPs that have at least Ω occurrences in X.

```
1: X̄ ← ∅
2: uniqueMDPs ← Get only the unique MDPs from X by duplication removal.      # Array
3: counts ← Count the number of occurrences in X of each MDP from uniqueMDPs.  # Array
4: for all count ∈ counts do
5:     if count > Ω then
6:         x_i ← The corresponding MDP from uniqueMDPs for count
7:         X̄ ← X̄ ∪ x_i
8:     end if
9: end for
10: return X̄
```

4.3 Transforming the SDA to MapReduce

The SDA is iterative. The process of finding similar MDPs to a specific initial designated MDP is referred to as an iteration of the SDA. Each iteration is transformed to be executed in a distributed manner as MapReduce jobs. The first MapReduce job finds the minimum and maximum values for the features scaling phase. This MapReduce job is performed once at the beginning of the application for all iterations. Then, each iteration is divided into two MapReduce jobs:

1. Nearest-neighbours MapReduce job.

2. Pass-threshold MapReduce job.

These two MapReduce jobs are repeated for $maxIterations$ iterations. At the end of the SDA, a final MapReduce job is performed that compares the results of the SDA to a ground truth labeled data.
A flow-chart describing how the SDA is divided into MapReduce jobs is depicted in Fig. 4.3.

