School of Engineering
InIT Institute of Computer Science

Bachelor thesis Computer Science

Anomaly Detection for Web Access Logs with Machine Learning

Author	Matthias Koch Oliver Wiedler
Main Supervisor	Dr. Ariane Trammell
External Expert	Pascal Imthurn
Industrial partner	Next Stride AG
Date	07.06.2024

DECLARATION OF ORIGINALITY

Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:	**Name Student:**
Winterthur, 07.06.2024	Matthias Koch
Winterthur, 07.06.2024	Oliver Wiedler

Zusammenfassung

Durch die zunehmende Digitalisierung werden immer mehr sensitive Daten in Webanwendungen verarbeitet. Diese sind ein attraktives Ziel für Cyberkriminelle. Um einen allfälligen Angriff entdecken zu können, loggen Webserver kontinuierlich Zugriffe auf gehostete Webanwendungen, was insbesondere in der Bankenbranche, wo Anwendungen mit sensiblen Daten und eingeschränkten Ressourcen arbeiten, von entscheidender Bedeutung ist. Regelmässige Zugriffsmuster sind typisch, aber bösartige Aktivitäten erzeugen unregelmässige Logs, insbesondere während systematischer Suchen nach Daten oder Schwachstellen oder auch bei Angriffen gegen die Verfügbarkeit des Systems (DoS). Diese Arbeit stellt drei Prototypen vor, die einen LSTM-Autoencoder nutzen, um solche Anomalien zu erkennen. Die Prototypen wurden mit etwa 14,5 Millionen Logeinträgen eines produktiven Webservers über einen Zeitraum von neun Wochen trainiert. Es wurde eine detaillierte manuelle Aufbereitung der Logs durchgeführt, bei der Daten extrahiert und kategorisiert wurden. Anschliessend wurden die Prototypen an Logs aus einem späteren Zeitraum getestet, in dem überwachte Angriffe mit Zustimmung der Beteiligten durchgeführt wurden. Die Leistung wurde anhand der Kennzahlen Recall, Precision und F1-Score bewertet. Der beste Prototyp erzielte einen Recall von 99,85%, eine Precision von 94,10% und einen F1-Score von 96,89%, was im Vergleich zu ähnlichen Publikationen konkurrenzfähige Werte sind. Die hohe Recall-Rate zeigt, dass der Prototyp in der Lage ist, Anomalien in Webserver-Zugriffsprotokollen zuverlässig zu erkennen.

Abstract

Web servers continuously log accesses to hosted web applications, a critical practice in sectors like banking where apps handle sensitive data and restricted resources. Regular access patterns are typical, but malicious activities create irregular logs, particularly during systematic searches for data or vulnerabilities or DoS attacks. This paper introduces three prototypes leveraging an LSTM autoencoder to detect such anomalies. The prototypes were trained on approximately 14.5 million log records from a production web server over nine weeks. Detailed manual preprocessing was conducted on the logs, involving the extraction and categorization of data. They were then tested on logs from a later period, during which supervised and consensual attacks were conducted. Performance was assessed using recall, precision, and F1-score metrics. The best prototype achieved a recall of 99.85%, a precision of 94.10%, and an F1-score of 96.89%, demonstrating competitive performance compared to existing methods. The high recall rate indicates the prototype's strong capability to reliably detect anomalies in web server access logs.

Contents

I INTRODUCTION

Due to the large number of cyber attacks a web server faces, it is crucial to be able to identify such attacks. To this end, web servers generate logs for each access to any application they are hosting. The sheer volume of data generated by these logs necessitates the use of machine learning techniques to analyze the data and detect potential attacks. In this work, three prototypes are developed to automate the recognition of anomalous access logs generated by a web server with an LSTM autoencoder. Additionally, this paper aims to provide insight into the inner workings of autoencoders, anomaly detection and data preprocessing as a general topic, as well as in the context of cybersecurity. A brief introduction to *neural networks*, *LSTM*, and *autoencoders* is provided in section II. The prototypes are evaluated in section VII and the one that performs the best is then taken further into a usable program. The recall of over 99% indicates that the prototype reliably detects anomalies in web server access logs. This introduction contains a more detailed definition of the attacks considered in this paper as well as an overview over the logs used. Furthermore, the logs in question originate from a web server that is running *Janereporting* [1]. Section II.*a* provides a short explanation of this application focusing on the essential interfaces and use-cases.

