Detecting outliers in web-based network traffic

Joël Stemmer j.stemmer@student.utwente.nl

June 4, 2012

Contents

Abstract

Regular anomaly detection approaches require the full network payload data or low-level access to the system. In cases where this kind of information is not available because of limited system access, encrypted data or privacy reasons these approaches cannot be used. We present an anomaly detection technique for these cases using an outlier detection algorithm. The individual requests from a request log are grouped together to reconstruct the original sessions. These sessions form a new dataset from which anomalies can be detected using a Self-Organizing Map. We train the Self-Organizing Map with a subset of the sessions and then perform the outlier detection on the rest of the dataset. Using this approach we are able to identify several automated attacks, however the lack of information in the individual requests make it hard to distinguish regular user behavior from manually crafted attacks.

Chapter 1

Introduction

The way we use the internet has changed significantly the last few years. Browser capabilities have increased at a fast pace, offering many features that were previously only available to local offline applications. Features such as local storage, a graphics canvas, media playback, filesystem API and fast javascript interpreters allow for the creation of rich web-based applications. Many desktop applications are being replaced by web-based equivalents, so that they can be accessed from any place and any device with a network connection.

However, moving functionalities and data from the desktop to the web also creates an interesting target for hackers. Attackers have different reasons for attacking these systems, from obtaining privacy sensitive data to completely shutting down the system and this will have consequences for a large number of users. In general, attacks can be categorized to target the Confidentiality, Integrity and/or Availability of a system and/or data.

The majority of the services running on servers keep track of errors, warnings and user actions during operation in logfiles. It can quickly become tedious or even impossible for someone to manually sift through all the collected data to find any evidence of malicious activities. If such malicious activities stand out in some way compared to regular activities, we might be able to find them in large collections of data. According to OWASP[20], the top web application security risks in 2010 included (SQL) injection, Cross-Site Scripting, Cross-Site Request Forgery and broken authentication and session management. Many of these attacks are related to unsanitized user input. SQL injection cannot only just affect the availability by dropping tables or the integrity by changing data the user should have no access to, but can even in extreme cases compromise the entire server by uploading and executing files. Cross-Site Scripting and Cross-Site Request Forgery are used to retrieve confidential information from other users, for example by stealing the cookies of a user with higher authorizations.

1.1 Outline of the thesis

This increased usage of webbased services also leads to an increased amount of information begin stored and logged. The most common information being logged is directly related to the activity performed by the users of the webservices, where each request from a user is directly translated into an entry in this source containing various pieces of data related to that client and request. This is an important source of information when looking for irregular or malicious user activity.

However, there are several reasons why this source of data may only contain a limited amount of information per request. When outsourcing the task of hardware configuration and maintenance, which is the case when using cloud services for example, a lack of control over the infrastructure prevents low level access to the network. The usage of encrypted network connections is another reason why it may be difficult to have access to the actual datastream. Proprietary software may lack the ability of logging certain kinds of data without any options of changing it, or software configured for high performance may intentionally choose not to log certain pieces of data.

We start in Chapter 2 by looking at specific problems with regard to detecting malicious activity in our case, the threat model and present our research question.

There are different techniques to analyze web requests, some of them simply look for predefined

patterns to identify malicious activities, others use more sophisticated machine learning techniques. In Chapter 3 we take a look at different anomaly detection techniques. Many of them use the actual network packets or the payload of the web requests and tend to focus on detecting specific kinds of attacks for specific services. This information might not always be available for various reasons. For one, capturing all network packets requires low-level access to a range of systems, such as load balancers and web servers. Accessing the contents of these packets can be simply impossible in case of encrypted traffic, or in some cases even illegal due to privacy laws. Furthermore, the data can come from a number of different sources and in different formats, so the amount and type of information can vary from service to service. For servers that are always running the information flow never stops, resulting in a continuous stream of data.

Instead of focusing on detecting a single type of attack for a single service, a more general approach is needed for our problem to find anomalous activities. The goal of outlier detection is finding unusual objects in large datasets. It is used, for example, in credit card fraud detection and marketing to find the kind of information that is unusual or unexpected. Chapter 4 shows the different approaches one can take to detect outliers, how they should be used and how they perform.

Next we discuss the datasets that are available to us, how they were obtained and what kind of data it contains in Chapter 5, along with the steps required to prepare the datasets for use with an outlier detection algorithm. Using the previously discussed threat model several attacks have been made and included in the datasets.

Chapter 6 presents our approach to transform the datasets so that they become suitable for an outlier detection algorithm, followed by the actual experiments and results we obtained in Chapter 7. Finally we end with the conclusion in Chapter 8.

Chapter 2

Problem statement

We will discuss several intrusion detection techniques in the next chapter. While every technique focuses on detecting a certain kind of attack aimed at exploiting an application's weakness, the majority focus on attacks related to improper handling of input data. This focus requires access to the actual payload data passing from the client to the server. When working with network based intrusion detection this data is often available since you have access to the physical hardware. For webbased intrusion detection the payload is not always as easy to obtain, especially when working with large collection of logfiles. These logfiles are a large source of information related to the activity the users performs, but often lacks the actual data the user creates or modifies because of various reasons such as storage space considerations or privacy issues.

Because of these issues we are unable to use the existing anomaly detection techniques on these logfiles. We will have to look at other ways of dealing with this data and extracting meaningful information to help us detect anomalous behavior. Outlier detection is one of the techniques commonly used in for example fraud detection, and it might be a solution to finding any strange behavior in large collections of data.

2.1 Research question

Based on this problem we reach the following question:

Can we use outlier detection to detect anomalous activity in logs of webbased network traffic lacking the payload data?

To answer this research question we will look in depth at existing webbased anomaly detection techniques and how they might be used. We will look at different outlier detection techniques, how they work and on what kind of data they are used. Finally we propose a way of applying an outlier detection algorithm to our datasets and analyze the results to see whether we are able to detect various attacks.

We have a large collection of logfiles containing usage data, obtained from webbased services such as webservers and web application servers. These logfiles only contain a small amount of information for each individual request handled by the servers. Given the privacy sensitive nature of the data that is accessible through these servers, security measures are in place to prevent unauthorized access. The software these webservers rely on is kept up to date at all times and there are other security measures in place such as two-factor authentication, preventing password-guessing attacks etc. Furthermore, the application software has also been audited by an external company. However, if an attacker somehow manages to gain access to the system we still want to be able to detect and report it as soon as possible to be able to take counter measures.

2.2 Threat Model

In the threat model we look at what we want to protect, our assets, and how they might be at risk. The kind of attacks or malicious behavior we are interested in finding is also described. Our goal is to detect

these threats in order to protect our assets. The threat model will also help us craft our attacks that will be performed during the experiments in which we test our approach.

2.2.1 Assets

The assets of a web application in general is the data of its users. In our specific case this consists of financial, medical and personal information of our users and their clients for our first dataset. Thousands of users need access to this data every single day, making the availability of this data important. Access to the data is given to authenticated and authorized users only, through our web application.

2.2.2 Application software

The most common attacks on web applications are input related attacks. The currently popular programming languages have either native support, application frameworks or libraries specifically designed for webbased use. These are either part of the official language specification or have been created by the community. Their goal is to simplify or automate the common tasks used by web application, such as handling user input, accessing the database, session and cookie management, authorization and authentication. Correct usage of these tools ensures that your applications is protected against these previously mentioned input based attacks.

Another kind of attack that is hard to protect against are attacks that do not rely on input related vulnerabilities. Situations in which unauthorized access is taking place in the system, with the goal of attacking the confidentiality, integrity or availability of the system are examples of such attacks. There are ways in which attackers can gain access to the system without having to exploit any weaknesses in the software itself, for example by using social engineering techniques. Confidentiality can be compromised when an attacker has gained unauthorized access and automatically gains all the access permissions of the compromised account. If this account also happens to have permissions to modify or delete information, the integrity and availability of the system can be compromised.

From the point of view of the application, these might as well have been valid actions performed by an authorized user. While they are all correct application workflows, we expect them to differ from regular user behavior, which if this assumption is correct allows us to detect them. These kinds of threats can be dangerous, especially in situations concerning privacy sensitive client data or corporate information, since they are not directly related to a single web request. The goal is to detect these kinds of attacks, using collected request logs, without requiring any low level access to the application servers or network payload data.

Given the limited amount of information contained in the request logs, the threats we are trying to identify will be specific for a small number of situations. They will be relevant given that requests logs often contain similar types of information, such as timestamps, user identifying data, information related to the request itself and possibly parts of the request payload.

2.2.3 Threats

The main focus of existing anomaly detection techniques is in the detection of input related attacks. Clients have a lot of control over the data being sent to the servers and they can use that to modify the data in a way that the servers do not expect. Attacks of these kinds are for example SQL injection attacks targeting the server database, or XSS attacks targeting other clients using that server. They achieve their goals by looking at individual network packets or web requests and analyzing the contents of these requests. The downside of this approach is that they ignore the bigger picture, each request is often part of a session.

Since we do not have access to the payload data, we cannot expect to detect the same attacks as the existing anomaly detection techniques. This also means we will not be able to identify attacks by a single request or network packet. Our focus will therefore be more towards workflow attacks described earlier, automated attacks and attacks on the integrity of the data rather than the confidentiality. So instead of trying to detect and prevent unauthorized read and write access to data we will focus on cases where an authorized user, whether this is an attacker using compromised credentials or an actual user with malicious intent, tries to access or modify data.

With this in mind we will focus on the following threats:

1. **Unauthorized data access**
   The collected request logs usually contain information that can identify clients. Based on this data a model can be constructed for each client describing the normal activities it performs. Any actions that are taken by that client that significantly differ from this model can be marked as anomalous. For example accessing large amounts of information in a short period, at irregular times or from multiple different user accounts.

2. **Modification or removal of information as acts of data vandalism**
   User identifying data can be used similar to the first threat, except in this case we do not look at the frequency or time information of a request, but the actions performed and leading up to this request by the user.

3. **Manipulation of request data with the intent of deceiving the system**
   While the previous two threats could be performed by simply using the provided user interface, another common source of attack is modifying or manipulating the actual payload send to the server. A common method is parameter injection, in which you send additional pieces of information to the server that was not part of the attack. An example of this threat is an online game contest, where people may try to submit anomalous scores by modifying the data sent to the server.

Chapter 3

Anomaly detection

The two most common approaches to detecting web-based attacks are signature-based detection and anomaly based detection. Signature-based detection relies on detecting patterns of known attacks to identify malicious behavior.While they are accurate, they have to be kept up-to-date with current attacks to be effective. Any attacks that are not in the signature or pattern database will therefore not be detected. This weakness can be exploited by creating different versions of a single attack.

Anomaly-based detection relies on statistical analysis of the data to find behavior that deviates from the normal activity. One of the big advantages over signature based attacks, if used correctly, is that it is able to detect variations of attacks or even completely new attacks. However, this could also result in normal activity being flagged as malicious.

The quality of various algorithms can be measured based on the number of false negatives and false positives they generate. A combination of these methods is possible for attempting to keep both the false negative rate and false positive rate as low as possible.

An important aspect of anomaly detection is deciding how to classify what is anomalous and what constitutes normal activity. For example, one could look at individual HTTP requests and their payload. A request in a typical webserver logfile often contains some of the following attributes: the request path and query parameters, user agent, HTTP version, referrer, time of the request, etc. Of all of these attributes, the request path and query parameters are the ones most likely to be useful since they contain the most important data processed by the server application. Another example would be to look at the behavior of the user by tracking usage patterns based on the pages he visits.

3.1 Anomaly Detection of Web-based Attacks

The anomaly detection described by *Kruegel and Vigna*[16] works on individual requests. The focus is mainly on the detection of various data input related attacks as described in Chapter 1, by analyzing various aspects of the request path of each request. The URI associated with each request (minus the domain name) is divided into three parts. The path, which consists of the resource path and program, and the parameters and their values. A program in this context, also called a resource, is defined by the last part of the path in the URI before the parameters start. Only HTTP GET requests that generated a response code by the web server indicating success[1] were used. This dataset is further reduced by eliminating any requests that do not contain any query parameters. The reasoning behind this is that they are focusing on the associations between programs (base request path) and parameters and their values. Finally, the set is partitioned into subsets based on their resource path. The analysis is run separately on each subset and will generate have a binary result for each request indicating either normal or anomalous.

The anomaly detection process uses models for each of the following attribute characteristics:

- Length
- Character distribution

[1] A response code between 200 and 300 is defined by the HTTP standard to indicate success

- Structural inference (Markov model)
- Token finder (detects enumerations, flags)
- Presence or absence
- Attribute order

The anomaly score for a request is based on the weighted sum of each individual score. When this score exceeds a predetermined threshold then the request is marked as anomalous.

The system must run in a training phase to determine the weights and thresholds. Only valid requests should be used for training the models to prevent anomalous requests from being incorrectly classified as valid. During the training a profile is created for each program and its attributes by evaluating requests using the different models. Based on these profiles, the different thresholds are calculated. These can be manually adjusted to fine-tune the results, either choosing for better detection accuracy or a lower false-positive rate.

Three datasets are used to test these models, *Apache* log files from servers at Google, the University of California and the Technical University Vienna. They have full access to the log files from the universities, so any malicious entries for the initial learning phase are removed. Malicious attacks were then injected in these log files, so that the effectiveness of the models could be tested. These attacks consisted of various real-world exploits, such as buffer overflows attacks, directory traversal attacks, several cross-site scripting attacks. All attacks were detected without requiring any adjustments after the initial learning phase, with low false positive rates ($< 0.2\%$).

3.2 Spectrogram

Song et al.[24] present a system, which is similar to the one described in Chapter 3.1, analyzing individual HTTP requests, but operating on a lower level. The major difference is that both HTTP GET and HTTP POST requests are analyzed and the entire request path including query parameters are treated as a single object. For a POST request, the request body containing the POST data is also used. It uses a combination of n-grams and Markov chains to calculate an anomaly score for this particular request.The given string is scanned and probabilities are calculated for the sequence of characters that occur in this string. It uses Expectation Maximization to find the optimal settings given the gram-size and the number of Markov chains to use in the training phase.

The Spectrogram system was tested using real data from two university web servers which was collected over a period of a month. These servers contained various scripts for the computer science department and personal homepages of students. Both of which can be interesting targets for attackers. Normalization is performed on the collected data by unescaping strings, removing whitespaces and numbers and converting all characters to lowercase. A manual inspection of the data ensures that the final dataset does not contain attacks of any kind. Finally all duplicate requests are removed to prevent creating a bias towards requests that occur more often than others. The resulting dataset was then used to train the model.

The attack-data includes remote file inclusion attacks, JavaScript and XSS[2] attacks, SQL injection and many unique shell code samples. The results were overall pretty good, with exceptional results in detecting worms, shell code attacks, SQL and XSS attacks. This is no surprise given the fact that these attacks usually contain a different set of characters compared to regular text and URI's. The area in which it is less effective is indeed the case of remote file inclusion, since it is hard to differentiate between URI's in the request and those in the query string. A false positive rate of about 1% is achieved in the first unbiased dataset. An even lower false positive rate of about 0.00006% is achieved on a dataset containing real life data collected over the period of one month.

[2] cross-site scripting

3.3 Detecting Anomalous and Unknown Intrusion Against Programs

Employing a neural network to detect malicious activity is proposed by *Ghosh et al.*[11]. A back propagation network is created which consists of a variable number of input nodes, ranging from 8 to 83, a single hidden layer with 125 nodes and one output node indicating positive or negative for the given input. The input dataset expected is a single string of data, in the case of this paper the input data to a printing program. This approach could be considered in a similar situation as described in Chapter 3.2 given the similarities in input data. Like previous systems, the neural network has to be trained prior to usage.

Experiments are performed in two different situations:

- For the black-box experiments, the authors use only data passed to the program, without having access to the programs source or state.
- For the white-box experiments, in addition to data used in the black-box experiments, they use internal program state data, which is only available when having access to the program sourcecode.