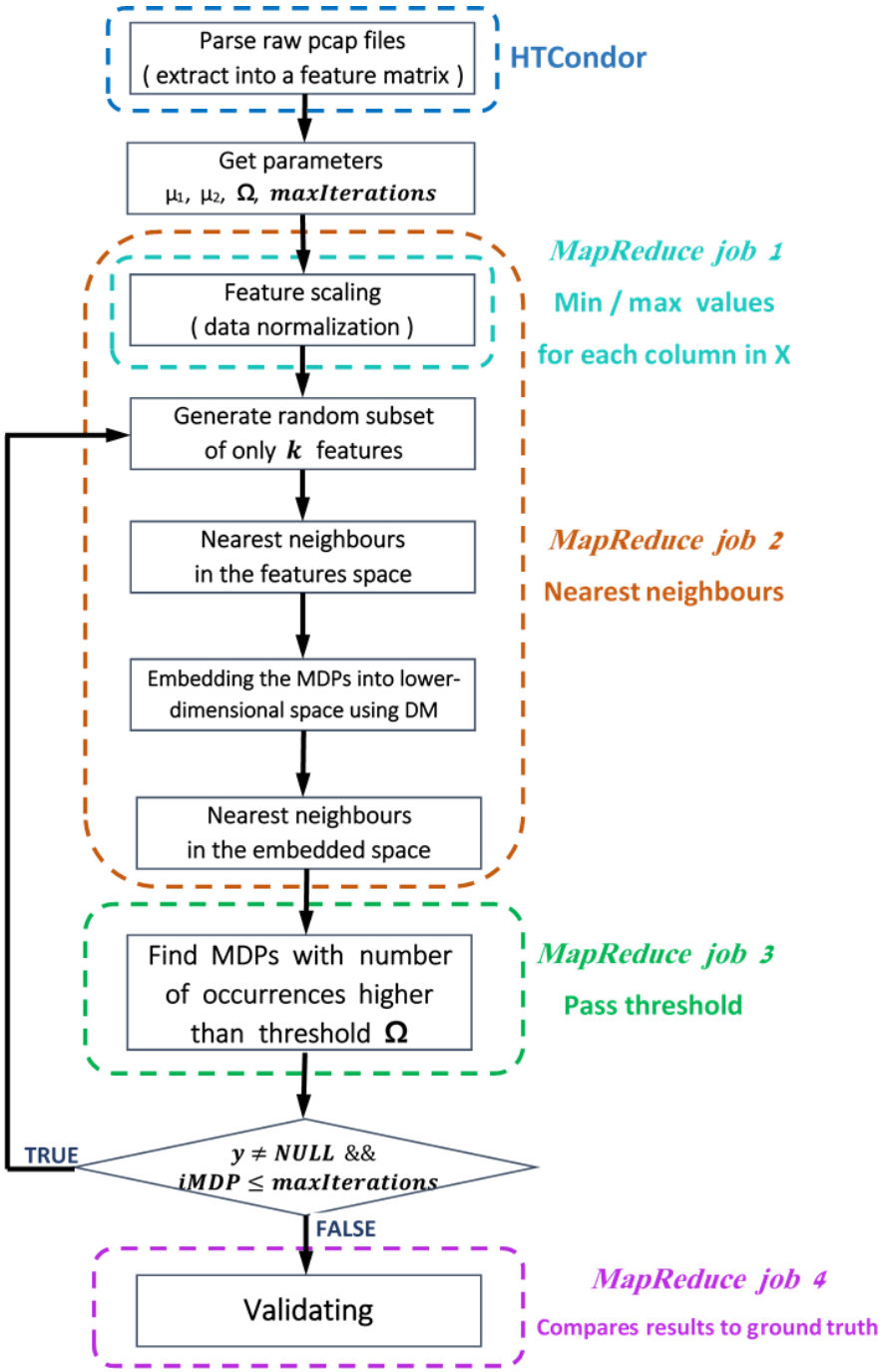


Figure 4.3: General algorithm flow chart divided into MapReduce jobs

The first MapReduce job finds the minimum and maximum values for each column of X where column i describes feature i for all the MDPs in X. The mapper of this job organizes all the features from the same column to go to the same reducer. Then, the reducer finds the minimum and maximum values. The input to this job is the output from the preprocessing phase.
The pseudo-code for the mapper and reducer of this job is depicted in Algorithm 5.

Algorithm 5: MapReduce job 1 -
Finding the maximum and minimum values for each column in the matrix X

```
function MAP( key, value )
    # key – Timestamp of the MDP. Metadata.
    # value – MDP. The delimiter between features is the character ','.
    x_n ← Parse value to tokens by a delimiter ','      # x_n is an array of size n
    n ← Number of tokens in x_n
    for iFeature = 1 to n do                  # For each feature (column)
      emit( iFeature, x_n[iFeature] )
    end for

function REDUCE( key, ⟨values⟩ )
    # key – Feature (column) index.
    # ⟨values⟩ – A list of {features}.
    max ← 0
    min ← ∞
    for all value ∈ ⟨values⟩ do
      if value > max then
            max = value
      end if
      if value < min then
            min = value
      end if
    end for
    emit( key, (max, min) )      # Output value is (max, min)
```

"*MapReduce job 2* Nearest-neighbours" finds the nearest-neighbours to y in the lower-dimensional space. The mapper generates a random subset for each permutation. Then, the mapper finds the nearest-neighbours MDPs for each permutation. The reducer finds the nearest-neighbours in the lower-dimensional space. The input to this job is the output from the preprocessing phase.

As a preliminary phase, a permutation matrix P of size $numPermutations \times n$ is saved in a separate file on HDFS. Each row in matrix P determines a specific permutation. Each row in P has n entries, each entry is '1' if the corresponding feature exists in the permutation or '0' otherwise. The number of 1's in each row, which is the number of features in the permutation, is chosen at random from the range $[minFeatures, maxFeatures]$. The delimiter between entries in each row is the character ','. An example of such a row in the matrix P is

$0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0$.

In this example, the permutation contains the features 2,3,4,5,8,11,12,19,20,23 and does not contain the features 1,6,7,9,10,13,14,15,16,17,18,21,22,25.

The pseudo-code for the mapper and reducer of this job is depicted in Algorithm 6.

Algorithm 6: MapReduce job 2 - Nearest-Neighbours (Mapper)

function MAP_SETUP
Called once at the beginning of the map task.
$P \leftarrow$ Read into memory from a file on HDFS.
$numPermutations \leftarrow$ Number of rows in matrix P
$(max, min) \leftarrow$ Read into memory from file on HDFS.
$\mu_1 \leftarrow$ Read as parameter.
$y_n \leftarrow$ Read as parameter. # y_n is an array of size n
$n \leftarrow$ Number of tokens in y_n.
for $iFeature = 1$ **to** n **do**
$y_n[iFeature] \leftarrow \frac{y_n[iFeature]-min}{max-min}$ # Normalize the initial designated MDP
end for

function MAP($key, value$)
key – Timestamp of the MDP. Metadata.
$value$ – MDP. The delimiter is the character ','.
$x_n \leftarrow$ Parse $value$ to tokens by the delimiter ',' # x_n is an array of size n
for $iFeature = 1$ **to** n **do**
$x_n[iFeature] = \frac{x_n[iFeature]-min}{max-min}$ # Normalize the input MDP
end for
# For each permutation in P:
for $iPermutation = 1$ **to** $numPermutations$ **do**
$distance \leftarrow 0$ # Initialize Euclidean distance
$permutation \leftarrow$ Parse line $iPermutation$ in P to tokens by the delimiter ','
for $iFeature = 1$ **to** n **do** # For each feature in the permutation $iPermutation$
$j \leftarrow 0$
if $permutation[iFeature] = 1$ **then**
$x_k[j] \leftarrow x_n[iFeature]$
$y_k[j] \leftarrow y_n[iFeature]$
$distance \leftarrow distance + (x_k[iFeature] - y_k[iFeature])^2$
$j \leftarrow j + 1$
end if
end for
$distance \leftarrow \sqrt{distance}$
if $value = y_n$ **then** # The MDP is the initial designated MDP
emit ($iPermutation, (timeStamp, 0, x_k)$)
end if
if $distance < \mu_1$ **then** # The MDP is nearest neighbour to the initial MDP
emit ($iPermutation, (timeStamp, x_n, x_k)$)
end if
end for