I.*a* CYBER THREATS

There are many types of cyber threats that can have a huge impact on privacy of businesses and individuals. It is also often crucial that business applications remain available. Typical threats range from Denial-of-Service (DoS) attacks, which disrupt server or application availability, to vulnerability scans that attackers use to gain elevated privileges [2, 3]. Tools like *Gatling* [4], *nmap* [5] and *Gobuster* [6] can help attackers find vulnerabilities in systems and initiate exploits [7]. Such attacks represent abnormal behavior compared to the regular usage of applications. If anomalies can be detected on systems, conclusions can be drawn about attacks. Since anomaly detection relies solely on identifying deviations from normal behavior, it can uncover most attacks, even unknown ones. Although anomaly detection cannot prevent attacks, it is essential for gathering information to determine if a system is under attack and to emphasize the importance of cyber defense.

I.*b* VAST AMOUNT OF DATA

An average usage of the application *Janereporting*, described in section II.*a*, generates around 300'000 access logs every day. Access logs are records of requests made to the web server with detailing information about URLs, response sizes and more attributes shown in table 1. This amount of data exceeds the capacity of manual checking for anomalies, resulting in a necessity of automatic checking to identify abnormal behaviours in application accesses.

I.*c* ASSUMPTIONS

This paper analyzes the given logs described in table 1 and develops an anomaly detection algorithm. The objective of the prototype is to analyze the problem of detecting anomalies on a fixed training and test set. The automation of relearning on current

data and updating the model periodically is out of scope for this paper. If further improvements are needed, the training can be done again with necessary changes. Since the application *Janereporting* and the web server are only accessible from an internal network, all logs used for training and validation are assumed to be safe and free of anomalies. This assumption is reinforced by the fact that no notion of any kind of attack can be found three months after the end of the period over which the logs were gathered. The problem of harmless anomalies is further discussed in section VIII. Due to limited data fields, test cases will be confined to DoS and brute-force attacks. To test a wider variety of attacks, fields such as user IDs and device IP addresses are needed. The IP address is essential for distinguishing between a DDoS and a DoS attack, while a user ID coupled with an access role model like LDAP enables the detection of privilege elevation.

The program developed in this work requires manual initiation by feeding log files in CSV format, matching the attributes described in section II.*a*, into the model. The output will be a CSV file containing all log records that have been identified as anomalous. This flow is shown in fig. 1.

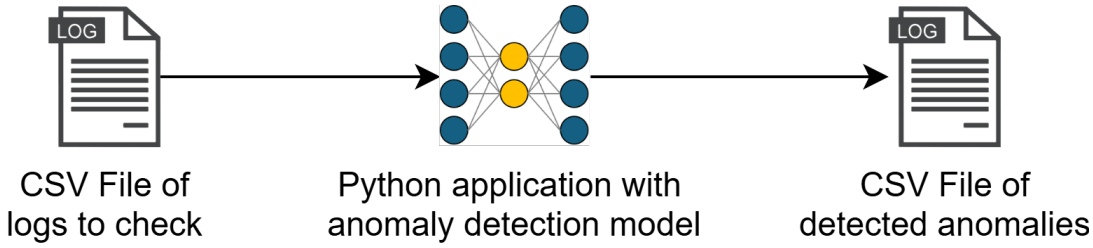

Figure 1: Requirement of in- and output of anomaly detection.

II BACKGROUND

This section outlines the application-specific and technical foundations for machine learning models that are necessary to understand the content of this paper. Section II.*a* provides an overview of the application, the access logs associated with it and used for training the prototypes. Section II.*b* is a technical deep dive that helps to understand the underlying technology for the anomaly detection models developed in this paper.

II.*a* JANEREPORTING

Janereporting is an application that is part of *Jane* [1], which is a business application developed by *Next Stride AG* [8]. This paper uses its logs as an example of access logs based on real, unsimulated application usage. *Janereporting* is a tool designed for the graphical visualization of a wide range of data. Detailed insights into the application are not necessary for understanding the collected logs, as they represent web access to the application.