In contrast to the previous detection systems, best results are achieved when both normal and anomalous data is used in the training phase. However, the training data has to be correctly classified, so manual inspection of the training data is still required. A false positive rate of 0% is achieved in all experiments, however average false negative rates ranging from 0.9% in the best case to 19.8% in the worst case means not all data was correctly classified. Best results are obtained using a diverse input dataset of regular data, malicious data and randomly generated data (which is classified as negative). These achieve a true positive and true negative with an average of 99.1% and an average false negative rate of 0.9%. Even though a false negative will result in 'noise' which has to be handled by someone, it is probably more desirable than a false positive since then a potential attack might be ignored.

3.4 SPHINX

A method is described by *Bolzoni and Etalle*[6] to identify malicious requests. It uses several methods dependent on the type of data encountered similarly to Chapter 3.1. During the training phase, for each resource, each parameter is classified as one of three options: numerical, short text, raw data by analyzing the data in the different requests. Finding unexpected data in numerical fields helps detection of various input related attacks on fields that should only contain numerical data (most likely SQL injection). Short text fields usually contain predictable data related to the application flow, such as page names or application states. Unexpected data in these areas can signal attempts at accessing restricted pages or breaking application flow. Raw data contains everything else, which requires other techniques for detecting malicious data.

Input data consists of the public DARPA 1999 data set and a private data set collected from a web server at the University of Twente. Results on the first dataset were excellent, achieving a 100% detection rate and 0% false positive rate. For the second dataset, the system is trained with a subset of the data where any malicious activity was removed. It achieves a detection rate of 100% with a false positive rate of only 0.2%, which is in the same range as the other systems discussed.

3.5 Flow based intrusion detection

All the approaches taken so far rely on the availability of detailed information inside a single request or network packet. Situations with limited amounts of information in a request or where most of the traffic is encrypted will not provide the data these algorithms require. *Sperotto*[25] focuses on network intrusion detection as opposed to web-based intrusion detection, by looking at SSH and DNS data. Since SSH traffic is encrypted, it is not possible as an observer to detect anomalous behavior by looking at the payload. The observed packets within a time frame are grouped together based on properties they might have in common, such as IP addresses, ports and protocol to form a flow. These flows have certain properties of their own, regardless of the payload contents of individual packets, including flows per

second, packets per second, bytes per second and number of packets in a flow. In this case, the flows per second measurements are used to classify flows as benign or malicious.

A model consisting of two states is constructed based on Markov Chains. The two states indicate either activity, where SSH traffic was observer, or inactivity. The dataset consists of real traffic collected from the University of Twente network. Only benign traffic is used to train the model. Based on this trained model, threshold values can be assigned to traffic flows. Classification of the flows are done based on these values, where flows exceeding a certain threshold are marked as malicious.

After training the model, two synthetic and two original data sets are used for testing. The original data is network traffic captured from the University of Twente network. Each of these data sets contains both malicious and normal traffic, the malicious data is manually labeled for the datasets containing real network traffic. The results varied between the synthetic and original data sets, where the results were significantly better for the synthetic data sets. As previously mentioned, there is always a trade-off between a good detection rate and a low false positive rate.

3.6 Swaddler

Cova et al.[8] take another look at anomaly detection by focusing on the internal application states instead of the external data access. While they acknowledge the importance of detecting input validation related attacks, they also note that there are a large number of techniques available to handle these kinds of attacks. Instead they describe a method of detecting workflow violation attacks called Swaddler. They are able to detect attacks that the mainstream techniques based on request analysis were unable to detect by keeping track of the internal application state and the data coming in from the requests. The internal application state consists of all the information related to a user, which is the user session stored on the server, cookies sent and received from the client and other information such as hidden form variables and request parameters. Before the system is able to detect anomalous activity it must be trained. The training phase builds models for the application based on attack-free application usage. This is done by capturing the values of all the internal variables and describe characteristics of these variables in a set of statistical models. These models are then used in the detection phase to identify anomalous application states. This solution requires lowlevel access to the execution environment on the server and underlying PHP interpreter.

3.7 Conclusion

We discussed several existing approaches to anomaly detection in this chapter. They require a lot of information to operate, in most cases complete access to the request payload. In some cases they need full access to the system or source code, or even require you to patch the software running your application.

The paper about flow based intrusion detection in Chapter 3.5 takes a different approach. Because the data consist of encrypted traffic (SSH) or of itself contains little information (DNS), the actual data flows are reconstructed. We will take a similar approach for the collected data in our case. However, once these flows are recreated, we will need a way to identify the flows that are different from the rest. To achieve this we will use an outlier detection algorithm. The following chapter will give an overview of the different approaches one can take when looking for outliers in a dataset.

Chapter 4

Outlier detection

Outliers are objects in a large dataset that are inconsistent with the remaining set of data. In the field of data mining it is interesting to find patterns or hidden relations in large collections of data. However, outliers are regarded here as unwanted noise which has to be ignored or filtered out to prevent it from influencing the outcome [13, p.451]. Some of the techniques discussed rely on this notion to find the outliers by using some of the algorithms to collect these outliers instead of their initially intended purpose to filter them out. Examples of this are discussed in Chapter 4.2 which uses clustering algorithms to find the outliers. Other approaches are only concerned in detecting outliers. The detection of outliers has a number of useful applications, most notably in fraud detection and security. In this case we look at outlier detection to detect abnormal user behavior for web-based applications.

The two most important factors contributing to the complexity of the algorithms are the dataset size and dimensionality. The size of dataset, usually indicated with the letter N, is the number of objects, items or points contained in the entire dataset. Each of these objects may have one or more values, attributes or properties associated with it, also called the dimensionality. For example a point in $\mathbb{R}^3$ is identified by three values, its x, y and z coordinates, so it has a dimensionality of three.

Not only the computational complexity is used to compare the algorithms, but also their ability to correctly classify the objects in a dataset. Correctly classified objects are indicated as true positives (TP) and true negatives (TN). While incorrectly classified objects are indicated as a false positives (FP) and false negatives (FN).

The *detection rate* of an algorithm indicates how reliable it is at detecting the positives and can be calculated by $\frac{TP}{TP+FN}$. A detection rate of 1 means all positives were correctly classified as such, however this can also be achieved by classifying all objects as positive. Another indicator is necessary, the *false alarm rate*, which indicates how many objects were falsely classified as positive, and can be calculated by $\frac{FP}{FP+TN}$. A perfect algorithm has a detection rate of 1 and a false alarm rate of 0.

4.1 Distance-based outlier detection

One of the common approaches to detect outliers within a dataset is distance-based outlier detection. Distance-based outlier detection identifies outliers in a dataset by calculating their distance in relation to all other points. In this context, the distance between two objects is a value that indicates the degree of similarity between them, and is not just limited to the euclidean distance between two objects. The further an object lies from its closest neighbors, the more likely it is a possible outlier. The formal definition of an outlier varies from paper to paper, but is based on the following definition: A point is considered an outlier when there are less than p points within a certain distance D of that point. The values for p and D depend on the distribution of the dataset and amount of outliers to be expected. The distance-function that calculates the distance for any two points is also dependent on the type of data in the dataset.

4.1.1 Algorithms and applications

The first algorithm described by *Knorr et al.*[15], the nested-loop algorithm, is designed for datasets with a high number of dimensions. It is suitable for systems with a limited amount of memory, where a large dataset might not entirely fit in the memory available. In short, this algorithm divides the dataset into blocks of a certain size that fit into memory. Each object in the block is taken and its distance to every other object in that block is calculated. When an object has at least a certain number of other objects within the predetermined distance D, this object cannot be an outlier. When all objects in a block have been processed, a second block is loaded into memory and this process is repeated. After checking the second block, the process is repeated for the objects between the two blocks. This continues until all blocks have been processed. At the end, all remaining objects that were not marked as non-outlier are the outliers. On a dataset with k dimensions, this leads to a complexity of $O(kN^2)$.

The next algorithm in the same paper is a cell-based approach, which partitions the dataset into cells. It assumes that the entire dataset will fit into memory. The major difference between using cells instead of blocks, is that cells are directly related to the objects they contain. This means that a certain object will always belong to the same cell, whereas that object may have turned up in any block depending on the initial ordering of the objects. The entire feature space will be divided into equal size cells based on the initial value for D. Choosing a length of $\frac{D}{2\sqrt{2}}$ will ensure that any pair of objects in a single cell will have a distance of at most $\frac{D}{2}$.

The number of objects in a cell are counted and based on these results the cells are assigned a color, which is initially white for all cells. Whenever the number of objects in a cell exceeds a certain threshold value, the cell is colored red. Because of the number of objects contained in it, a red cell will never contain any outliers. Any white cell that is a neighbor of a red cell is colored pink. Because of the previously chosen size of the cells, the largest distance that two objects in two neighboring cells can have is D, as can be seen in Figure 4.1.

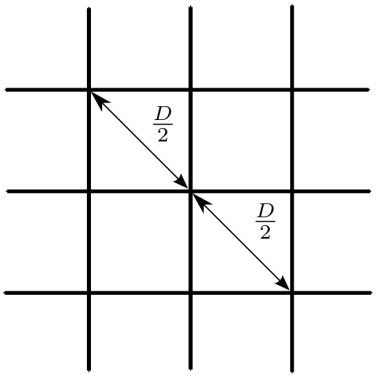

Figure 4.1: Maximum distance between points in two neighboring cells

This means that a pink cell can not contain any outliers either. All that remains is to check each non-empty white cell, where the distance is calculated of each point to each other point within the same cell and the neighborhood cells. If the number of points within that distance is under a certain number, it is considered an outlier. The paper then shows that the complexity of this algorithm is $O(c^k + N)$ for a constant c.

To show the performance and efficiency of these algorithms, *Knorr et al.*[15] perform various tests using three different datasets. The first dataset contains a tuple for every player who appeared in some NHL[1] games during a certain year. Each tuple contains various information related to that player such as number of games played, goals scored, penalty minutes, etc. Comparisons are then made of both algorithms on several subsets of different sizes and dimensions. In datasets with low dimensions, the cell-based algorithm outperforms the nested loop algorithm in every case. However, for datasets with five dimensions the nested loop algorithm significantly outperforms the cell-based algorithm. They do not perform any tests for datasets with higher dimensions.

[1] National Hockey League

The second test is done on a dataset with four dimensions, however no performance results are presented. Due to privacy agreements, they do not go into detail on the last test. None of the datasets used for testing is made available.

4.1.2 Efficient Algorithms for Mining Outliers from Large Data Sets

Based on the previous paper by *Knorr et al.*[15], *Ramaswamy et al.*[21] mention the following drawbacks of using their outlier detection algorithm:

- The initial values for both p and D is required to be user specified.
- No ranking of the outliers is provided, each outlier is regarded the same as any other outlier.
- The cell-based algorithm does not scale for higher dimensions.

They propose a new definition for an outlier and develop new algorithms to address these shortcomings. Instead of relying on a predetermined value for D, a function $D^k(p)$ is defined as the distance of the k^{th} nearest neighbor of p. An object with a high value for D^k is more likely to be an outlier than one with a low value. The top n objects with highest values for D^k are then considered outliers. Now the user only has to specify the number of outliers he is interested in.

The *nested-loop algorithm* keeps track of a list of closest neighbors for each object. Whenever an object is discovered that is closer than the k^{th} nearest neighbor so far, it is included in the list. If the size of the list for an object exceeds k, the objects furthest away are discarded. The complexity of this algorithm is similar to the previously discussed nested-loop algorithm, $O(c^k + N)$ for a constant c.

They then propose an *index-based join algorithm* that discards subsets of the dataset while computing the distances. Whenever a $D^k(p)$ has been calculated for a subset of the dataset, this value can be seen as the upper bound. This value can decrease, but never increase when examining the rest of the dataset. Using a spatial index like the *R*-tree*, entire subtrees that exceed this distance can be discarded. Also, since only n outliers are expected to be returned, any object that lies outside the currently found n objects can be discarded.

Finally they introduce a *partition-based algorithm*. The idea is to partition the dataset and based on the given n quickly determine which objects can be discarded as potential outliers, thereby decreasing the amount of calculations that will need to be done on the dataset. They start by clustering the data, where each cluster is treated as a partition. Then a lower-bound and upper-bound value for $D^k(p)$ is calculated for each cluster, which means that the D^k value for all the objects will fall in between the range of these bounds. Using the lower-bound values for each partition and the value for n, a value can be computed that will be the minimum distance required for an object to be considered an outlier. Any partition that has an upper-bound value lower than this value can now be discarded. For all objects in the partitions that are left, the $D^k(p)$ is calculated to find the outliers. No complexity is given for either the *index-based join algorithm* or the *partition-based algorithm*.

The performance of these algorithms is measured using several real and synthetic datasets. Similar to *Knorr et al.*[15] the first dataset is composed of sports statistics of players, this time for the NBA[2]. To ensure each attribute is given equal weight, their values are normalized. The resulting values are then interpreted to give a reason for being an outlier, however no performance comparisons are given on these algorithms.

A number of different sized synthetic datasets are created, each with varying size, amount of outliers and dimensions. In all cases, the partition-based algorithm outperforms the other ones by more than an order of magnitude, whereas the nested-loop is the worst algorithm. Their performance results are illustrated in Figure 4.2, for different dataset sizes in (a) and a different number of dimensions in (b). The *partition-based algorithm* scales better than the others with respect to dataset size, as can be seen in (a), and is the clear favorite of the three algorithms in terms of performance. From (b) we can conclude that this algorithm also performs well when the number dimensions are increased.

4.1.3 Distance-based Detection and Prediction of Outliers

Another approach to improve the efficiency of the distance-based outlier detection problem is taken by *Angiulli et al.*[2]. Their focus lies in two separate tasks, which is the unsupervised detection and prediction

[2]National Basketball Association

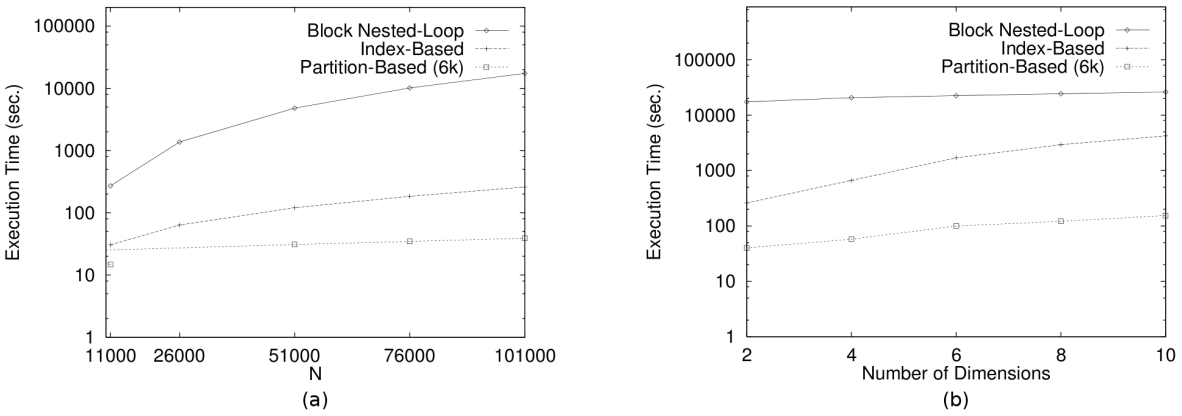

Figure 4.2: Performance results of the outlier detection algorithms by *Ramaswamy et al.*[21]

of outliers. The reason is that the previous papers are focusing on developing outlier detection methods, while ignoring the prediction part. First a subset of the dataset is constructed in such a way that the distances of the objects in this subset to any object outside this set is large enough to consider these as potential outliers. This subset is called the solving set, and can be used to determine if an object can be considered an outlier.