Algorithm 6: MapReduce job 2 Continuation - Nearest-Neighbours (Reducer)

```
function REDUCE_SETUP
    # Called once at the beginning of the reduce task.
    μ2 ← Read as parameter.

function REDUCE( key, ⟨values⟩ )
    # key – Permutation index.
    # ⟨values⟩ – A list of { (timeStamp, x_n, x_k) }.
    datasetN ← ∅
    datasetK ← ∅
    for all value ∈ ⟨values⟩ do
        x_n ← Extract x_n from value
        x_k ← Extract x_k from value
        timeStamp ← Extract timeStamp from value
        datasetN ← datasetN ∪ ( timeStamp, x_n )
        datasetK ← datasetK ∪ x_k
    end for
    k ← Number of elements in x_k
    δ ← ⌊k/2⌋        # Number of dimensions in the lower dimensional space
    samples ← size[datasetK]
    if δ < samples then                    # If there are enough samples in datasetK
        epsilon ← computeEpsilon(datasetK)        # Epsilon for DM
        dm ← diffusionMaps(datasetK, epsilon, δ)        # Embed datasetK to δ dimensions space
        map ← ∅        # Initialize a map of key-value pairs
        for iSample = 1 to samples do
            key ← Element iSample from datasetN
            value ← Element iSample from dm
            map ← map ∪ (key, value)
            x_n ← Extract x_n from key
            if x_n = 0 then
                y_δ ← value        # The designated MDP in the embedded space
            end if
        end for
        for all entry ∈ map do                # entry is a key-value pair from map
            x_δ ← get value from entry
            distance ← ‖x_δ − y_δ‖
            if distance < μ2 and distance > 0 then
                key ← get key from entry
                emit key        # Emit only key, which is ( timeStamp, x_n )
            end if
        end for
    end if
```

The functions "computeEpsilon" and "diffusionMaps" mentioned in Algorithm 6 are computed according to [15] and section 3.4, respectively.

"*MapReduce job 3* Pass threshold", which is the next MapReduce job, finds the MDPs that pass the threshold Ω. The input to this job is the output of the job "*MapReduce job 2* Nearest-neighbours". The mapper of this job is a transparent (identity) function. The reducer sums up the occurrences of each MDP and checks if this sum passes the threshold Ω. For optimization purposes, this job has an extra combiner which sums up the occurrences of each MDP for each mapper.
The pseudo-code for the mapper, combiner and reducer of this job is depicted in Algorithm 7.

Algorithm 7: MapReduce job 3 - Pass Threshold

function MAP($key, value$)
  # key – Offset, ignored.
  # $value$ – $(timeStamp, x_n)$.
  emit ($value, 1$)

function COMBINER($key, \langle values \rangle$)
  # key – $(timeStamp, x_n)$, the MDP which is μ_2-close to y_δ in the embedded space
  # $\langle values \rangle$ – A list of $\{1\}$.
  $sum \leftarrow 0$
  for all $value \in \langle values \rangle$ **do**
    $sum \leftarrow sum + value$
  end for
  emit (key, sum)

function REDUCE_SETUP
  # Called once at the beginning of the reduce task.
  $\Omega \leftarrow$ Read as parameter.

function REDUCE($key, \langle values \rangle$)
  # key – $(timeStamp, x_n)$, the EMDP which is μ_2-close to y_δ
  # $\langle values \rangle$ – A list of $\{count\}$,
  # $count$ is the number of 1's from each mapper after the application of the combiner.
  $sum \leftarrow 0$
  for all $value \in \langle values \rangle$ **do**
    $sum \leftarrow sum + value$
  end for
  if $\Omega \leq sum$ **then**
    emit (key)
  end if

The last MapReduce job in this flow is "*MapReduce job 4* Compares results to the ground truth". This job compares the results from the SDA to the ground truth data. It is used only for verification. The mapper emits the results that correspond to the ground truth. If a result does not appear, the map function will emit "FALSE POSITIVE" and the corresponding result. The reducer is an identity function. The input to this MapReduce job is the output of the MapReduce job "*MapReduce job 3* Pass threshold". The ground truth data of the security-events (attacks) was provided in [4]. The ground truth contains only the basic information about the events. It includes event type, source and destination IPs and ports and the start and the end time of the attack (day, hour and minutes). The ground truth data is arranged in an excel file. Each attack is depicted in a separate row in the excel file. An example of a row in the ground truth data file is:

Event Type	Source IP	Source Port	Destination IP	Destination Port	Start Time	Stop Time
spam bot	172.28.11.150	0	77.91.104.22	80	04/11/2009 04:30	04/11/2009 04:30

The pseudo-code for the mapper and reducer of this job is depicted in Algorithm 8.