Figure 2 visualizes how the logs in this paper are generated. On customer side, there exist two physical application servers with a running instance of *Janereporting*. These servers are located in two different data centers. A request to the server will be handled by a failover handler. By default the failover handler sends requests to the instance *Jane00100*. If *Jane00100* is unavailable, for example due to system updates or high traffic, the request will be sent to instance *Jane00120*. Both servers are accessible only through an internal network. The access logs associated with a server are each stored on the server itself and are periodically written to the log database.

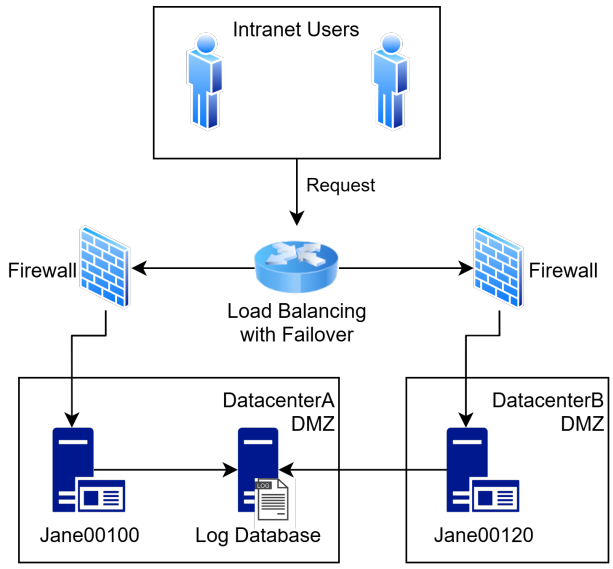

Figure 2: Architecture of customer's application access.

The attributes logged for application access are listed in table 1. These logs are all based on web access and primarily include common attributes of the HTTP protocol.

Attribute	Description	Examples
AGENT	Information of which software is making the request to janereporting.	• Wget/1.21.2 • Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0 Safari/537.36 Edg/119.0.0.0

HTTP_HOST	Requested server mostly accessed by domain name.	• jane-prod.intranet.customer.com • 10.10.100.101
METHOD	HTTP Method which was performed.	• GET • POST
PATH	Requested path **with** parameters.	• /janereporting/janereporting/miscService • /janereporting/janereporting/userServlet ?_operationType=getCurrentUser
PATH_CONTEXT	First path of URL, in the context of the application *Janereporting* it is always 'janereporting'.	• janereporting
PATH_SHORT	Shortened requested path **without** parameters.	• janereporting/janereporting/miscService • janereporting/janereporting/userServlet
REFERER	Web address of site which initiated the request.	• https://jane-prod.intranet.customer.com/ janereporting/ • https://jane-prod.intranet.customer.com/ janereporting/outputServlet? 86F2B0CF693C491DBA9B28E0762403C3/ eoqcbMotIWTy8uv1yvy1.html
REMOTE_HOST	Requested server accessed by IP address.	• 10.10.100.101 • 10.10.101.101
RESPONSE_CODE	The response code of the server.	• 200 • 300
SERVER	Which server was addressed for the request.	• jane00100 • jane00120
SHORT_LOG_ FILE_NAME	File name of the logs on the server but shortened. Either https or http.	• ns_proxy__https • ns_proxy__http
SIZE	Size of the response in bytes.	• 45 • 424
SOURCE_FILE	File name of the logs on the server. Either https or http.	• _proxy__https-access.log • _proxy__http-access.log
SOURCE_TS	Timestamp of the request. It is in the format YYYYMMD-DHHmmssSSS. The last three digits (ms) are always 0.	• 20240225134834000 • 20240313090557000

Table 1: Log-Attributes which are collected for access to the application *Janereporting* and can be used for anomaly detection.

Following attributes were not used for the anomaly detection model and therefore directly dropped.

- PATH_CONTEXT: Always the same entry *janereporting* because this is the key for the logs of the application discussed in this paper.
- REMOTE_HOST: The IPs have a one to one mapping to the domain names of attribute *'HTTP_HOST'*. Since DNS is more stable than IP addresses, this attribute was dropped.
- SOURCE_FILE: This attribute has the same information as *'SHORT_LOG_FILE_NAME'*.

All other listed attributes were split up into different fields to pass into the anomaly detection algorithm. How these attributes are interpreted and used is described in section V.*a*.