Figure 4.3 illustrates the results of this approach. The initial dataset of 1000 points in two-dimensional space is shown in 4.3(a). A *solving set* of 96 points is shown in 4.3(b), where the stars indicate the *solution set*. Using this solving set, 4.3(c) shows the results obtained whether an object is an outlier or not for 10000 random objects. The gray points are the objects classified as outliers. Finally in 4.3(d) the points are colored based on their weight to distance ratio based on the dataset.

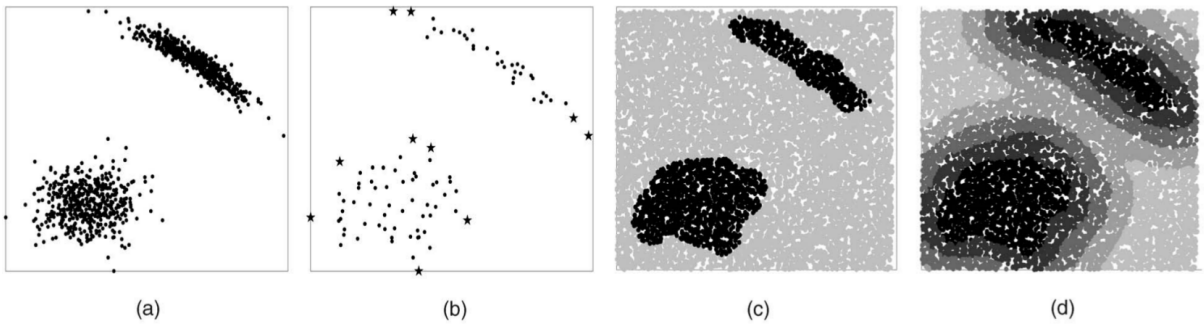

Figure 4.3: Outlier prediction using solving sets by *Angiulli et al.*[2]

Many experiments are performed using this algorithm, both on synthetic and real datasets. Not only the performance of this algorithm is tested, but also the detection accuracy by keeping track of the true and false positives and negatives. The first experiment is performed on a two-dimensional dataset to visualize the results of this algorithm. Subsequent experiments use a synthetic *Gaussian* dataset and several real high dimensional datasets, ranging from 9 up to 60 dimensions. Various sizes of these datasets are used in the experiments. Finally, they use four real datasets specifically meant for classification and intrusion detection. While this paper discusses extensively the quality of the results obtained by these experiments, the performance of the algorithms compared to the other algorithms is not reported. Apart from the obvious fact that an increased size leads to increased complexity.

4.1.4 Finding intensional Knowledge of Distance-Based Outliers

So far we have seen when something is classified as an outlier, but not necessarily why. This particular problem is addressed by *Knorr and Ng*[14]. When an object has been classified as an outlier, it is not yet known in what way it was regarded as such. They introduce the concept of a *(non-)trivial outlier*, a *strongest outlier* and a *weak outlier*. An outlier P is regarded as *non-trivial* in the attribute space A_p, if

P is not an outlier in any subspace $B \in A_p$, where a subspace is the projection of an object of a subset of its attributes. Given A_p is an attribute space containing outliers, then it is called a *strongest outlying space* no outliers exist in any subspace of A_p. All the objects in a *strongest outlying space* are called *strongest outliers*. Finally, given a non-trivial outlier P, then P is a *weak outlier* if it is not a strongest outlier.

The strongest outliers are the most useful, they indicate the minimal set of attributes needed to classify it as an outlier. This information can later be used to improve the quality of the outlier detection. In some cases outliers that occur in certain subspaces might be more rare than in other subspaces. This information is also useful in deciding what actions to take after outliers have been found, or for reporting details about the outliers to the user.

The basic steps to find the different types of outliers are presented in the naive algorithm. The naive approach works as follows: Given a set of attributes A, all the subsets of A are put in a queue. Each subset in this queue is then checked for outliers. If only strongest outliers are found, none of the supersets can contain strongest outliers so they are removed from the queue. Basically what happens is that first outliers are checked in a one-dimensional subspaces, then the two-dimensional subspaces, etc. Based on the number of dimensions in a subspace, a particular method for finding outliers is selected. More efficient algorithms than the naive are also discussed.

To test the performances of the algorithms, they use a real dataset containing player performance statistics from the NHL. It is a relatively small dataset, containing only 855 objects, but each object has many different attributes. A synthetic dataset is also created, using a data distribution similar to the NHL dataset with about 2 million objects.

The complexity of this algorithm is $O(c^k N)$ and based on these results they suggest that their algorithm in the preferred choice when working with datasets of 4 dimensions or less. It is unclear how well this translates to modern hardware, given that the paper was published thirteen years ago and they talk about memory usage in the ranges of 8MB and 16MB.

4.2 Cluster-based outlier detection

Another approach one can take to find outliers is to find clusters within the dataset. These clusters are formed by a large number of objects that are similar to each other. Then by eliminating these clusters from the dataset, only outliers are expected to be left.

A technique is described by *Yu et al.*[26] to perform this kind of outlier detection. Using a modified version of *WaveCluster* by *Sheikholeslami et al.*[23], a grid-based clustering algorithm, they find and eliminate clusters in a dataset. All the objects from the dataset that do not fall within one of the detected clusters are therefore considered outliers.

The first step is to divide the objects into a discrete, non-overlapping number of cells to create a quantized space for each attribute. The cells are all equal in width, and this is similar to the way a histogram works. For each attribute, a predefined operation will determine in which cell the object belongs.

Now that all objects belong to a cell, any number of statistically relevant properties of each cell can be used for the next step. Based on these properties, relations between the various cells can be found.

In this particular case, the number of objects that belong to a cell is used and it is called the cell density. A cell is called a *significant* cell if the weighted density exceeds a certain threshold. Two cells are considered *neighbors* if they are both *significant* cells and the distance between them is less than a certain predetermined value. Clusters are then defined as a collection of *neighbors*.

In the last step, the objects in the cells identified as clusters are removed from the original dataset resulting in a collection of outliers.

Figure 4.4 shows the results of this approach. The initial dataset is illustrated in 4.4(a), where the darkness of the pixels indicate the number of objects in that cell. Two distinct clusters found with WaveCluster are shown in 4.4(b) and the dataset with the clusters removed is shown in 4.4(c). Here the white pixels indicate empty cells and the black pixels non-empty cells, which accounts for the many black pixels in the entire image compared to the image of the initial dataset. Finally, the objects close to the clusters are also discarded and the resulting set in 4.4(d) are the outliers.

Tests are performed on synthetic datasets with two dimensions and a real-world dataset also with two dimensions. The complexity of this algorithm is $O(N)$, based on the assumption that N is bigger

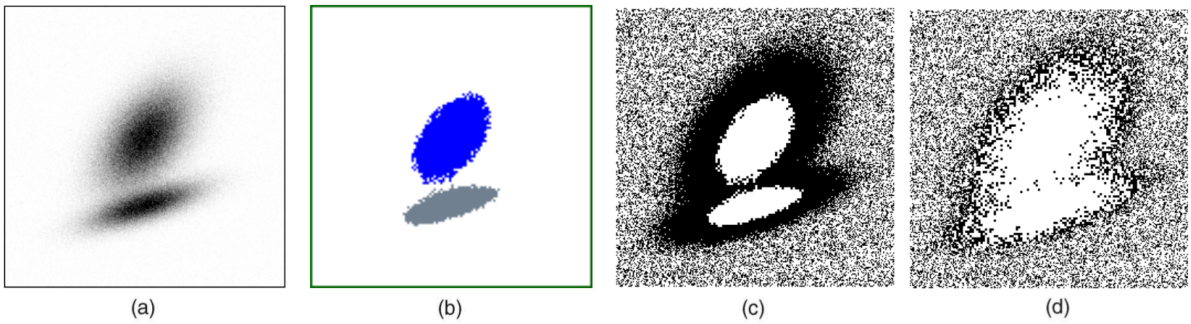

Figure 4.4: Cluster-based detection of outliers by *Yu et al.*[26]

than K, where $K = m^d$ for m number of cells and d number of dimensions. This is the case in their tests with a dataset of size $1,000,000$ and the number of dimensions equal to or less than 6.

4.2.1 Distance Distribution Clustering

The major drawback often encountered in outlier detection algorithms is that they do not scale well with regard to dimensionality. *Niu et al.*[19] address this issue using a technique called Distance Distribution Clustering. They use a combination of distance-based and cluster-based techniques to detect outliers.

The first step is to calculate the distances between all objects in the dataset. A distance sphere is created for each object, which consists of a set of rings around that object as can be seen in Figure 4.5. These rings are called spherical shells, and their size is chosen as $\frac{\sqrt{k}}{\varepsilon}$ where k indicates the number of dimensions and a predetermined value for ε.

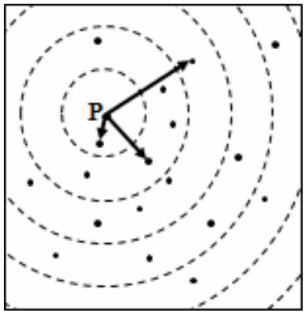

Figure 4.5: Distance sphere of an object by *Niu et al.*[19]

Given an object p, every other object in the dataset can be assigned to one of these spherical shells using the computed distance to p. This technique has some similarities to the previously mention way of dividing objects into discrete cells, except using spherical cells based on an objects location instead of cells in a grid. The number of objects in a single spherical shell is used to determine the distribution of objects around this object. The vector containing the number of objects for each spherical shell is called the transformation vector. Figure 4.6 shows the distributions for a normal object in (a) and an outlier in (b). The normal object has a large number of object in its first spherical shell, whereas the outlier has no objects within its first two spherical shells.

The next step is to perform the clustering on the transformation dataset, which is the collection of transformation vectors of all objects. By using the transformation dataset for clustering and outlier detection instead of the actual attribute values of an object, a sort of normalization is done based on the distances between the objects. This results in a more effective clustering than direct clustering, because it is done based on the relative location which reflects the isolation of a point.

And finally the outliers can be found by examining the size of each cluster. An outlier is defined as the set of objects in a cluster whose size is less than a predefined threshold value.

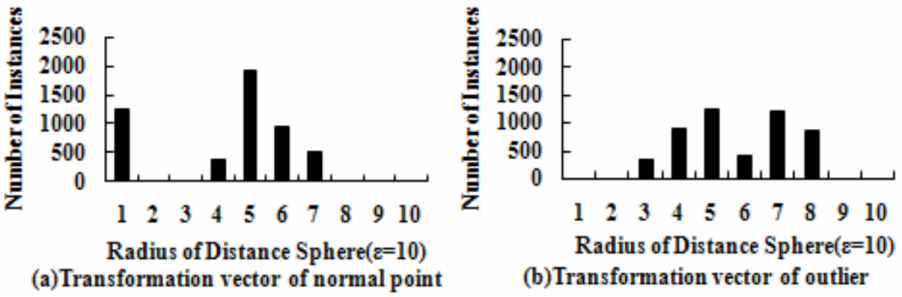

Figure 4.6: Distribution of objects within each spherical shell by *Niu et al.*[19]

Several experiments are done to compare this algorithm to an existing algorithm described in Chapter 4.5.1 to find the Local Outlier Factor (LOF). A synthetic dataset and three real datasets are used to compare the accuracy of both algorithms. The accuracy is measured by comparing the *detection rate* and the *false alarm rate* of both algorithms. A detection rate of 100% and a false alarm rate of 0.10% is achieved on a synthetic two-dimensional dataset containing 5000 objects and 53 outliers. It is an improvement over the LOF algorithm, which has a detection rate of only 92.45%. The false alarm rate however is slightly higher than the 0.08% for LOF. Three real datasets from *UCI*[10] with a significant increase in dimensionality, respectively 9, 18 and 34, were used for the next algorithms. As can be seen from the results in Table 4.1, this algorithm handles high dimensions better than LOF.

			Distribution Clustering		LOF	
Dataset	Size	Dim.	DR	FP	DR	FP
Synthetic	5000	2	100%	0.10%	92.45%	0.08%
Breast Cancer	699	9	98.76%	6.55%	62.24%	0%
Lymphography	148	18	100%	0.70%	*failure*	*failure*
Ionosphere	350	34	100%	10.71%	15.08%	6.25%

Table 4.1: Detection results of Distribution Clustering versus Local Outlier Factor (LOF) by *Niu et al.*[19]. DR stands for Detection Rate and FP is the False Positive Rate.

4.3 Link-based outlier detection

A real-world dataset not only contain continuous attributes, but also categorical attributes. Distances between two objects using categorical data may not be well defined. *Ghoting et al.*[12] present a link-based outlier detection algorithm that addresses this issue. If two objects in a dataset are similar, they can be considered linked, which in this case means they share at least one common attribute. The degree of similarity, measured by comparing all attributes between two linked objects, is called the link strength. An outlier in this dataset is then defined as a point without any links, only a few links or with weak links to other objects. To find the top outliers, a score function must be defined that assigns a score to each object based on the sum of its link strengths, where a high score corresponds to a low strength.

They test the algorithm using many different real-world datasets and compare the performance against ORCA by *Bay and Schwabacher*[3], which is a distance-based outlier detection algorithm of complexity $O(N^2)$. Of these datasets, they document the results of the KDDCup 1999 intrusion detection dataset, the US Census Bureau's Adult Incoming dataset and the Congressional Votes dataset. The focus of these tests is in determining the quality of the detected outliers, as opposed to the performance. They achieve a false positive rate of 0.35% and a detection rate surpassing the results of ORCA on most of the tests, with significantly faster execution times, for example 8 minutes versus ORCA's 212 minutes for 10% (about 2.1 million objects) of the KDDCup dataset.

4.4 Outlier detection using Self-Organizing Maps

A self-organizing map is a neural network consisting of only a single layer of nodes. It is used to map multi-dimensional data onto one- or two-dimensional space. The nodes are interconnected with their neighbors, a node connected to four neighbors will results in a square grid as depicted in Figure 4.7. Other configurations are also possible, for example nodes connected to six neighbors will result in a hexagonal grid. The resolution of the resulting map is dependent on the number of nodes in the neural layer. A higher resolution improves the results, but also decreases the performance. Once a node configuration has been chosen, it cannot change while being used.

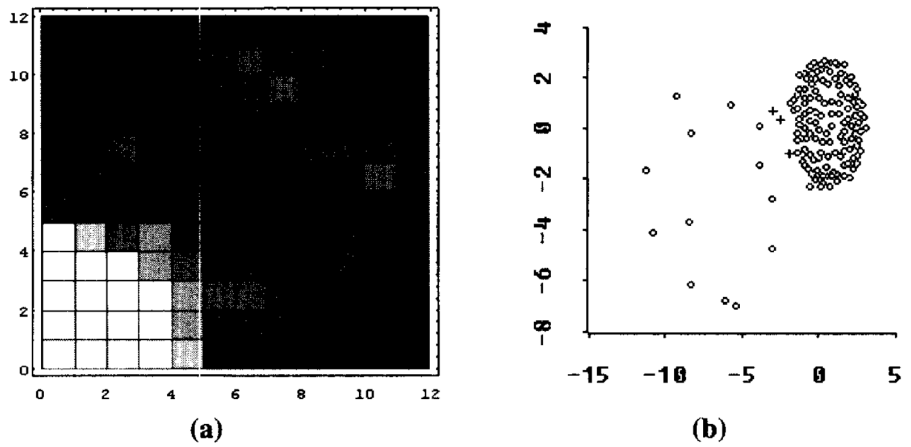

Figure 4.7: Visualization of a self-organizing map by *Muñoz and Muruzábal*[18]

Each node has a weight associated with it, which is a vector with the same dimension as the input data. These weights will be assigned random values in the initialization phase.

During the training phase, weights are adjusted based on the input training data as follows. For each object in the dataset, a winning node is selected by finding the smallest distance between the object x and the node's weight vector n_w using a distance function. All the nodes within a certain radius r of the winning node will have their weight vectors updated using the formula $n'_{w_i} = n_{w_i} + \alpha * (x_i - n_{w_i})$ where α is the learning rate.