Algorithm 8: MapReduce job 4 - Compares Results to Ground Truth (Mapper)

```
function MAP_SETUP
    workbook ← Get workbook from the excel file
    sheet ← Get sheet 0 from workbook
    numRows ← Number of rows in sheet
    startTimeCells ← Get all cells from column of "Start Time"
    startTimes ← Convert each cell from startTimeCells to milliseconds representation
    stopTimeCells ← Get all cells from column of "Stop Time"
    stopTimes ← Convert each cell from stopTimeCells to milliseconds representation
    eventsCells ← Get all cells from column of "Event Type"

function MAP( key, value )
    # key – timeStamp - Timestamp of the MDP. Metadata.
    # value – The MDP. The features are separated by the delimiter ','.
    time ← Convert key to milliseconds representation
    falsePositiveFlag ← true
    for iRow = 1 to numRows do
       if startTimes[iRow] ≤ time and time ≤ stopTimes[iRow] then
              emit ( eventsCells[iRow], (key, value) )
              falsePositiveFlag ← false      # This MDP is a real attack, so not false positive
              break for loop      # Terminate the for loop when the first attack is found
       end if
    end for
    if falsePositiveFlag = true then
       emit ( FALSE POSITIVE!, (key, value) )
    end if
```

Additional MapReduce job was performed prior to the testing phase. It was executed once for evaluation purposes only and are not part of the SDA. The input to this job is the output from the preprocessing phase. It counts the occurrences of the normal MDPs (not attacks) and each attack type in the input dataset to the SDA. The map function emits the type of the MDP as key and '1' as value. The type of the MDP is either 'Normal' or the attack type. The reduce function aggregates all the 1's and emits the sum.
The pseudo-code for the mapper and reducer of this job are depicted in Algorithm 9.

Algorithm 9: MapReduce job 5 - Counts the occurrences of the normal MDPs and each attack type in the input dataset to the SDA

```
function MAPPER_SETUP
    workbook ← Get workbook from excel file
    sheet ← Get sheet 0 from workbook
    numRows ← Number of rows in sheet
    startTimeCells ← Get all cells from column of "Start Time"
    startTimes ← Convert each cell from startTimeCells to milliseconds representation
    stopTimeCells ← Get all cells from column of "Stop Time"
    stopTimes ← Convert each cell from stopTimeCells to milliseconds representation
    eventsCells ← Get all cells from column of "Event Type"

function MAP( key, value )
    # key – timeStamp - Timestamp of the MDP. Metadata.
    # value – The MDP. The features are separated by the delimiter ','
    time ← Convert key to milliseconds representation
    normalFlag ← true
    for iRow = 1 to numRows do
      if startTimes[iRow] ≤ time and time ≤ stopTimes[iRow] then
            emit ( eventsCells[iRow], 1 )
            normalFlag ← false      # This MDP is an attack, so not normal
            break for loop      # Terminate the for loop when the first attack is found
      end if
    end for
    if normalFlag = true then          # If this MDP is normal (not an attack)
      emit ( Normal, 1 )
    end if

function REDUCE( key, ⟨values⟩ )
    # key – pattern type
    # ⟨values⟩ – A list of {1}.
    sum ← 0
    for all value ∈ ⟨values⟩ do
      sum ← sum + value
    end for
    emit ( key, sum )
```

5 Experimental Results On Networking Data

5.1 Description of the DARPA Data

DARPA 2009 intrusion detection dataset [21] was used for testing the SDA. The dataset was created to aid in the evaluation of networks IDSs performance. It was created with a synthesized traffic to imitate Internet traffic between a /16 subnet (172.28.0.0/16) and the Internet. The internal traffic inside the local subnet was not simulated. A simplified configuration of the simulated data is depicted in Fig. 5.1.

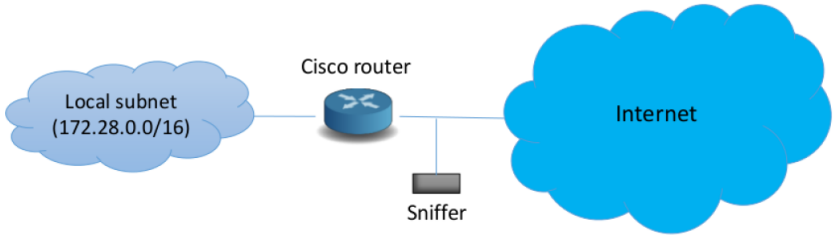


Figure 5.1: A simplified layout for capturing of the synthesized traffic in the dataset [21]

The dataset spans a period of 10 days between the 3rd and the 12th of November 2009. The dataset contains synthetic Transmission Control Protocol (TCP), User Datagram Protocol (UDP) and Internet Control Message Protocol (ICMP) packets. The dataset also contains other protocols, such as Logical Link Control (LLC) and Address Resolution Protocol (ARP), but their aggregated volume is negligible. The dataset contains a variety of different types of security events, such as DDoS, spambots, DNS redirecting, etc. The dataset consists of 7000 pcap files with around 6.6TB of total size. The size of each pcap file is 954MB. Each pcap file typically covers around one to two minutes time window depending on the traffic rate.