II.*b* Technical Knowledge

In this paper, an *LSTM-autoencoder* is used to detect anomalies within given log files. An autoencoder is a special format of a neural network, LSTM stands for Long Short-Term Memory. All of these terms will be further explained in this section. The underlying theory used in this paper is partly from the machine-learning and data mining lecture of Mark Cieliebak in 2022, specifically from the lessons L09 [9] and L10 [10], and partly from the book Neural Networks and Deep Learning by Michael Nielsen [11].

II.*b.1* Neural Network

A neural network consists of one input layer, an arbitrary number of hidden layers, and one output layer. Each layer consists of at least one artificial neuron, where a neuron has some number of inputs, an activation function, and an output. A simple example is shown in fig. 3.

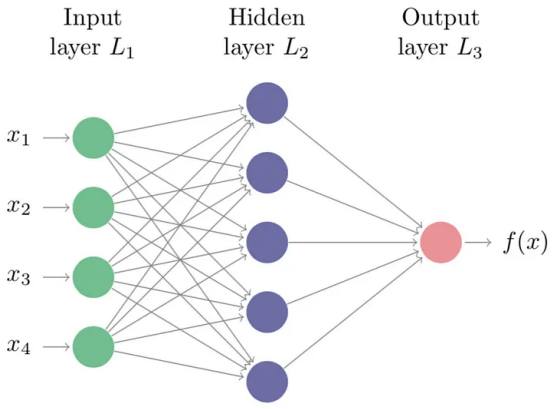

Figure 3: Neural network diagram with a single hidden layer. [12]

> "The simplest artificial neuron is the threshold logic unit or TLU. Its basic operation is to perform a weighted sum of its inputs and then output a "1" if this sum exceeds a threshold, and a "0" otherwise. The TLU is supposed to model the basic "integrate-and-fire" mechanism of real neurons." [13]

The activation function changes the TLU from something binary to something more *biological*, and it can, theoretically, be any real-valued function. In this paper, the only activation function used is *tanh*, as described in section IV.*b*. The output of a neuron with inputs x_1 and x_2, activation function *tanh*, bias b and weights w_1 and w_2 is given as

$$y = \tanh(w_1 * x_1 + w_2 * x_2 + b). \tag{1}$$

More generally, for the neuron q with N inputs, the output y_q is

$$y_q = \tanh\left(\sum_{i=0}^{N}(w_i * x_i) + b_q\right). \tag{2}$$

For the input layer, the output is given directly as $y = x$, as the input layer is the interface to external data and therefore defines the dimensions of the data to be processed. It has no parameters and is more of an abstraction than actually implemented. Each of the hidden layers, as well as the output layer, get the outputs of the previous layer as inputs. For simplicity, a fully connected network is considered here. Therefore, the output of layer L_p is given as

$$\vec{y}_p = \sum_{q=0}^{N_p}\left(\tanh\left(\sum_{i=0}^{N_{p-1}}(w_i * x_i) + b_q\right) \cdot e_q\right), \tag{3}$$

where $\vec{x} = \vec{y}_{p-1}$, and e_q are the unit vectors to get the scalar values in the correct vector space. To train such a network, one defines a loss function, and then uses backpropagation to update the weights and biases [10].

> "The Loss Function measures the error during training process that a neural network makes for the given samples." [9]

II.*b*.2 AUTOENCODER

An autoencoder is a special case of a neural network where the output layer L_p has the same size as the input layer L_1, see fig. 4. As the name suggests, the autoencoder encodes an input record to a lower-dimensional representation, and the use-case is to reconstruct the input values from this low dimensional representation on the output layer [15]. Therefore, an autoencoder can be trained unsupervised. For anomaly detection, an autoencoder has a specific advantage compared to other machine learning algorithms. The autoencoder learns to replicate input by learning what input it is fed. If it is given input it has never seen before,