As the training phase progresses, the radius and the learning rate decrease, which decreases the influence a single object has on the map. This is accomplished by calculating the radius as $r = r_{init} * \frac{N-i}{N}$ and the learning rate as $\alpha = 1.0 + (\alpha_{init} - 1.0) * \frac{N-i}{N}$, where r_{init} is the initial radius value, α_{init} is the initial learning rate value, N is the size of the training dataset and i is the i-th object. Figure 4.8 from *Bolzoni*[5, p.31] illustrates the training phase. As each object is processed, the weights of the connections in the map converge to a stable configuration. The self-organizing map resulting from a dataset tends to preserve the topological order and mimics the distribution of the dataset as described by *Muñoz and Muruzábal*[18].

There are two ways of finding outliers using the map. In this map, a node is an outlying node when it lies relatively far away from its neighbors. All the objects that are mapped onto an outlying node can be marked as an outlier. The second case occurs when outliers and regular data map to the same node, the outlier can then be found by making use of quantization errors (QEs). Quantization errors are the differences between the calculated value of an object and the weight of the node it is mapped to. High quantization errors occur when an outlier is projected on a non-outlying node. In both cases, *Muñoz and Muruzábal*[18] use visual inspection of the map to determine outlying nodes and high quantization errors.

Using this technique, *Rhodes et al.*[22] create multiple self-organizing maps for intrusion detection in a network. The reason for using multiple self-organizing maps is the large differences in the data from the different sources they use, such as network packet source and destination addresses, packet headers, number of packets in a certain timespan and duration of the responses and packet payloads. They investigate which of these sources can be used for a single map, and which should be split into

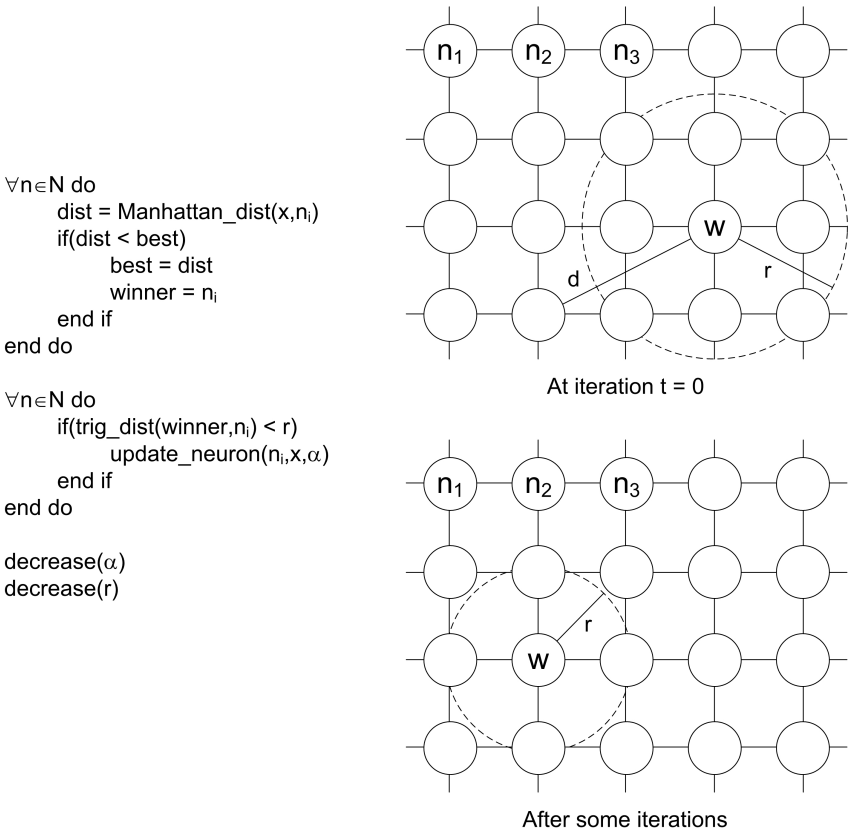

Figure 4.8: Training phase of a self-organizing map by *Bolzoni*[5]

different maps. Based on the idea that if too many sources are combined, an object might not be seen as anomalous. The function that should be used to measure the difference between two objects is the other part they investigate. Testing is done using a dataset containing DNS network packets using TCP, for which they also have two buffer overflow exploits. The dataset is small, containing only 40 packets of which 30 were used to train the system, but both exploits are detected.

In *Bivens et al.*[4], self-organizing maps are used to detect attacks in TCP dumps from the DARPA 1999 dataset. Their conclusion is that, once trained, the neural network can make decisions quickly. However the results indicate some difficulties in detecting attacks when the dataset contained many different attacks, reaching a correct attack prediction of only 24% and a false positive rate of 76%.

Zanero[27] uses an approach consisting of two stages to analyze network patterns. The first stage classifies each packet based on the payload using a 10×10 self-organizing map, with a hexagonal topology. The second stage detects anomalies in single packets and collections of packets. The same DARPA 1999 dataset as before is used to evaluate the performance of this approach. Given a detection rate of 66.7%, with a false positive rate of only 0.03%, the conclusion is that the most reliable performance is achieved when the attack rate is not too high.

Finally, the Improved Competitive Learning Network is described by *Lei and Ghorbani*[17]. It is an improved version of a simple single layer neural network, which differs from a self-organizing map. During the training phase, the neurons of this competitive learning network compete for a given object, after which the weight of the winner is updated. This means that each neuron moves toward a cluster of objects. The initial node configuration influences the resulting neural network configuration. This approach is compared to the previously described self-organizing map approach for varying node configuration. Using the KDD-99 dataset, the results of both methods were nearly identical, reaching an accuracy of 97.89% and a precision of 98.42%. However, comparing the time taken to complete these tests, the improved competitive learning network outperformed the self-organizing map.

4.5 Other ways of detecting outliers

It can be difficult to define a good distance-function for objects with a large number of attributes. Once the distances are calculated between two objects, any other hidden relations in a higher dimension they might have had, has disappeared. This can especially be a problem in datasets with a high number of dimensions, since the differences in the dimensions can average out.

4.5.1 Local Outlier Factor

Other problems that occur is that most of the previously described approaches look at the dataset as a whole, thus the detected outliers can be regarded as global outliers. In *Breunig et al.*[7] it is shown that objects that lie relatively outside of a local neighborhood should also be considered outliers, the so called *local outliers*. An example is shown in Figure 4.9, containing two clusters C_1 and C_2 and two outliers o_1 and o_2. Of these two outliers, only o_1 would be considered an outlier according to the definition for distance-based outliers as given in Chapter 4.1.1.

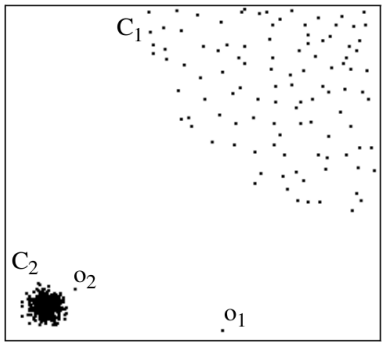

Figure 4.9: Example of a global outlier and a local outlier by *Breunig et al.*[7]

They introduce an Outlier Factor which is a value for an object that indicates the degree to which that object can be called an outlier. The actual value depends on the density of the area surrounding the object. For an object in the middle of a cluster, the LOF-value is close to 1 and it increases as the density of the surrounding area decreases. This means that the outliers in a dataset can be found by finding the objects with the highest LOF-value. To better illustrate this, the first experiment uses a synthetic two-dimensional dataset, for which the LOF-value is calculated for each object and plotted in a graph, the result of which can be seen in Figure 4.10.

The next experiments use real data from the NHL and the German national soccer league, both with three dimensions. Objects having high LOF values in the NHL dataset are almost identical to the outliers detected by the distance-based algorithms discussed earlier. Outliers found in the second real dataset are examined and can indeed be regarded as outliers. The actual complexity of the algorithm is dependent on the complexity of the chosen algorithm of a single step, but for low dimensionality it is shown to have a complexity of $O(nlogn)$ and $O(n^2)$ for high dimensionality. To test the actual performance of this algorithm, a number of experiments are conducted with dataset sizes varying from 200 to 800 and up to 20 dimensions.

4.5.2 Density distribution of projections

Another way of finding outliers in high dimensional data is described by *Aggarwal and Yu*[1]. It looks at new ways of detecting outliers in high dimensional data without the drawbacks mentioned earlier.

This paper starts by describing a method to project the dataset to a lower dimension. Each of the attributes of the objects are divided into a number of cells. However, the width of the cells are not equal like in Chapter 4.2 but adjusted in such a way as to equalize the number of objects contained in each cell.

The result is a distribution where the density of each cell is equal, but the width is variable. When performed on a uniformly distributed dataset with statistically independent attributes this should result

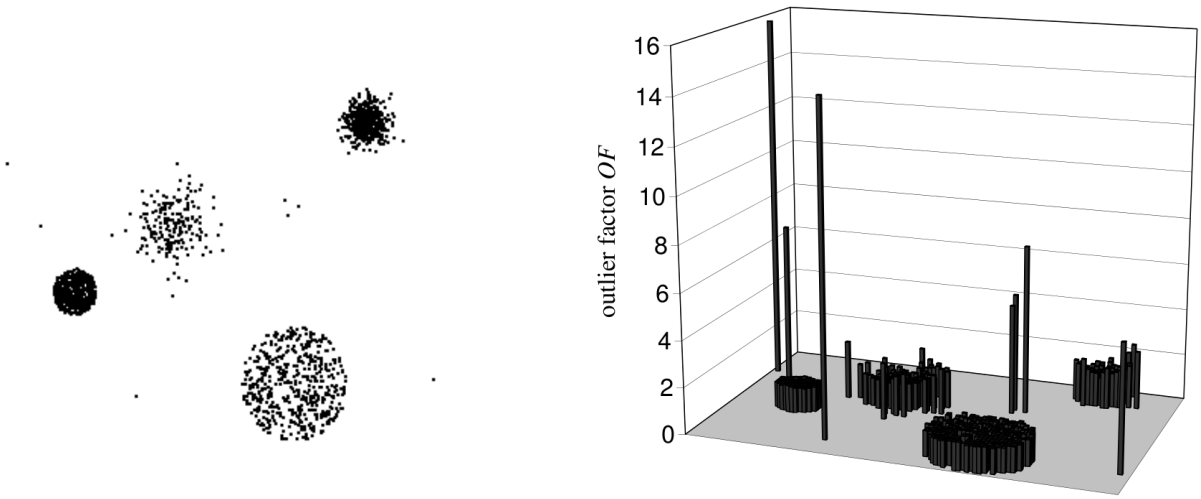

Figure 4.10: Outlier factors for a synthetic dataset by *Breunig et al.*[7]

in an equal density for each projection. The objects within the dataset that occur within a projection where this density is significantly different from the expected density are considered outliers. The projection to a lower dimension will however result in the loss of the other dimensions, but not all projections will be equally suited to find certain hidden relations or outliers. The brute-force technique tries all possible lower projections given a dimensionality and determines the sparsity of each projection.

In an effort to optimize this brute-force technique, they describe a way to solve this with a genetic algorithm. In short, a genetic algorithm determines the fitness of each member of a small population. The most fit candidates are then combined, using a crossover technique, to form new members of the next population. With a certain probability, mutations can be applied to prevent getting stuck in a local maximum. This process of selection, crossover and mutation is then repeated a number of times.

In this case, the projection to a particular lower dimension has to be encoded in such a way that it can be combined, altered and changed back. Care has to be taken during the crossover that the dimensionality is preserved. Two crossover techniques are described, an unbiased two-point crossover that simply determines a point in the encoded string and takes the left part of the first parent and the right part of the second parent. The optimized crossover technique tries all possible crossover combinations and selects the best performing.

The genetic algorithm outperforms the brute force algorithm in terms of speed with good results in terms of quality. Advantage over other outliers detection algorithms is that it handles datasets with high dimensionality better.

4.6 Summary

There are several heterogeneous approaches one can take to detect outliers in datasets. The type of data under analysis is the most important factor in choosing an appropriate algorithm. Further investigation and experimentation is needed on a dataset representative for the required use to be able to choose an effective algorithm. A necessary first step would be to test the different algorithms using identical datasets to compare their performances.

Table 4.2 provides a summary of the algorithms and their characteristics. The complexities of the algorithms, where available, use k to indicate the number of dimensions, N as the dataset size and c for an unknown constant. Note that this does not include certain algorithm-specific parameters and their impacts on result quality and execution speeds. In the case where the complexity is not explicitly reported, we included an estimate based on the available information. These estimations are indicated with a †-symbol. The dataset sizes and dimensions are the ones actually used in the performed tests, it does not necessarily indicate the maximum possible size supported by that algorithm.

Ref.	Algorithm	Complexity	Tested datasets	Dataset size	Dimensions
[15]	Nested-loop Cell-based	$O(kN^2)$ $O(c^k + N)$	1 real and multiple synthetic	855–2million	13
[21]	Block Nested-loop join Index-based join Partition-based	$O(kN^2)$ $\Omega(N)$ † $\Omega(N)$ †	1 real and multiple synthetic	335–1million	2–10
[2]	Solving set	$O(N^2)$ †	4 real and multiple synthetic	400–100,000	9–38
[26]	Grid-based	$\Omega(N)$ †	1 modified real and multiple synthetic	65,193–735,000	2–20
[19]	Distribution-clustering	$\Omega(N^2)$ †	3 real and 1 synthetic	148–5000	2–34
[12]	Link-based	$\Omega(N)$ †	3 real, modified	2.1million	14–41
[18]	Self-organizing map	$O(ckN)$	1 real and multiple synthetic	86–5000	2
[7]	Local Outlier Factor	$\Omega(N)$ †	1 synthetic and 2 real	375–800,000	2–20
[1]	Brute-force Genetic algorithm	$O(N^k)$ † $O(ckN)$ †	5 real	unknown	8–160

Table 4.2: Outlier detection algorithms overview. N denotes the dataset size, k the number of dimensions and c a constant.
† estimated complexity given the presented algorithm details

4.7 Choosing an outlier detection algorithm

An estimated complexity should be precise enough to discard the algorithms that are inefficient at first glance, such as the ones with an exponential complexity, for the remaining algorithms the complexity is not their most important property on which we base our decision for a suitable algorithm.

The data we will use can come from many different sources. These sources can be different web server implementations, but also completely different services such as load balancers, database and other kinds of storage, email and messaging and in-house developed proprietary software. All these services will output log data in various formats at various levels of verbosity, but may have a number similar types of data such as timestamps, network addresses, user names, etc. Looking at the HTTP protocol in specific, for example, one can extract many types of information, as is demonstrated in the next chapter in detail when we look at our datasets. Even though it is a stateless protocol, steps are often taken to preserve state in between requests. The preservation of the state is usually done using cookies, URI rewriting and

parameters that are hidden to the end user. By grouping individual web requests into groups or sessions, it is possible to extract additional metadata. This includes date and time of the session, number of requests, session length. Therefore the ability of an algorithm to handle a high number of dimensions is important, since we do not know the number of attributes and it may change depending on the dataset.

When operating in a continuous mode, the detection algorithms also need to be able to adapt to a stream of data instead of just a fixed dataset. Detecting outliers in a fixed dataset can be useful in situations where you are looking for attacks during a specific timespan. A more realistic use of anomaly detection is in a setup where it is in continuous operation. Algorithms that require a dataset specific configuration of parameters will be at a disadvantage for non-fixed sized datasets. Because of this requirement we discard the distance-based and clustering algorithms. Another reason to discard these algorithms is that they cannot be trained in advance. If enough outliers form a cluster, then they will not be outliers anymore by definition of those algorithms. But if an outlier detection algorithm was trained using only normal, expected data, then these anomalies would still be classified as outliers to the original trained model.