The number of normal MDPs and the number of selected attacks types MDPs from the 2009 DARPA dataset are depicted in table 5.1.

Normal MDPs	**Attacks MDPs**			
	DDoS	**Spambots**	**DNS Redirecting**	**Phishing**
11987 Normal	555 DDoS	89 spam bot	11 break-DNS exploit echo	87 phishing email exploit malware trawler
	153 noisy c2+ tcp control channel exfil nc	15 spambot client compromise	3 break-DNS home administrator attack scripts sdu	52 post-phishing client compromise + malicious download
	92 malware ddos	210 spambot malicious download	11 router-redirect home administrator attack scripts	38 post-phishing c2 exploit malware malclient.pl
	4 c2+ tcp control channel exfil - no precursor nc		261 out2in dns	31 post-phishing c2 heartbeat exploit malware malclie
			235 out2in	1 noisy phishing email exploit malware trawler

Table 5.1: Number of MDPs (normal and attacks) in the 2009 DARPA dataset

5.2 Cluster Information

The experimental tests and verification were conducted on Tel-Aviv university Hadoop-cluster. Hadoop version installed on the cluster was 2.5.1. 111 of active nodes are available where each has available of 8GB of RAM. Total memory available is 888GB.

5.3 Experimental Results on DARPA Data

In this section, we present the experimental results from the similarity detection algorithm applied to the 2009 DARPA dataset (section 5.1).

The permutation matrix used for the experiments was of size 50×25, i.e., 50 permutations with 25 features for each permutation. The features were defined in section 4.1. The parameters of the permutation matrix were chosen to be $minFeatures = 8$ and $maxFeatures = 15$.

The time interval chosen for the features extraction was one minute. This parameter was chosen empirically after several tests with a smaller time interval, such as 10 milliseconds, 0.5 seconds, etc, that resulted in a high false alarm rate. A time interval of one minute fully describes an attack. The experimental results were performed on several attacks types, as explained in section 3.3. The experimental results of the algorithm for several attack types as the initial designated MDP are shown in table 5.2.

Reference Attack Type (Initial MDP)	**Parameters**				**Results**			
	μ_1	μ_2	Ω	***max Iterations***	**Number of Same Attack. True Positive**	**Number of Other Attacks. True Positive**	**Number of False Positive**	**False Alarm Rate** [%]
ddos	0.45	0.028	50	1	41	253	728	6.07
noisy c2+ tcp control channel exfil nc	0.15	0.035	50	1	23	198	514	4.28
noisy c2+ tcp control channel exfil nc	0.15	0.02	50	1	3	12	42	0.3
noisy c2+ tcp control channel exfil nc	0.15	0.02	50	5	14	148	480	4
spam bot	0.2	0.02	50	1	4	65	205	1.7
spambot malicious download	0.2	0.02	50	1	27	199	558	4.65
out2in dns	0.5	0.03	47	1	15	109	356	2.9
out2in	0.5	0.03	48	1	18	187	492	4.1
phishing email exploit malware trawler	0.2	0.01	50	1	11	280	812	6.7
post-phishing client compromise + malicious download	0.4	0.03	50	1	5	380	803	6.69

Table 5.2: Results from the 2009 DARPA dataset. The false alarm rates in % were computed in relation to table 5.1

where false positive is a result that indicates that a given condition has been satisfied, when it actually not. In our case, a false positive means that the algorithm wrongfully identifies a 'normal' MDP as an attack. False alarm rate is defined as the number of 'normal' patterns classified as attacks (false positive) divided by the total number of 'normal' patterns mentioned in table 5.1.

The results show that the SDA detects also other types of attacks, different from the initial designated MDP. For example, the results for the initial designated MDP of the type "noisy c2+ tcp control channel exfil nc" indicate the detection of attacks of types "c2+ tcp control channel exfil nc", "client compromise exfil sams launch vulnerable cli", "ddos", "failed attack exploit iis asp overflow", "failed attack or scan", "exploit/bin/iis nsiislog.pl", "failed attack or scan exploit/bin/webstar ftp user", "noisy c2+ tcp control channel exfil nc", "noisy client compromise + malicious download exfil", "noisy blackhole exploit echo", "post phishing client compromise + malicious download", "scan /usr/bin/nmap", "spam bot" and "spambot malicious download".

The results corresponding to the column "Number of Same Attack. True Positive" in table 5.2 are the numbers of attacks correctly identified by the SDA that are the same type as the initial designated MDP. The results corresponding to the column "number of other attacks. true positive" in table 5.2 are the numbers of attacks correctly identified by the SDA that are of different type from the initial designated MDP.