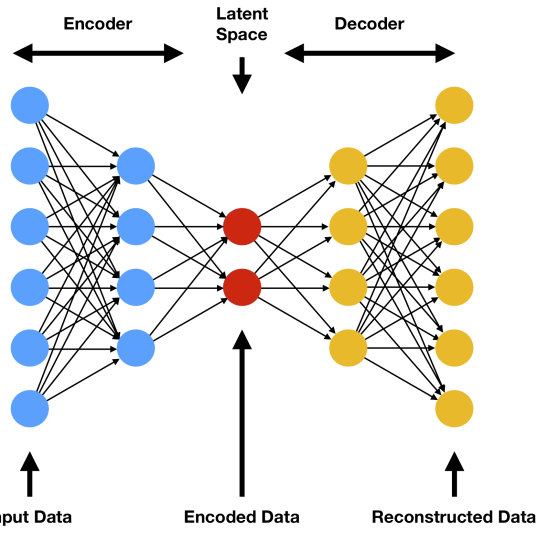

Figure 4: An autoencoder consists of an encoder and a decoder part, the middle layer acts as an interface between the two, and data in this middle layer is represented in a latent space. [14]

the autoencoder fails to replicate this input, resulting in large values for

$$f(y_p, y_1) = \left|\vec{y}_p - \vec{y}_1\right|. \qquad (4)$$

Using eq. (4) as a loss function, this property is trained for and therefore reinforced. Furthermore, if the autoencoder is trained on a dataset without any anomalies, then by definition, an anomaly is input the autoencoder has never seen before and thus yields a comparatively large value for $f(y_p, y_1)$. Thereby, an autoencoder changes the problem of defining *what means out of the ordinary* to finding out *how much out of the ordinary* a record is. After training, the autoencoder defines a metric which can then be compared to a simple threshold parameter to decide whether or not the record should be considered anomalous.

II.*b.3* LSTM

When looking at log files, a single log-entry often contains only a small amount of information. Therefore, when debugging or searching for anomalies, an engineer looks at several consecutive log-entries, to see whether the order or any sequencing might contain anomalies.

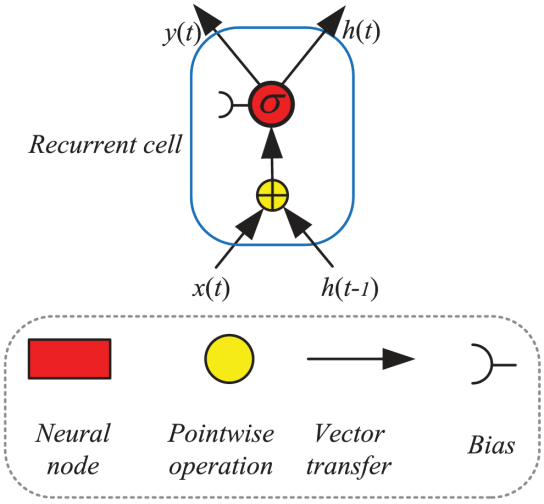

Figure 5: A recurrent cell shows two separate outputs: y(t), the output to the next layer, and h(t), the output to the next recurrent cell of the same layer. y(t) and h(t) contain the same information, the difference lies in the fundamental change of direction. Being handed over to another cell within the same layer in a later timestep, this allows the layer to *remember* information [16].

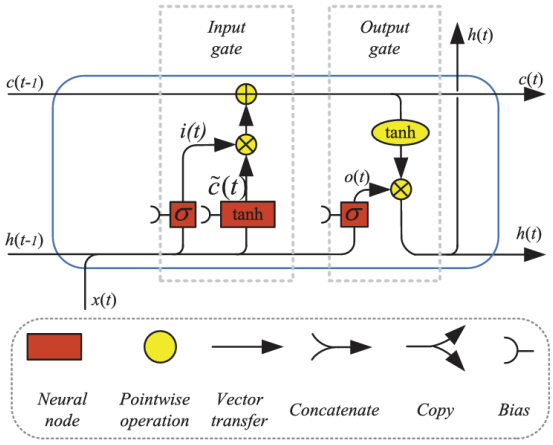

Figure 6: An LSTM cell as proposed by Hochreiter and Schmidhuber[17] shows better performance than the standard recurrent cell shown in fig. 5 [16].