Based on these observations the outlier detection algorithm has to meet the following requirements:

- **Requirement 1**: Handle large datasets
- **Requirement 2**: Handle high dimensional datasets
- **Requirement 3**: Work in a continuous stream mode

The capabilities of each outlier detection algorithm with respect to these requirements is summarized in Table 4.3. For the first requirement we look at the complexity of the algorithm, any algorithm with a quadratic complexity or worse regarding the dataset size is indicated with a negative mark. The second requirement looks at the complexity of the algorithms and the observations and recommendations made by their respective authors to see how they perform when the dimensionality of the dataset increases. For the final requirement we give the algorithm a positive mark if it can be used on a dataset where the size is not initially known.

Algorithm	Req. 1	Req. 2	Req. 3
Nested-loop	−	−	−
Cell-based	+	+	−
Block Nested-loop join	−	+	−
Index-based join	+	+	−
Partition-based	+	+	−
Solving set	−	+	−
Grid-based	+	−	−
Distribution-clustering	−	−	−
Link-based	+	+	+/−
Self-organizing map	+	+	+
Local Outlier Factor	+	+	−
Brute-force	−	−	−
Genetic algorithm	+	+	+

Table 4.3: Outlier detection algorithms capabilities with regard to our requirements

Given these results we decide to use the Self-Organizing Maps. Using this algorithm will require some manual configuration to obtain a good detection performance. Once configured, it can be used for data that is continuously being logged as long as the initially trained models are accurate.

The contents of the detected outlying requests might not always be available. For example in situations where traffic is encrypted, or deep packet inspection is prohibited. If the contents of the traffic is available, additional knowledge may be obtained from the detected anomalies. The method described in Chapter 4.1.4 could be used to identify and classify malicious activity. Existing anomaly detection techniques for web-based traffic, relying in on the actual payload of each request, could also be used.

Usage of the detected anomalies is not limited to intrusion detection, but can also lead to possible bottlenecks or problems on a larger scale between services within the network. Depending on the actual information available the collected data, the contents could also be inspected to detect specifics with regard to why these objects are outliers. Finally, all the aggregated data can be reported to the system administrator, who will then only have to look through the interesting bits of information.

Chapter 5

Datasets

To test and evaluate the detection of the various cases, we have collected request logs from two distinct and unrelated web applications. The first one is a large Java-based web application, only accessible to authenticated users of a single company, containing highly privacy sensitive information. The second one is a log collected from a game, created exclusively for an online contest, which can be played in the webbrowser. At the end of the contest, the top scoring players will each be awarded with a prize. Both applications present an interesting target for attackers, the first for its information and the second for its prizes.

5.1 Dataset 1

The infrastructural setup of this application, as illustrated in Figure 5.1, consists of two loadbalancers and an application server cluster for a single deployment. In this particular case there is only a single cluster, but in the future this could be increased to multiple clusters. All requests coming in from the internet are handled by the loadbalancers. Based on the domain in the request, the loadbalancers route the request to the appropriate application servers. Each application server is capable of hosting multiple deployments, but the deployments are all isolated from each other and every deployment has its own logfile. Dataset 1 contains all the information logged by the application server and the loadbalancer for a single deployment.

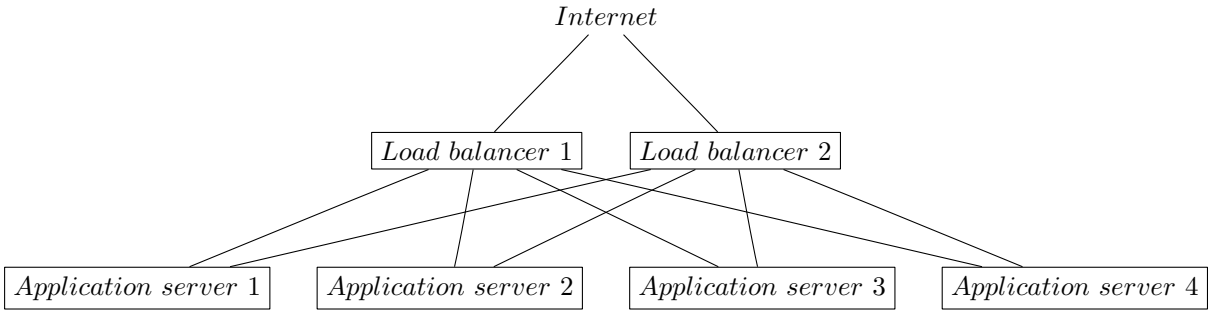

Figure 5.1: Hardware infrastructure

The application has a number of different services it provides such as CRM-like functionalities with extensive searching capabilities, time registration, financial reporting, importing and exporting of data, etc. There are also a number of distinct user roles each with specific authorizations for the various parts and the users will have regularly reoccurring workflows. This will allow us to create a baseline of expected user behavior for each user. The number of unique users is close to 600, which is not expected to change much during the collection period, given that only employees of a single company are given access.

Each loadbalancer keeps a log of the requests it handled in its own logfile. A single request is represented as a single line in this logfile in a customized Apache format, which looks like the following:

```
hostname.example.com 443 127.0.0.1 - - [26/Jun/2011:08:23:06 +0200] "GET /url/page HTTP/1.1"↩
    200 3593 "https://hostname.example.com/url/page" "User agent string" "-" 0 20623
```

This single line is treated as a single object in the dataset and contains the following information:

host
The domain or hostname of the server.

port
Portnumber on which the server listens for client requests.

IP address
IP address of the client performing the request.

timestamp
Date and time of the request, in the timezone of the server.

http method
String indicating the request method, of which the most used methods are GET and POST.

path
Path of the request as requested by the client.

http response
HTTP response code sent to the client.

response size
Size of the complete response in bytes.

referrer
URL of the referrer as given by the client.

user agent
User agent string of the client application performing the request.

Similar to the loadbalancer, each application server also keeps track of the requests it handles in logfiles. Each request is logged as a single line, but the format and contents are different:

```
08:23:06,328 INFO  [RequestLogFilter] user=admin /url/page viewId=/path/file time=219
```

A new file is created every day at midnight and the filename contains the date. This line contains the following information:

timestamp
Date an time of the request, in the timezone of the server.

IP address
IP address of the loadbalancer responsible for routing this request to the webserver.

host
The domain or hostname of the server.

username
Name of the user, if logged in, empty otherwise.

time
Time in milliseconds needed to complete the request.

Source	Collection period	Number of days	Number of requests
Load balancer 1	26-06-2011 - 30-09-2011	97 days	508 743
Load balancer 2	26-06-2011 - 02-10-2011	99 days	3 679 950
Application request log	17-06-2011 - 31-08-2011	75 days	954 090

Table 5.1: Overview of collection periods for the first dataset

path
The path as requested by the client.

server path
The actual path of the file that is mapped to the request path.

parameters (optional)
The request POST parameters, if available. These are filtered or omitted when they contain privacy sensitive information.

The application makes use of URL rewriting techniques, which explains the presence of the *server path* attribute. The *parameters* attribute is only available for POST requests, and may not contain all the original parameters since privacy sensitive information is filtered out prior to logging the request.

The application servers also keep track of an application debug log. This logfile is used by the various frameworks and developers to log information during a request. The result is a large textfile, where a single request can generate multiple lines of debugging information. It is not directly a usable source for outlier detection, but it can be a valuable source of information for reporting purposes once the outliers are identified.

The various logfiles are collected from a number of different physical machines, in different geographical locations. Due to configuration differences, network latency and processing times the actual timestamp in the various logfiles can vary by small amounts. This can make it difficult to directly link related requests from different sources by exact timestamp.

The presence of a firewall and the requirement for each user to be logged in before he can use the application somewhat limits the amount of malicious activity we expect to see. Given the demographics of our target users we can assume that the majority of our users also lack the knowledge to intentionally hack into or break the application. Our focus will therefore be on detecting strange application behavior, such as for example unusually long sessions, response times or unexpected navigation behavior. Outlier detection in this case will help detect or diagnose problems in the application, which in turn could lead to finding previously unknown bugs or potential performance problems.

The data was collected over a period of several months, from June 2011 until October 2011 and consist of over a gigabyte of text data in mulitple files and formats. Table 5.1 contains the collection period and sizes of the logs to give an indication of the amount of data we are working with. Before the data can be used however some requests must be filtered out. These requests are not relevant as they are the result of our internal monitoring tools and the exchange of data between application servers. We use the requests that fall within the period from 26-06-2011 until 31-08-2011, since this period is covered by all sources. This will account for the small decrease in size of the actual datasets being used in the next chapters.

Figure 5.2 illustrates the amount of requests handled per day. A weekly pattern is visible, the number of requests drops during the weekends as mentioned earlier.

Before we can use this dataset we have to make sure that it contains only the attacks we actually want to detect. The first step is to manually inspect the request log to make sure that it does not contain any malicious activity. Then we perform attacks on the application server, making sure that our attacks match the described threats in the threat model. All the attacks have taken place in the last week of the collection period. This was done so that enough data would be collected to construct reliable models for normal user activity. It was not possible to perform the attacks on the actual production deployment of this application. The reason was to preserve the data integrity and the privacy sensitive nature of the data. To perform the actual attacks, an application instance running locally was used containing an anonymized copy of the database. The resulting request log was merged with the actual request log

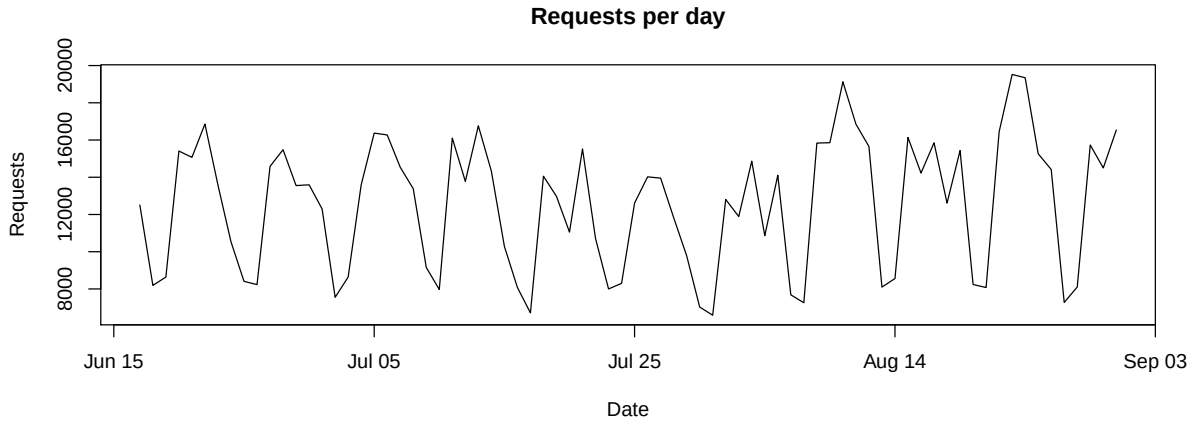

Figure 5.2: Requests per day

before any analysis was done and we have a dataset containing attacks as described by our threat model, suitable for use in our experiments.

The attacks have been performed both manually and automatically using scripts. User accounts to perform these attacks from where chosen based on the amount of activity logged in the period. To simulate the first two threats described in our threat model the following attacks have been performed:

- Manually requesting all the client detail pages from the search results of a single query.
- Manually modifying or removing pieces of information from the clients requested in the previous attack.
- Automatically scrape all client detail pages from the search result of a single query.
- Automatically scrape a number of client detail pages using modified URL's where unique id's are incremented.
- Automatically posting multiple reports for each client requested in the previous attack.

5.2 Dataset 2

The only source of information for the second dataset comes from a small Flash-based game on a Facebook fanpage of a large business, we are unable to disclose any additional details due to a non-disclosure agreement. This game was accessible during a limited time in September through October 2011. It has a competition element where users are awarded a prize based on their performance in the game after the competition period has ended. The game itself is a game based on memory and logic, similar to Mastermind. A single game from start to finish is fairly short, typically taking just a couple of minutes, with a single round in the game taking up to 30 seconds. The score is based on the amount of time it takes a user to complete the given puzzle.

Before a user is allowed to play the game, he must be logged in on Facebook and he has to click the *Like* button on the company fanpage. Given the involvement of prizes for high-scores, some users may attempt to cheat the system in various ways. There are ways to cheat the system, from tampering with the data containing the score information being sent back to the server to using tools that put a high load on the CPU in an attempt to slow down the game.

The game logs only the following pieces of information:

user id
: Numerical value uniquely identifying this user.

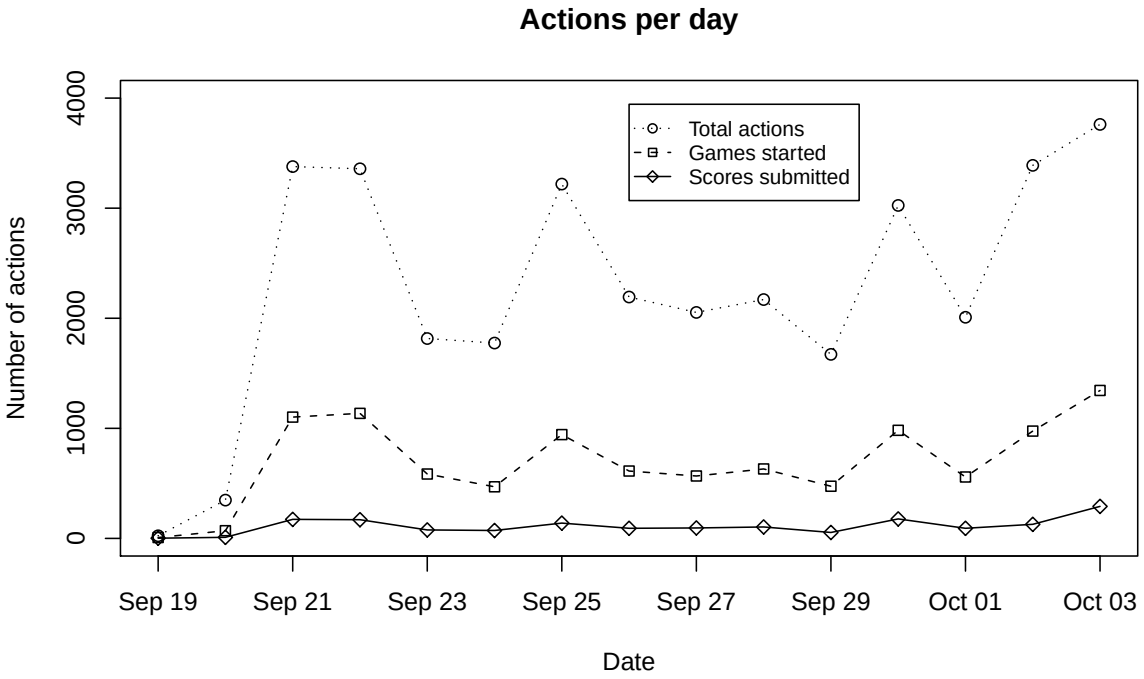

Figure 5.3: Actions per day

IP address
IP address of the client playing the game

timestamp
Date and time, in the timezone of the server.

action
Text describing the user action that caused this request, such as starting a new game or submitting a score.

The *action* attribute describes what the user actually did, this includes logging in, loading the page, playing the game and submitting a score. Many of these actions depend on a previous action being available. For example, a user should only be able to send a score if he started a game. A successfully played game therefore consists of a number of different entries in the logfile.

The data was collected over a period of 14 days, starting from September 19 2011 until October 3 2011. Figure 5.3 shows the number of actions logged and number of games played on each day.

Table 5.2 contains detailed information about the contents of log. The difference between the number of games started and the number of submitted scores can be attributed to the fact that these players did not finish their game after they started it. The large difference between unique IP addresses and unique users can be explained by users attempting to play the game that were not logged in and therefore their name was unavailable. This can be seen in the log as a number of different IP addresses that account for only a single entry in the entire dataset. Given the lack of additional information, these entries do not contribute in a meaningful way so we filter them out.