Performance Evaluation

To manifest the scalability of the SDA, we measured its completion time as a function of its input data size. The performance evaluation was performed on a cluster and on a single Hadoop-node. Figure 5.2 illustrates that 10 iterations of the SDA performed on a cluster were completed within 10 minutes for 1TB and 24 minutes for 6TB, which is over 3 and 10 times faster than their completion time performed by only one node. These results illustrate the increased performance enhancement as the volume of the input traffic becomes larger.

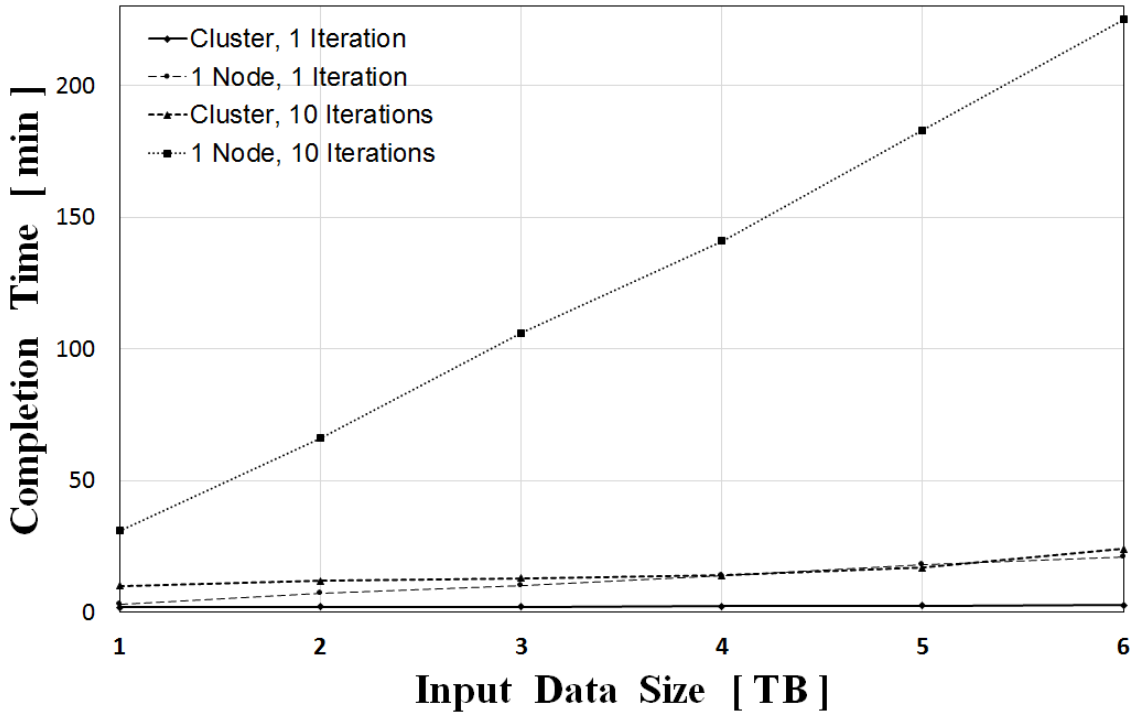


Figure 5.2: Completion time of the SDA regarding various sizes of the input data

6 Conclusions and Future Work

The work described in this paper is concerned with the development and application of a similarity detection algorithm to big-data using MapReduce methodology implemented on Hadoop cluster.

The experimental tests were performed on the 2009 DARPA intrusion dataset. The proposed approach aims at detecting intrusion attacks in a networking dataset. The proposed approach aims at gaining maximum detection of each attack with minimum false positive rate. The experimental results show that the proposed algorithm detects also other types of attacks, different from the initial designated MDP. This approach will be very useful for the attacks detection in today's changing attack methodologies.

Although the results presented here have demonstrated the effectiveness of the similarity detection algorithm, it could be further developed in a number of ways:

1. **Extending the algorithm to use a more sophisticated features selection:** Using the best quality of features, which represent the whole data while removing redundant and irrelevant features, is an important task of anomaly detection systems due to the immense amount of data they need to process. Thus, a more sophisticated method for each subset features selection, such as in [35], should increase the detection rate and decrease the false positive rate.

2. **Extending the algorithm to implement the preprocessing phase on Hadoop:** Implementing the preprocessing phase of features extraction as several MapReduce jobs or as any other additional software packages that can be installed on top of or

alongside Hadoop (e.g. Apache Pig, Apache Hive, Apache HBase, etc) will help to perform the entire process as one unit.

3. **Dividing the attacks search to separate cases according to protocol:** Using only the features that where extracted from a specific protocol to find the attacks that are related to this protocol. For example, using only TCP/IP related features to find TCP/IP related attacks, such as DDoS. This can potentially reduce the false alarm rate.