To reflect this behaviour and to empower a neural network to *remember* previous records, long short-term memory or LSTM neural networks have been developed [18]. LSTM networks are a variation of recurrent neural networks, RNNs, which consist of

LSTM cells, see fig. 6, instead of, or additional to, artificial neurons. To understand the LSTM cell, one should first understand the basic principle it relies on, which is the recurrent cell shown in fig. 5. The recurrent cell changes the neural network from being a so-called *feed-forward* network, to a recurrent network containing feedback-loops. This difference is also shown in fig. 7. Feedback-loops enable the RNN to save a state in-between records being passed through the network, which in turn leads to that state containing information about the sequences in the data. The LSTM cell shown in fig. 6 introduces a *gate* into the cell, improving the capacity to remember compared to the standard recurrent cell. [16]

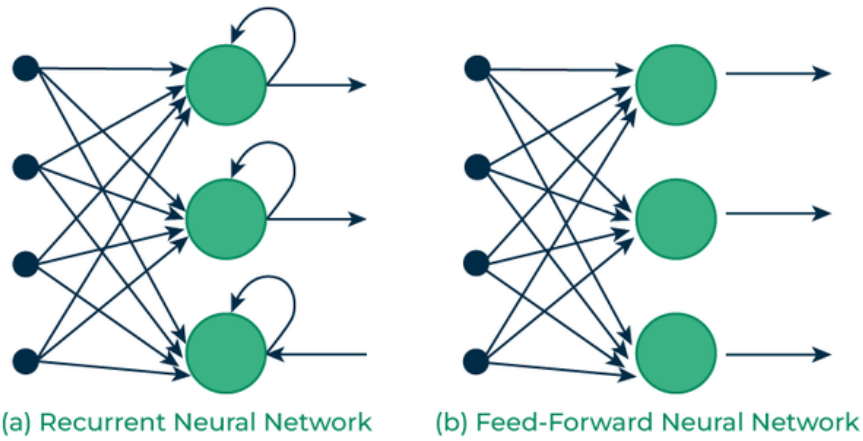

Figure 7: The difference between a recurrent neural network (RNN) and a feed-forward neural network (FFNN) is that the RNN contains feed-back loops and the FFNN does not. [19]

III Related Work

There are several publications concerning anomaly detection with machine learning. Many of them use *accuracy* and *F1-score* to measure the performance of their models. The empirical study of Kwon et al. [20] compares different convolution neural network (CNN) architectures to fully connected neural networks (FCN), variational autoencoder (VAE), and to sequence-to-sequence structure with LSTM (Seq2Seq-LSTM). Their work focuses on the comparison of these models on three different public traffic data sets. This paper, however, focuses on LSTM autoencoders on a private data set.

> "We observed that the shallow CNN model occasionally outperforms VAE models, but does not work better than FCN and Seq2Seq-LSTM."[20]

Le and Zhang [21] automated the preprocessing of logs by using *NeuralLog* to parse logs into semantic vectors. *NeuralLog* achieves an F1-score greater than 95%. The logs considered in the paper at hand do not have a semantic meaning, as they are machine-generated and not intended to contain phrases or similar linguistic constructs. The experience report of He et al. [22] contains a detailed review of commonly-used anomaly detection methods. Their findings include a general tendency of unsupervised learning methods being inferior to supervised methods. Again, publicly available data sets were used to compare different methods. The goal of their work is to provide guidelines for adopting these methods. For the design of the prototype developed in this paper, their guidelines are considered and used as a reference.

> "Unsupervised methods generally achieve inferior performance against supervised methods."[22]

Lu et al. [23] compare a CNN architecture to multilayer perceptron (MLP) and an LSTM neural network. They find accuracy scores of over 99% and precision and recall values of 97.7% and 99.3%, respectively, for their CNN. The MLP in their work achieves 98.1% precision and 98.0% recall, the LSTM they used achieved 95% recall and precision [24]. This provides a compelling argument to combine MLP and LSTM architectures, which is implemented in this study through the use of an autoencoder. The autoencoder, in essence, is an enhanced version of an MLP. Lindemann et al. [18] show a comparison of various different approaches to anomaly detection, including networks using LSTM technology and combinations of LSTM with autoencoders and VAE. However, their *TABLE 1* points out, that LSTM autoencoder lack comparisons to other methods. The most closely related method compared to others is the LSTM VAE, performing better than support vector machines (SVM) and autoencoder without LSTM. The autoencoder developed by Catillo et al. [15] achieves a recall in the range of 96% to 99%. Nassif et al. [25] recommend research use recent data and describe feature selection and extraction type in more detail than the papers they analyzed from the years 2000 to 2020. This document aims to follow their recommendations and provide a more detailed description of feature selection. Furthermore, Patcha and Park [26] highlight the importance of anomaly detection as a means to detect attacks not previously seen, called *zero day* attacks.