This dataset is used to detect threats that try to manipulate the scores. The outliers we are looking for in this dataset are the set of objects that together might indicate some kind of tampering or cheating having taken place. This can be scores that significantly differ from the other scores, games that have taken an unusually long time to complete. High scores from players who have played multiple game could be more reliable. Other suspicious activity that are likely the cause of player tampering are game

Total size	34 190
Unique IP addresses	5 039
Unique user id's	727
Games started	9 308
Scores submitted	1 676
Minimum score	5 638
Maximum score	30 000

Table 5.2: Overview of the game dataset

start actions and score submissions from players that did not complete the initial requirements of logging in and liking the page.

Chapter 6

Approach

Assuming an attacker has gained access using valid credentials of an existing user, the system will treat him as a valid user. However, it is most likely that the usage patterns of this intruder will differ from the ones of the actual user whose account he has hijacked. Given the limited amount of information present in a single object in the application log dataset, we must transform the original data before it can be used by an outlier detection algorithm.

Because HTTP is a stateless protocol, the application has to keep track of its sessions using cookies if it want to preserve the state across multiple requests, meaning that the majority of the requests are part of a session. Every session starts when the user logs in and ends when the user logs out. In most cases a user simply closes the webbrowser instead of logging out, so to prevent a session from lasting indefinitely, it is also ended after a certain time has passed in which there was no activity, this is called the session timeout. The value of the session timeout depends on the configuration of the application server and is typically specified in number of minutes. This approach can also be taken for other datasources that log information in a similar way and where each individual object has an attribute that can be used to group together similar entries.

6.1 Recreating sessions

We assume a request log is sorted by request time. A hashmap using the username as key and a session is used to keep track of all active sessions. We iterate over each request object in the dataset and find the currently active session for the user responsible for that request. If a session exists for that user we compare the last access time of that session to the access time of this request. If these access times are within the SESSION_TIMEOUT constant minutes of each other then this request belongs to the current session. Otherwise we create a new session for that user. The following pseudo-code describes the necessary steps to reconstruct the user sessions given a request log.

Algorithm 1 Group requests into sessions

```
for each r ← requests do
    s ← sessions[r.username]
    if Exists(s) ∧ (r.AccessTime − s.LastAccessTime) < SESSION_TIMEOUT then
        s.AddRequest(r)
        s.LastAccessTime ← r.AccessTime
    else
        s ← NewSession()
        s.AddRequest(r)
        s.LastAccessTime ← r.AccessTime
        sessions[r.username] ← s
    end if
end for
```

6.2 Normalizing the data

Once we have recreated the sessions, we can aggregate some of the attributes it contains and use the results to create a new dataset. The actual attributes that we can use for this vary depending on the information contained in the dataset. The goal is to extract meaningful data from a collection of objects that contain little information individually. Meaningful data in this context can be defined as numerical data, which can be compared to each other so that any outliers may be found.

The final step we take is normalizing the dataset so that all numerical values will be in the range of $[-1.0, 1.0]$. To achieve this we find the minimum (MIN) and maximum (MAX) values for each attribute. Then we iterate over each value and replace it with the result of following formula:

$$1 - 2 * \frac{value - MIN}{MAX - MIN}$$

The contents of this normalized dataset can be used by the outlier detection algorithm. We will use a Self-Organizing Map to train a model based on the usage of the system from the actual users. The trained model will be used to classify the objects and find any outliers.

Using this basic approach Chapter 7 will explore various ways of creating this new dataset suitable for outlier detection and then use it to train the Self-Organizing Map and to detect the outliers. In addition, the configuration of the Self-Organizing Map will be explained and the effect various configuration parameters have on the detection results. Apart from the actual datasets, all information necessary to recreate the Self-Organizing Map and reproduce the results will be present.

Chapter 7

Experiments

In the following experiments we will use a Self-Organizing Map to detect outliers in a number of datasets. The same approach is taken as described in Chapter 4.4. We use a rectangular network layout, which means that each node has four neighbors (north, east, south and west) unless it is a node on the edge of the network. The size of the networks are expressed as $a \times b$, where a signifies the width and b the height of the network. During the training phase, we iterate over the training set until a desired number of iterations has been reached, this amount is called the number of *epochs*.

Each of these parameters is specific to the situation in which you use the Self-Organizing Map. In *Bolzoni*[5, p.35] on the subject of parameter tuning, it is said that "a small network yields a too course classification, while a large network yields a too sparse one". We use the values presented in [5] as a starting point for our own experiments. To determine the impact each of these parameters has on the outlier classification, we perform the training phase and detection phase using a number of different values.

All the experiments are performed using our own software implementation. Appendix A contains a description of all the software utilities required to perform the experiments.

7.1 Detecting unauthorized data access

In this first case we will use the application request logs from dataset 1. The application servers are configured to use a timeout value of 30 minutes, so we will use this value for the SESSION TIMEOUT when recreating the sessions. It is important to use the same value that is used by the server for recreating our sessions. If we were to use a higher timeout value, our sessions would last longer than they actually did on the server, resulting in a lower number of total sessions created from the dataset. Conversely, using a lower timeout value would result in a higher number of recreated sessions since they would last shorter than they would have on the server. In both these cases the requests contained in each session would not necessarily reflect the actual collection of requests belonging to that original session.

Grouping the requests into sessions for our initial dataset of 954090 requests using a timeout of 30 minutes results in a new *sessions dataset* containing 22747 objects. A unique identifier is also added, it is a unique, sequential number which can later be used to identify any sessions that have been classified as outliers. We end up with a new dataset with the following attributes:

- Unique identifier
- User name
- Session start time
- Session end time
- Session duration in seconds
- Total number of requests
- Number of unique pages requested

- Number of requests per minute

7.1.1 Detecting attacks

In these situations with limited information we are also limited in the type of attacks we can detect. Based on the data available in this first dataset, we focus on detecting anomalous behavior related to information access. After inspection of the dataset we did not find any attacks as mentioned in Chapter 5, so we designed and performed some attacks on the application ourself. The following attacks were designed to violate the information confidentiality and each has unique characteristics. For each attack we also describe the *expected* characteristics of the session containing the attack compared to a regular session. The resulting attack sessions and their characteristics as they appear in the dataset are listed in Table 7.1, along with some regular sessions from different users and their characteristics as a comparison.

Attack 1.1: Accessing client information
The attacker uses the standard search functionality of the application to access information for a large number of distinct clients. This session will be of average length, have an average number of requests to a small amount of distinct pages. Each request for client information will be represented by the same page in the logfile, regardless of the client identifier, which is the reason for the small amount of distinct pages.

Attack 1.2: Information request on a specific target
This attack only performs simple search and retrieval of information on a specific client in the system. This session will be short and contain only a few requests.

Attack 1.3: Automated web crawler
The attacker retrieves the contents of every page that is linked to on the current page, and continues to do so. This session will be over a longer period and contains a large number of requests to a large number of distinct pages.

Attack 1.4: Sequential page crawler
Using additional knowledge of the system and an automated tool a large number of requests are performed on a single page with an ever increasing identifier. It is common for web application to use a database to store all the data related to that application. Every object stored in the database must be uniquely identifiable by a piece of data, often called the primary key. An often used method for generating these unique keys is by using a numerical value that is incremented by one for each object. Given an existing object with an identifier n, we can try to request the object with the identifier $n + 1$. As long as the server replies with an existing object, we can continue incrementing this identifier. This session will be of average length and contains a large number of requests to a small number of distinct pages.

id	start	length	RC	DPC	req/min	type
21337	2011-08-27 10:40:13	2250	69	6	1.64	Normal session
21340	2011-08-27 10:50:18	211	16	5	5.33	Normal session
21352	2011-08-27 11:33:31	5064	72	6	0.86	Normal session
21354	2011-08-27 11:40:42	714	30	4	2.73	Normal session
21360	2011-08-27 12:35:35	905	42	4	2.78	Attack 1.1
21709	2011-08-28 22:05:10	78	7	2	5.38	Attack 1.2
21786	2011-08-29 09:25:49	3526	35877	59	610.50	Attack 1.3
22261	2011-08-30 15:27:42	507	6982	5	826.77	Attack 1.4

Table 7.1: Sessions containing an attack in the application request logs of dataset 1. The session length is measured in seconds. RC stands for Request Count: the number of requests contained in the session. DPC is the Distinct Page Count: the number of distinct pages accessed in the session.

7.1.2 Setting up the Self-Organizing Map

First we look at different network sizes to see what impact they have on the quantization errors of the objects in the dataset and classification accuracy. The quantization error, as explained in Chapter 4.4, is the difference between an object and its closest matching node. Once the quantization error crosses a certain threshold, the difference is significant enough that we can call this object an outlier. This quantization error is based on the differences between the object and a node in the Self-Organizing Map and the threshold for an outlier should be determined on a case-by-case basis. A lower threshold increases the number of outliers we detect, however the downside is that we also increase the possibility of incorrectly classifying an outlier as non-outlier, resulting in false positives.

Size	Quantization error	Outliers	FP	FN	run time
12×8	0.50	5	3	2	1379 ms
16×8	0.50	4	4	2	1847 ms
16×16	0.50	7	7	2	3590 ms
64×16	0.50	0	0	4	13961 ms
32×32	0.55	2	0	2	14017 ms
64×64	0.50	5	3	2	38330 ms

Table 7.2: Results for differently sized trained networks. Outliers indicates the number of outliers detected. FP and FN show the number of false positives and false negatives respectively. The learning rate is 0.1, training size is 5000.

For now we use a *learning rate* of 0.1, a training phase of 10000 epochs and a training dataset consisting of the first 5000 objects of our dataset, which is about 22 percent of the total size of our dataset. The learning rate directly influences the rate at which an individual object changes the network during training. When working with larger training sets, an individual object should only have a minor impact on the network, so we use a small learning rate. The learning rate decreases as the training progresses and as we will see in the other experiments we need to start with a larger learning rate when the size of the training set is smaller. The reason for which is that with a smaller training dataset you want the network to reach a stable configuration sooner, the consequence is that the final network might be less accurate than the one trained with more objects over a longer period.

7.1.3 Training the Self-Organizing Map

We have taken the training data from the beginning of the dataset, since the actual attacks were performed near the end of the collection period. The actual size of the training dataset is determined based on the size of the complete dataset and the kind of data. In our case the user activity varied from day to day, mostly based on individual user roles and the days and hours each user had to work. The actual user activity followed a weekly pattern, as we have seen in Figure 5.2 in Chapter 5. Based on this observation we decided to use a period that is a multiple of a full week for objects in our training dataset. Three weeks contains enough objects to have a diverse enough dataset to train our model with, which corresponds to about the first 5000 objects.

Next we have to define the attributes that are used in the training and classification phases. These attributes will be the actual pieces of information which will decide whether a given object is an outlier or not. In this case we want to detect sessions with a distinctly different session length, number of requests, number of unique pages accessed or the number requests per minute. The session length is used in combination with the other attributes, although the number itself can also signal strange behavior if it is significantly longer than the average. Next is the number of requests, this is important to determine how much activity was generated during the session. The number of unique pages tells us more about what the user did during the session, a low number means the session was concentrated on a small portion of the application. The number of requests per minute is derived from the session length and number of request attributes, this tells us how active the user was during the entire session. These four attributes correspond to the 4 dimensions.

Table 7.2 shows the results that can be obtained for differently sized trained networks with these parameters. As we can see from this table, an increased size leads to better detection results up to

a certain point, where a size of 32×32 gives the best results. An object was classified as an outlier if its quantization error exceeded the threshold of 0.5. The size of the training set was large enough for the network to reach a stable configuration even for different sizes. Because of this we can use the same quantization error threshold even when using different network sizes. When working with smaller datasets, choosing a quantization error threshold for differently sized networks has a bigger impact.

Choosing a good quantization error threshold must be done manually, and there is a trade-off between a low false positive rate, an accurate true positive rate and a low false negative rate. Figure 7.1 shows the quantization errors for all the sessions in this dataset to give an idea of the range of the quantization errors and how the quantization error threshold relates to the detection of correct outliers. The actual values and distribution of these quantization errors can vary slightly when training and running the Self-Organizing Map multiple times, since it depends on the random initialization of the network.

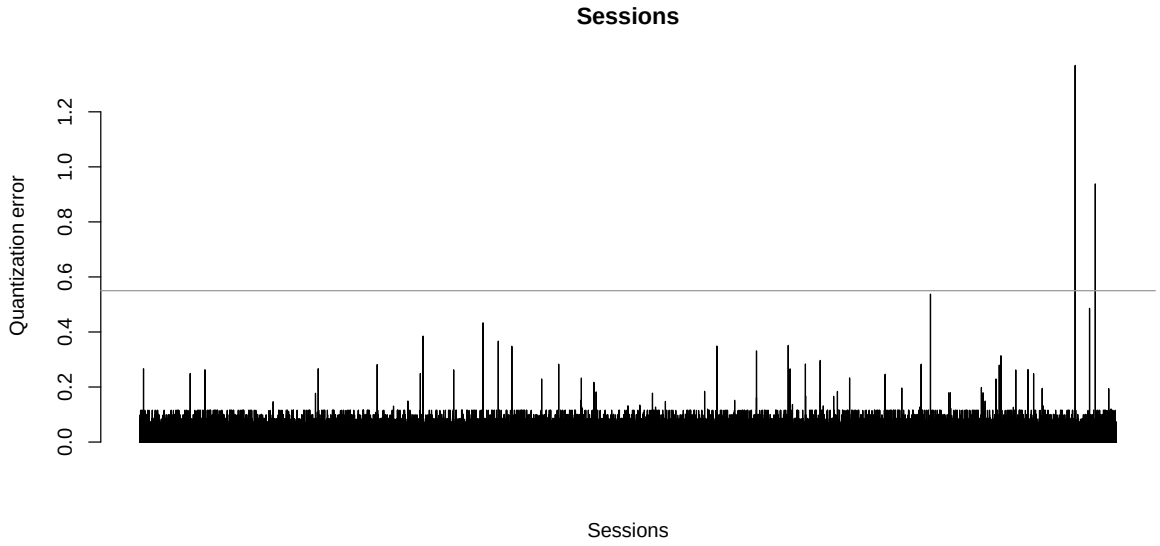

Figure 7.1: Quantization errors for all sessions, with a threshold of 0.55.

The time it took to train and detect the outliers is also included in this table as a way to illustrate the performance impact the different sizes have, while they may vary depending on the hardware used it still shows that an increase in network size negatively affects the run time significantly. Given these results we will now continue to use 32×32 sized networks.

7.1.4 Results

Having built a network of the sessions with the correct parameters, we can now perform the outlier detection. Running the outlier detection algorithm on this dataset results in three outliers, as can be seen in Table 7.3. Two of these outliers are caused by attacks 1.3 and 1.4, the objects with id's 21786 and 22261 respectively. They were seen as outliers based on their requests per minute, which was significantly higher than any user session in the dataset. The attacks in these sessions are done by an automated script, explaining the high number of requests per minute.

Attacks 1.1 and 1.2 were performed manually, over a relatively normal period of time. The characteristics from these two attacks were not significantly different from other user behavior, which explains why they were not classified as outliers, as can be seen in 7.1. The approach taken here seems to work for automated attacks where the speed and efficiency of a computer outperforms user actions by several orders of magnitude.