Acknowledgments

This research was partially supported by the US-Israel Binational Science Foundation (BSF 2012282), Israel Ministry of Science & Technology (Grants No. 3-9096, 3-10898), and Blavatnik Computer Science Research Fund.

References

[1] Apache hadoop 0.23 is here! `http://hortonworks.com/blog/apache-hadoop-is-here/`, 2011.

[2] Moving ahead with hadoop yarn. `http://www.ibm.com/developerworks/library/bd-hadoopyarn/`, 2013.

[3] Apache hadoop. `http://hadoop.apache.org/`, 2015.

[4] DARPA 2009 intrusion detection dataset. `http://www.darpa2009.netsec.colostate.edu/`, 2015.

[5] HTCondor - High Throughput Computing. `http://research.cs.wisc.edu/htcondor/htc.html`, 2015.

[6] Wireshark. `https://www.wireshark.org/`, 2015.

[7] T. Abbes, A. Bouhoula, and M. Rusinowitch. Protocol analysis in intrusion detection using decision tree. In *Information Technology: Coding and Computing*, volume 1, pages 404–408, 2004.

[8] I. Aljarah and S.A. Ludwig. Mapreduce intrusion detection system based on a particle swarm optimization clustering algorithm. In *Evolutionary Computation (CEC)*, pages 955–962, 2013.

[9] M. Bahrololum, E. Salahi, and M. Khaleghi. Anomaly intrusion detection design using hybrid of unsupervised and supervised neural network. *International Journal of Computer Networks and Communications*, 1, 2009.

[10] Daniel Barbar and Sushil Jajodia. *Applications of Data Mining in Computer Security*. Springer Science and Business Media, 2012.

[11] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys (CSUR)*, 41:1–58, 2009.

[12] A. Chauhan, G. Mishra, and G. Kumar. Survey on data mining techniques in intrusion detection. *International Journal of Scientific and Engineering Research*, 2, 2011.

[13] J.J. Cheon and T.Y. Choe. Distributed processing of snort alert log using hadoop. *International Journal of Engineering and Technology*, 5:2685–2690, 2013.

[14] R.R. Coifman, S. Lafon, A.B. Lee, M. Maggioni, B. Nadler, F. Warner, and S.W. Zucker. Geometric diffusions as a tool for harmonic analysis and structure definition of data: Diffusion maps. *Proceedings of the National Academy of Sciences of the USA*, 102:7426–7431, 2005.

[15] R.R. Coifman, Y. Shkolnisky, F.J. Sigworth, and A. Singer. Graph laplacian tomography from unknown random projections. *Trans. Image Process.*, 17:1891–1899, 2008.

[16] P.J. Criscuolo. Distributed denial of service. Technical report, Department of Energy Computer Incident Advisory Capability, 2000.

[17] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[18] J. Dean and S. Ghemawat. Mapreduce: A flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[19] D.K. Denatious and A. John. Survey on data mining techniques to enhance intrusion detection. In *Computer Communication and Informatics (ICCCI)*, pages 1–5. IEEE, 2012.

[20] Q. Fu, J.G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining*, pages 149–158, 2009.

[21] M. Gharaibeh and C. Papadopoulos. DARPA-2009 Intrusion Detection Dataset Report. Technical report, Colorado State University, 2014.

[22] M. Hein and J.Y. Audibert. Intrinsic dimensionality estimation of submanifold in Rd. In Y. Weiss, B. Schölkopf, and J. Platt, editors, *Proceedings of the 22Nd International Conference on Machine Learning*, pages 289–296. ACM, 2005.

[23] K. Hwang, M. Cai, Y. Chen, and M. Qin. Hybrid intrusion detection with weighted signature generation over anomalous internet episodes. *Dependable and Secure Computing, IEEE Transactions on*, 4:41–55, 2007.

[24] P.J.F. Groenen I. Borg. *Modern multidimensional scaling: Theory and Applications*. Springer, 1997.

[25] M. Janbeglou and M. Zamani. Redirecting network traffic toward a fake DNS server on a LAN. 2010.

[26] H.J. Jeong, H. WooSeok, L. Jiyoung, and Y. Ilsun. Anomaly teletraffic intrusion detection systems on hadoop-based platforms: A survey of some problems and solutions. In *Network-Based Information Systems (NBiS)*, pages 766–770, 2012.

[27] P. Jian, S.J. Upadhyaya, F. Farooq, and V. Govindaraju. Data mining for intrusion detection: techniques, applications and systems. In *Data Engineering*, pages 877–877, 2004.

[28] C. Kruegel and T. Toth. *6th International Symposium*, chapter Using Decision Trees to Improve Signature-Based Intrusion Detection, pages 173–191. Springer Berlin Heidelberg, 2003.