The final result does not contain any outliers when the quantization error threshold is chosen correctly. Looking at the quantization errors of the session in Figure 7.1 there are two session near the end that have the highest quantization errors without being classified as an outlier. These are unfortunately

not the other attacks, but they are valid sessions. The reason for their high quantization errors is the exceptionally high session lengths, lasting several hours in both cases. Manual inspection of the data confirms that this is a valid session, consisting of a lot of manual data input by a single user. These sessions are very rare and only occurred twice in a period of several months, they were responsible for the false positives we encountered with the other network sizes.

id	start	length	RC	DPC	req/min	outlier type
21786	2011-08-29 09:25:49	3526	35877	59	610.50	TP: Attack 1.3
22261	2011-08-30 15:27:42	507	6982	5	826.77	TP: Attack 1.4

Table 7.3: Outliers in sessions from dataset 1. TP indicates a True Positive outlier.

After tweaking the outlier algorithm parameters, these results were the most promising. Other attempts only increased the number of false positives detected and none of them was able to correctly detect the sessions of the two manually performed attacks. It is unlikely that the current approach based on user activity contains enough information to separate the regular session from these attacks. Lowering the quantization error threshold in this case is not a possibility, as it only leads to an increase in the number of false positives detected without actually detecting additional attacks. It is not even certain that manual inspection of the requests will be able to identify the first two attacks.

7.2 Per-user model training

The first attempt uses a Self-Organizing Map to build a single model for all user sessions. This assumes however that all users behave in somewhat the same way, at least enough to use a single model. Large web applications usually have many different user roles, each responsible and authorized for certain tasks. To determine whether this has a significant impact on the outlier detection we take a different approach by building a model for each user.

The initial sessions dataset is created in the same way, but then we split this dataset by username, resulting in a collection of datasets, one for each distinct user. Our application requestlog dataset contains 583 unique users, each of which now has its own requests dataset. Some of these datasets will be unusable, either because they are too small or because (in our case) the username is unknown.

7.2.1 Detecting outliers

We pick two users at random from this collection of user-session datasets to perform the next experiments. The dataset of the first user contains 39 sessions, of which we use the first half as training data. This dataset contains the session that performs attack 1.1. For the second user we have 222 sessions, of which again we use half as training data. This dataset contains the session that performs attack 1.3.

These datasets are smaller than the one used in our first experiments. This impacts the choice of the various parameters used for the Self-Organizing Map. Most importantly the learning rate, this should be increased as the network has less data to work with to stabilise itself. Table 7.4 shows the outcome of the experiments done on these datasets with different network sizes.

Because of the smaller datasets, the quantization errors were more varied and the actual outliers were less distinctive from the rest of the dataset than the ones in the first experiment. This makes choosing a good quantization error threshold more difficult. In all these cases the actual threshold value was chosen to try and get the actual attacks while minimizing the false positive rate. The smaller network sizes for user 2 however were able to detect all the attacks, while having a false positive rate of 0%.

Given these results we will now continue to use the 16×16 size for the first user and a size of 6×6 for the second user. Figures 7.2 and 7.3 show the quantization errors for the entire dataset of both users.

7.2.2 Results

The following results were the best obtainable with these datasets after tweaking the algorithm parameters. These results are similar to the ones from out first experiment that used a single model. The outlier that is detected in the first dataset is shown in Table 7.5 and is a false positive. Given the length

User	Size	Quantization error threshold	Outliers	FP	FN	run time
User 1	6×6	0.65	2	2	1	7 ms
	8×8	0.55	1	1	1	10 ms
	12×12	0.55	3	3	1	16 ms
	16×16	0.55	1	1	1	22 ms
User 2	6×6	0.50	2	0	0	10 ms
	8×8	0.55	2	0	0	13 ms
	12×12	0.30	3	1	0	19 ms
	16×16	0.20	2	1	1	17 ms

Table 7.4: Results for differently sized trained networks. Outliers indicates the number of outliers detected. FP and FN show the number of false positives and false negatives respectively. The learning rate is 0.7, training size is 100.

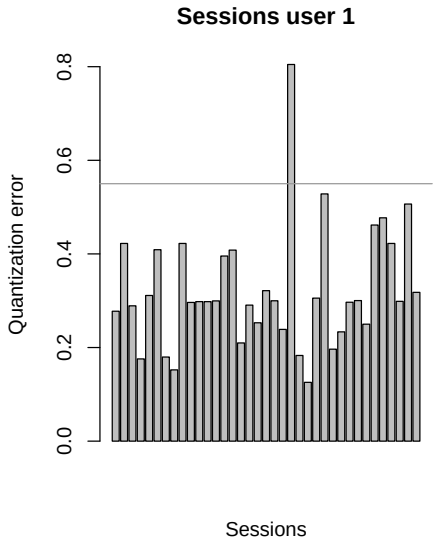

Figure 7.2: Quantization errors for the user 1 session dataset, with a threshold of 0.55.

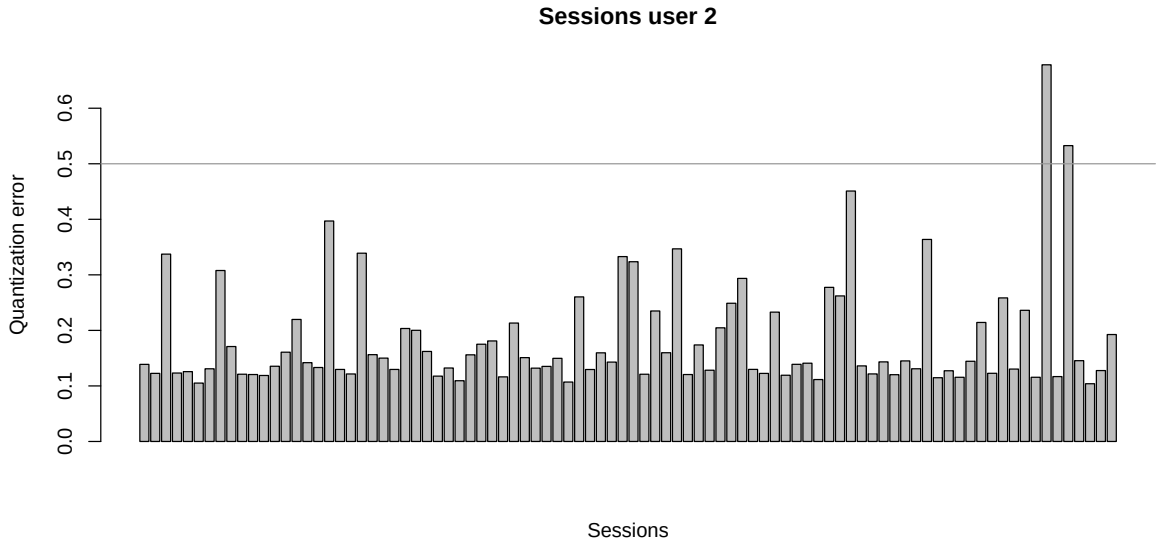

Figure 7.3: Quantization errors for the user 2 session dataset, with a threshold of 0.5.

of the session and the low number of requests and unique pages made this an outlier. The attributes of the actual attack in this dataset were not distinctive enough to be seen as an outlier.

id	start	length	RC	DPC	req/min	outlier type
21715	2011-08-28 22:15:29	2115	39	5	1.11	FP

Table 7.5: Detected outliers from the dataset containing the sessions of user 1.

Table 7.6 shows the outliers for the second user. These results were better than those of the first user, the fact that these sessions were created by automated attack resulted in higher and more distinctive values for their respective attributes. A larger size of the dataset and therefore also the training dataset also attributed to a better distribution of the quantization errors, making it easier to choose a good quantization error threshold. This can also be seen in Figures 7.2 and 7.3 where the majority of the sessions for user 2 have a lower quantization error.

id	start	length	RC	DPC	req/min	outlier type
21786	2011-08-29 09:25:49	3526	35877	59	610.50	Attack 1.3
22123	2011-08-30 08:36:13	13220	1062	46	4.83	Attack 1.3

Table 7.6: Detected outliers from the dataset containing the sessions of user 2.

7.3 Detecting vandalism

Until now we have only looked at behavior that is specific to the retrieval of web pages, the reason being that that part of the dataset did not differentiate between retrieving and creating or modifying information. The first dataset also contains the loadbalancer requestlog, which contains the specific HTTP method that the request performed. In this case we are interested in the requests that create or modify data. A web application that correctly follows the HTTP specification ensures that all GET requests are idempotent, as defined in Section 9 of the RFC 2616[9], meaning that any number of identical GET requests should always result in the same response without modifying any information. We assume

that only POST requests are capable of modifying data, which means we can ignore GET requests for these specific attacks dealing with data modification.

Attack 2.1: Modifying multiple pages
The attacker modifies the contents of a number of different pages in a single session. This attack is similar to attack 1.1, except that in this case we also modify the pages we access. The session will be characterized by a large number of POST requests to different pages.

Attack 2.2: Scraping multiple pages
The attacker performs an automated attack on a URL to download the contents, incrementing the resource id for each request. This attack is similar to attack 1.4 but not generating as many requests per minute. The session will be characterized by a larger number of HTTP errors as the user will not have access to all the requested resource and some resource do not even exist, resulting in 404 errors.

Attack 2.3: Posting a large number spam messages to a single page
The attacker creates a lot of data on a single page. This will also result in a large number of POST requests in a single session, but this time they are all targeted at a single page.

The loadbalancer requestlog dataset consists of single HTTP requests and like the application requestlog dataset they are part of a session. We do not have access to specific user names, only the client IP addresses. There can be difficulties when the IP address is the only piece with which to identify a client when a single public IP address is being shared by multiple clients through Network Address Translation (NAT). This is noticeable in our current dataset by the fact that the number of unique clients (IP addresses) is lower than the number of unique users we see in our application log.

Some differences compared to the previous dataset is the way the request path is recorded. The actual request URL as performed by the client is stored in the requestlog, instead of a more general file path. This means that a request to a single page for different entities of an object, identified by different id's, will be seen as two distinct pages. An example of this is the client information page that can be retrieved by $/client/1$ for client with the unique identifier 1 and $/client/2$ for client with the identifier 2, which are two distinct URL's. The consequence is that the number of unique pages are not as useful as in application log dataset, where the example client page was considered a single page because the parameters were not part of the path.

There are a few new attributes available that we can take advantage of. The HTTP method indicates whether the request is a GET or a POST request. The HTTP error is an HTTP response code as defined in Section 10 of the RFC 2616[9] and we consider a response code of 400 and higher to be an error. Each request also contains the size in bytes of the response.

Taking the same approach as before we group the requests together, this time using the IP address and the same session timeout of 30 minutes. The final session dataset has the following attributes:

- Unique identifier
- IP address
- Session start time
- Session end time
- Session duration in seconds
- Total number of GET requests
- Total number of POST requests
- Number of HTTP errors
- Number of GET requests per minute
- Number of POST requests per minute
- Total size in bytes of data transferred in this session
- Average size in bytes per request

7.3.1 Training the Self-Organizing Map

Our session dataset from load balancer 1 contains 841 sessions of which we use the first half to train our model. All the new attributes are used during training.

In most cases a POST request is preceded and followed by a GET request, simply because before a user can change anything he has to navigate to that page. To prevent a page reload modifying the same information, a user is usually redirected to another page after a POST, which accounts for the GET request after the POST. The number of GET requests should outnumber the number of POST requests by a factor of at least 3 to 4 and is usually even higher (as a comparison, the ratio in our total dataset is almost $1 : 14$). They are related however and any significant difference in the ratio can certainly be used to detect sessions where a lot of data is being changed.

The number of HTTP errors is interesting because it indicates how many times the user tried to access non-existing information, information it did not have permission to access or other problems such as for example server errors. Ideally this number should be zero (or close to zero), if the user navigates using only the links in the application. There are some cases where an outdated bookmark for example can lead to "404 Page Not Found"-errors, but most of the time this error code means something unusual is going on.

Finally the number of bytes in the response tells us something about the amount of data a user is requesting in a session. The load balancer logs also contain the requests for all the assets used in the application, such as stylesheets, javascript, images, etc. A lower than average number might be an indication of automated attacks, where the script does not bother to download any of the assets for each request. However, it might be hard to distinguish from clients that use certain forms of caching.

We use a subset of 397 session to train the Self-Organizing Map, this size has been determined similarly to our first attempt. Table 7.7 gives an overview of the results we obtain when using different network sizes. The dataset is larger than the user specific session in experiment 2, so we start with a slightly decreased learning rate. The lower learning rate means that a single object in the training dataset has less of an influence on the network when training. To make sure the Self-Organizing Map is accurate when detecting outliers we use an epoch of 1000, this means the network has more time to stabilise itself.

Size	Quantization error threshold	Outliers	FP	FN	run time
4×4	0.89	2	2	3	26 ms
8×8	0.60	3	2	1	59 ms
12×8	0.63	5	3	1	83 ms
16×8	0.60	4	3	2	109 ms
16×16	0.60	1	1	1	203 ms

Table 7.7: Results for differently sized trained networks. Outliers indicates the number of outliers detected. FP and FN show the number of false positives and false negatives respectively. The learning rate is 0.4, training size is 397 with an epoch of 1000.

The larger number of attributes chosen, compared to our previous experiments, caused an increase in the number of false positives detected. Where previously the distance of every attribute counted equally when calculating the quantization error, we now introduce an additional weighting factor. This allows us to give certain attributes a lower or higher priority, where they otherwise would have dominated or would have been cancelled out when determining the final quantization error.

Like the choice of the quantization error threshold, the actual values for these weights is a manual process. We want the total number of POST requests and the number of HTTP errors to be more important so they will have a larger weight factor. After some experiments a weight factor of 2.0 for these important attributes and 0.9 for all other attributes made their contributions to the quantization error fair. Given these results we will now continue to use the 8×8 network size.

7.3.2 Results

There are three objects classified as outliers and they are shown in Table 7.8. The first false positive is characterized by a large number of POST requests and errors and the second by an even larger number

of POST requests. The attack that was also classified as an outlier had a large amount of errors. These two attributes have a higher weight than the rest, which explains why these sessions were seen as outliers.

id	start	length	# POST requests	# errors	outlier type
251	08-07-2011 11:08:08	3301	92	35	FP
597	01-08-2011 19:51:42	868	2	29	Attack 2.1
756	15-08-2011 10:58:17	3784	195	0	FP

Table 7.8: Outliers in loadbalancer sessions from dataset 1. The length is measured in seconds. PRC is the POST request count: the number of POST requests in the session.

After investigation of the data contained in the outlying sessions, we see that the reason for the high number of POST requests in the false positives is due to a particular feature in the web application. That particular feature consists of a large number of pages through which a user can quickly browse using 'next' and 'previous' buttons. Each of these buttons generates a POST request since anything on this page can be modified, similar to an online survey with multiple pages. These pages are also extensively used to lookup information which is done by quickly browsing and skipping through a large number of pages in a short period of time, generating a large number of POST requests.

The reason the two other attacks are not classified as outliers is because while their attributes might be different than the normal data, they are not different enough to have a high enough quantization error. Figure 7.4 gives an overview of the quantization errors in this dataset.

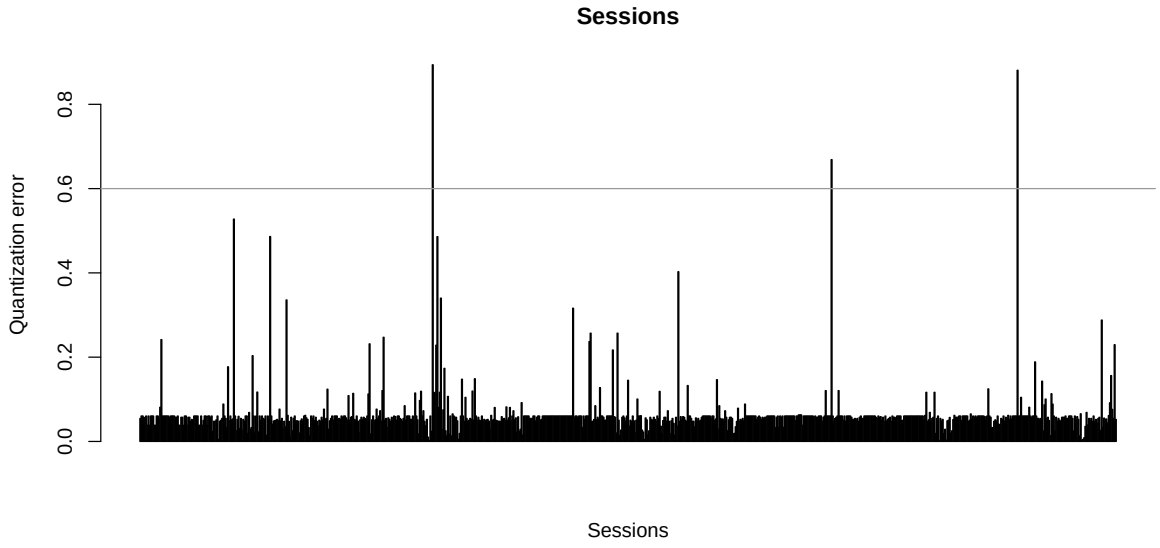

Figure 7.4: Quantization errors for the loadbalancer sessions dataset, with a threshold of 0.6.