[29] K.Shvachko, H. Kuang, S. Radia, and R. Chansler. *Hadoop: The Definitive Guide*, chapter The Hadoop Distributed File System, pages 1–10. O'Reilly Media, 2010.

[30] V. Kumar and O.P. Sangwan. Signature based intrusion detection system using SNORT. *International Journal of Computer Applications and Information Technology*, 1:35–41, 2013.

[31] G.D. Kurundkar, N.A. Naik, and S.D. Khamitkar. Network intrusion detection using SNORT. *International Journal of Engineering Research and Applications*, 2:1288–1296, 2012.

[32] S. Lafon. *Diffusion Maps and Geometric Harmonics*. PhD thesis, Yale University, 2004.

[33] J.R. Lee, Y. Sang-Kug, and H.D.J. Jeong. Detecting anomaly teletraffic using stochastic self-similarity based on hadoop. In *Network-Based Information Systems (NBiS)*, pages 282–287, 2013.

[34] J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014.

[35] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 2000.

[36] M. Loukides. *Applications of Data Mining in Computer Security*. O'Reilly Media, 2009.

[37] B. Mukherjee, L.T. Heberlein, and K.N. Levitt. Network intrusion detection. *Network, IEEE*, 8:26–41, 1994.

[38] R. Nene and M.J. Nene. A survey on latest DoS attacks : Classification and defense mechanisms. *International Journal of Innovative Research in Computer and Communication Engineering*, 1:1847–1860, 2013.

[39] S. Omatu, M.P. Rocha, J. Bravo, F. Fernndez, A. Bustillo E. Corchado, and J.M. Corchado. *Distributed Computing, Artificial Intelligence, Bioinformatics, Soft Computing, and Ambient Assisted Living*, chapter Design of a Snort-Based Hybrid Intrusion Detection System, pages 515–522. Springer Berlin Heidelberg, 2009.

[40] L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *In Proceedings of ACM CSS Workshop on Data Mining Applied to Security*, pages 5–8, 2001.

[41] M. Roesch. Snort – lightweight intrusion detection for networks. In *LISA '99 Proceedings of the 13th USENIX conference on System administration*, pages 229–238. USENIX Association Berkeley, 1999.

[42] R.R.Coifman and S. Lafon. Diffusion maps. *Appl. Comput. Harmon. Anal.*, 21:5–30, 2006.

[43] A. Almomani A. Mishra S. Tripathi, B. Gupta and S. Veluru. Hadoop based defense solution to handle distributed denial of service (DDoS) attacks. *Journal of Information Security*, 4:150–164, 2013.

[44] C. Sanders. *Practical Packet Analysis: Using Wireshark to Solve Real-World Network Problems*. William Pollock, 2011.

[45] S. Shamshirband, A. Amini, N.B. Anuar, L.M. Kiah, Y.W. Teh, and S. Furnell. D-FICCA: A density-based fuzzy imperialist competitive clustering algorithm for intrusion detection in wireless sensor networks. *Measurement*, 55:212–226, 2014.

[46] W. A. Shewhart. *Economic Control of Quality of Manufactured Product*. D. Van Nostrand Company, New York, 1931.

[47] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Beowulf cluster computing with linux. chapter Condor: A Distributed Job Scheduler, pages 307–350. MIT Press, 2002.

[48] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, 2005.

[49] T. Verwoerd and R. Hunt. Intrusion detection techniques and approaches. *Computer Communications*, 25:1356–1365, 2002.

[50] K. Wankhade, S. Patka, and R. Thool. An overview of intrusion detection based on data mining techniques. In *Communication Systems and Network Technologies (CSNT)*, pages 626–629. IEEE, 2013.

[51] M.E. Whitman and H.J. Mattord. *Principles of Information Security*. Course Technology, 2011.

[52] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, 2(13):37 – 52, 1987.

[53] L. Yeonhee, K. Wonchul, and L. Youngseok. A hadoop-based packet trace processing tool. In *Proceedings of the Third International Conference on Traffic Monitoring and Analysis*, pages 51–63. Springer-Verlag, 2011.

[54] L. Yeonhee and L. Youngseok. Detecting DDoS attacks with hadoop. In *Proceedings of The ACM CoNEXT Student Workshop*, pages 7:1–7:2. ACM, 2011.

[55] D.Y. Yeung and C. Chow. Parzen-window network intrusion detectors. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 385–388, 2002.

[56] L. Youngseok, W. Kang, and H. Son. An internet traffic analysis method with mapreduce. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pages 357–361, 2010.

[57] S.T. Zargar, J. Joshi, and D. Tipper. A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *Communications Surveys Tutorials*, 15:2046–2069, 2013.

[58] S. Zhong, T. Khoshgoftaar, and N. Seliya. Clustering-based network intrusion detection. *Int. J. Rel. Qual. Saf. Eng.*, 14, 2007.