7.4 Detecting malicious data manipulation

Our second dataset contains the actions performed by users playing an online game. The various actions a user performs are logged together with the final score, should the user decide to submit it. This dataset is also limited in the amount of information that is present, but the characteristics are similar to our first dataset. A single game consist of a number of distinct actions that need to be performed before a valid score can be submitted. It is possible that some games have multiple score submissions due to some limitations in reconstructing the game session based on individual actions. This is the reason we keep track of the number of times a score was submitted. We reconstruct the games and their actions

in a similar way we reconstructed the http sessions, resulting in a new game session dataset with the following attributes:

- Unique identifier
- Game start time
- Game end time
- User identifier (IP address)
- Game length in seconds
- Number of actions performed during the game
- Number of times a score was submitted
- The maximum score submitted

Given the fact that users can receive prizes based on their score, there is an incentive for users to cheat the game. The attacks we performed are designed to trick the system into accepting a highscore in a situation where it should not have. Users do not have access to a list of current high scores, which makes it hard to guess a score that is only marginally better than the current high score. Note that a score is based on the speed of which the user completed the game. The faster a game was completed, the lower the score, which means that a lower score is better. The best scoring player at this time has a score of 5638 and the worst scoring player has a score of 30000.

Attack 3.1: Submitting a high score
The attacker intercepts the request that sends the highscore and modifies the data to a much higher score. The session itself will be like any other valid game, except the high score is obviously incorrect and impossible for an honest player to achieve.

Attack 3.2: High score without playing
The attacker crafts an HTTP requests that submits a score without playing the game. The session will be short since the player did not complete the game, but it will contain many score submit actions.

7.4.1 Results

The Self-Organizing Map is configured again with different parameters to test their effectiveness in detecting these attacks. The dataset contains 797 game sessions where a user submits a score, so we choose the training set to be the first 400 sessions.

Size	Quantization error	Outliers	FP	FN	run time
6×6	0.56	2	0	0	35 ms
8×8	0.80	2	2	2	53 ms
10×10	0.80	2	2	2	108 ms
12×12	0.80	2	2	2	192 ms

Table 7.9: Results for differently sized trained networks. Outliers indicates the number of outliers detected. FP and FN show the number of false positives and false negatives respectively. The learning rate is 0.3, training size is 400.

Table 7.9 shows the results for the different network sizes. The smallest size, 6×6, gives the best results by detecting both attacks without any false positives. These results are obtained by again manually choosing the correct quantization error threshold. An interesting result is that all the other sizes have similar results and increasing the size does not seem to have an effect on the distribution of the quantization errors. A possible explanation is that the dataset is small and the contents only have a small amount of information. Increasing the number of nodes in the network in this case does not

change the trained model. The outliers that are detected are all false positives and the same objects are classified as outlier regardless of the size of the network.

It is clear, based on these results, that only the smallest network size for the Self-Organizing Map is usable. In Figure 7.5 the quantization errors for all sessions are shown.

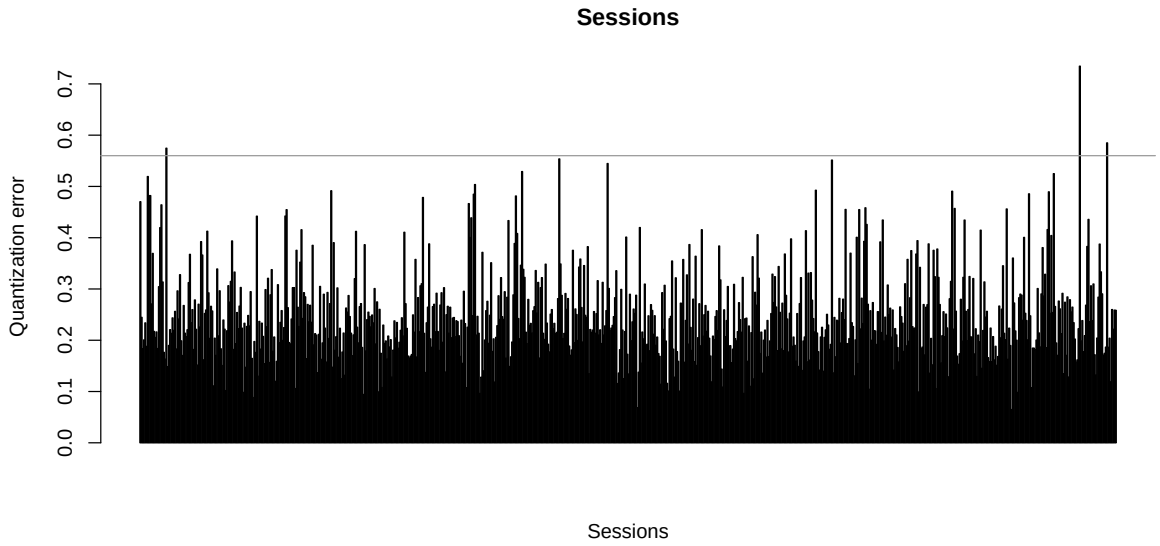

Figure 7.5: Quantization errors for the game sessions dataset, with a threshold of 0.56.

The quantization errors are higher than we have seen so far in the other datasets. This makes the choice for a good quantization error threshold much more difficult and it makes false positives more likely, especially when this dataset will grow over time without adjusting the Self-Organizing Map parameters. Given these results, outlier detection is not really recommended for this particular dataset as there are no obvious outliers when the differences between the outliers and non-outliers are so small.

Table 7.10 contains the two game sessions classified as outliers, both of which are the attacks we performed, 768 and 790.

id	start	length	# actions	# score
768	2011-10-03 20:10:45	66	8	100
790	2011-10-03 21:45:59	10	1	5500

Table 7.10: Outliers in the game session dataset. The length is measured in seconds.

Chapter 8

Conclusion

Having seen the capabilities of existing webbased anomaly detection techniques in Chapter 3 and their limitations in our particular case, we look at other ways of detecting anomalous behavior using outlier detection, for instance when no full payload is available due to privacy concerns. The goal presented in this paper is to find out if outlier detection techniques are able to detect anomalous behavior in webbased network traffic lacking the payload data that is often relied on. Our approach therefore focuses on the cases where only a limited amount of information is available in the captured logfiles, for which these anomaly detection techniques are not suitable. This also means that the kind of attacks we try to detect are different than the ones in the existing anomaly detection techniques.

Based on the ideas presented in the flow-based intrusion detection paper, discussed in Chapter 3.5, we propose a method of recreating the flows in the collected logfiles in order to extract meaningful information from the data. Using information from the logfiles that can identify, for each request, the user that performed that request, we are able to reconstruct the original user session that generated the entries in the logfile.

Every request from the original dataset is mapped to a session and can be used to aggregate data from the individual requests. The actual information that can be obtained depends on the available attributes from the original dataset, such as for example timestamps, error codes, etc. These sessions form the basis of a new dataset which now has sufficient information so it can be supplied to an outlier detection algorithm. There are many different outlier detection algorithms to choose from, as we have discussed in Chapter 4, from which we chose the Self-Organizing Map approach.

To answer our research question, whether we can use outlier detection to detect anomalous activity in logs of webbased network traffic lacking the payload data, we perform a number of experiments on various datasets to evaluate our approach. Based on these results we can say that outlier detection works only in a limited number of scenarios.

The Self-Organizing Map is a good choice in our case and even though it requires some knowledge about the datasets to correctly configure and tune the parameters, it is able to detect the outliers. Using this approach we are limited to threats that have sufficiently different characteristics compared to non-threats. In practice this limits us in detecting mostly automated threats. These are characterized by a very high number of actions or requests in a short period of time. Other threats that are detectable are the ones where numerical data can be directly related to whether or not it is a threat, as we can conclude from the results in experiment 4 in Chapter 7.4.1.

The manually performed attacks however are very hard to differentiate from normal user behavior, both using a general model of all user behavior and using user-specific detection models, when using our current approach. Because of the similarity of the attacks versus the normal activity, the outlier detection technique is unable to reliably detect the attacks as outliers. Trying to increase the detection rate of the outlier detection algorithm leads as a consequence to an increase in the number of false positives being raised.

The current approach of grouping the requests into sessions and simply using aggregated data from this session does not produce the required information needed to detect most of the attacks. As a consequence, we do not expect other outlier detection algorithms to perform different. To improve the anomaly detection accuracy in these cases other approaches must be taken in addition to the current way of recreating the user sessions, to extract meaningful data suitable of differentiating between normal

and anomalous behavior.

One of the possible improvements could be in the order in which various pages are being requested. The data we we work with contains the full path to the file or page being requested, this allows you to calculate the probabilities that a sequence of pages are being accessed in that order. If the sequence of requests in a session can be compared to this model in a way that produces a number indicating whether or not it is an expected workflow, then this is an added attribute for the outlier detection algorithm. The same can be done by using the timestamp to build a model for each user, if they regularly access the web service at particular times or days we can detect any deviation from that pattern.

Another possibility we have not investigated is combining various information sources. A single request can often lead to a number of actions on different services, each with their own logfile. For example, we could combine the logfiles from the load balancer, the application server and the database server. If the logfile entries of different sources can be linked together, this could be used as another way to see what kind of actions are caused by that single request.

Appendix A

Software implementation

Our software implementation consists of a number of different utilities. What follows in this appendix is an overview of all the utilities and their purpose. The source code can be obtained at `http://stemmertech.com/outlier-detection/`.

All the utilities operate on files in a common file format containing the datasets. The format chosen to represent the datasets is the Attribute-Relation File Format [1] (ARFF). This is a text-based, comma-separated values format also containing a description of each of the attributes contained in that file and their data types.

parser
: The *parser* takes a logfile as input and produces an ARFF file as output. It support a number of different logfile formats and can be extended to support other formats.

combine
: *Combine* takes a collection of ARFF files and combines them into a single ARFF file by concatenating the data of each input file. It is assumed that the attribute description and data types of all the input files are identical.

group
: *Group* groups the contents on the input file based on the given attribute and produces a new dataset with aggregated values from the created groups. This utility is superseded by the *session* utility.

convert
: *Convert* converts an ARFF file into a Structured Query Language (SQL) file suitable for being imported into a database.

var
: *Var* analyzes the contents of an ARFF file and reports the variability of each attribute. This utility can be used to identify attributes that can be suitable for the grouping utility to group the data by.

session
: The *session* utility is the actual implementation of Algorithm 1 and groups the dataset by a given attribute, taking into account a session timeout value and producing a new dataset with the aggregated values of each group.

som
: The *som* utility is the actual Self-Organizing Map implementation. It performs both the training of the model and the detection of outliers. It takes a configuration file as input and produces a file containing the outliers as output. The configuration file contains the location of the input and output files and all the necessary configuration parameters such as for example the training size, network size and quantization error threshold.

[1] `http://weka.wikispaces.com/ARFF`

Bibliography

[1] Charu C. Aggarwal and Philip S. Yu. Outlier detection for high dimensional data. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 37–46, New York, NY, USA, 2001. ACM.

[2] Fabrizio Angiulli, Stefano Basta, and Clara Pizzuti. Detection and prediction of distance-based outliers. In *Proceedings of the 2005 ACM symposium on Applied computing*, SAC '05, pages 537–542, New York, NY, USA, 2005. ACM.

[3] Stephen D. Bay and Mark Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 29–38, New York, NY, USA, 2003. ACM.

[4] Alan Bivens, Rasheda Smith, Mark Embrechts, Chandrika Palagiri, and Boleslaw Szymanski. Network-based intrusion detection using neural networks. In *Proc. ANNIE 2002 Conference*, pages 10–13. ASME Press, 2002.

[5] D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. PhD thesis, University of Twente, Enschede, June 2009.

[6] Damiano Bolzoni and Sandro Etalle. Boosting web intrusion detection systems by inferring positive signatures. In *Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems*, OTM '08, pages 938–955, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Markus Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jrg Sander. Lof: Identifying density-based local outliers, 2000.

[8] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In Christopher Kruegel, Richard Lippmann, and Andrew Clark, editors, *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 63–86. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74320-0_4.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[10] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010. [Online; accessed 8-April-2011].

[11] Anup K. Ghosh, James Wanken, and Frank Charron. Detecting anomalous and unknown intrusions against programs. In *In Proceedings of the Annual Computer Security Application Conference (ACSAC98)*, pages 259–267, 1998.

[12] Amol Ghoting, Matthew Eric Otey, Srinivasan Parthasarathy, and The Ohio. Loaded: Link-based outlier and anomaly detection in evolving data sets. In *In: 4th IEEE International Conference on Data Mining, IEEE Computer Society*, pages 387–390, 2004.

[13] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[14] Edwin M. Knorr and Raymond T. Ng. Finding intensional knowledge of distance-based outliers. In *VLDB*, pages 211–222, 1999.

[15] Edwin M. Knorr, Raymond T. Ng, and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal*, 8:237–253, February 2000.

[16] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 251–261, New York, NY, USA, 2003. ACM.

[17] J.Z. Lei and A. Ghorbani. Network intrusion detection using an improved competitive learning neural network. In *Proceedings of the Second Annual Conference on Communication Networks and Services Research*, pages 190–197, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Alberto Muoz and Jorge Muruzbal. Self-organizing maps for outlier detection. In *Neurocomputing*, volume 18, issues 1–3, pages 33 – 60, 1998.

[19] Kun Niu, Chong Huang, Shubo Zhang, and Junliang Chen. Oddc: outlier detection using distance distribution clustering. In *Proceedings of the 2007 international conference on Emerging technologies in knowledge discovery and data mining*, PAKDD'07, pages 332–343, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] Open Web Application Security Project. Owasp top 10 for 2010. `http://www.owasp.org/index.php?title=Category:OWASP_Top_Ten_Project&oldid=106979`, 2010. [Online; accessed 19-March-2011].

[21] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. *SIGMOD Rec.*, 29:427–438, May 2000.

[22] Craig Rhodes, James A. Mahaffey, and James D. Cannady. Multiple self-organizing maps for intrusion detection. In *Proceedings of the 23rd National Information Systems Security Conference*, 2000.

[23] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. Wavecluster: A multi-resolution clustering approach for very large spatial databases. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 428–439, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[24] Yingbo Song, Angelos D. Keromytis, and Salvatore J. Stolfo. Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic.

[25] Anna Sperotto. *Flow-Based Intrusion Detection*. Wöhrmann Print Service, 2010.

[26] Dantong Yu, Gholamhosein Sheikholeslami, and Aidong Zhang. Findout: finding outliers in very large datasets. *Knowl. Inf. Syst.*, 4:387–412, October 2002.

[27] Stefano Zanero. Analyzing tcp traffic patterns using self organizing maps. In *ICIAP 05: Proc. 13th International Conference on Image Analysis and Processing, volume 3617 of LNCS*, pages 8–8. SpringerVerlag, 2005.