IV DESIGN

This chapter delves into the requirements for anomaly detection, outlining all the prerequisites and decisions necessary for a well-performing algorithm.

IV.*a* FROM RAW LOGS TO ANOMALY DETECTION MODEL

Figure 8 shows the development process for the anomaly detection model. First of all, training data is required. These are collected from the server and stored in a csv file with the log content described in table 1. Neural networks have an input layer and each node needs a float value as input. More theoretical insights can be found in section II.*b.1*.

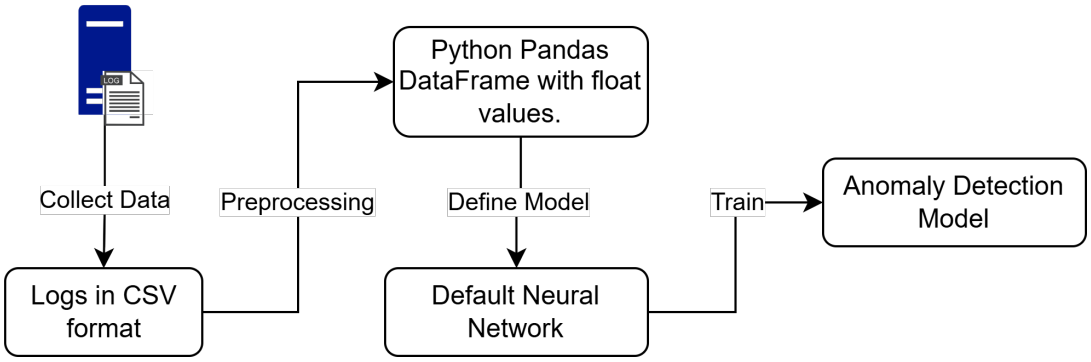

Figure 8: General procedure of creating the anomaly detection model.

To achieve this numerical input from textual log-data, preprocessing is necessary. One approach, as presented by Le and Zhang [21], utilizes automatic tokenization and classification with other models. However, for this paper, manual feature extraction was chosen to get the most control about which fields were preprocessed into which categories. The preprocessing is detailed in section V.*a*. With the data now represented only in numerical values, it is possible to pass them into a machine learning algorithm. The input layer of the model needs to fit the size of the preprocessed data. More details of the neural network are given in sections IV.*b* and V.*b*. After defining the model's architecture, the training process begins. This will tinker the weights and biases of the neurons (section II.*b.1*) and after a certain amount of training cycles (epochs), the model is done.

IV.*b* MACHINE LEARNING ALGORITHM

The amount of data which has to be combed through to find anomalous logs, combined with the task to learn what is anomalous, necessitates the application of a machine learning algorithm. To decide which kind of algorithm should be used in this prototype, several sources were considered [15, 18, 20, 21, 22, 23, 24, 25, 26], see section III. The analysis showed that several machine learning algorithms can be used to detect anomalies. Well known algorithms, such as Support Vector Machines (SVM) and decision trees, must be adapted specifically to solve single-class problems such as anomaly detection. For the SVM, this results in the so-called one-class-SVM. Additionally, the analysis indicated that a combination of LSTM with MLP, or an LSTM

autoencoder is a promising direction for further investigation.

Autoencoders themselves learn the patterns of normal logs. Because the reconstructed logs of unknown log patterns result in larger errors, these logs act like outliers and can be marked as anomalies. Therefore, it is not necessary to have labeled data, and the model can be trained using unsupervised learning. This enables the detection of anomalies that were not included in any evaluation processes, which assists in the identification of unknown attacks. Furthermore, an autoencoder alone would only be able to analyze a single log entry. Since logs are mostly dependent on each other, forming collective and contextual anomalies, it is necessary to evaluate them in a sequence.

> "The conducted study indicates that different LSTM network architectures are available and capable of precisely detecting a varying range of complex anomalies, such as collective and contextual anomalies."[18]

With the help of LSTM cells, it is possible to hold several logs in memory in a sequence and create connections between them. As a result of the analysis above, this paper focuses on the approach with an LSTM autoencoder. The choice of activation function, see section II.*b.1*, often depends on the hardware on which the prototypes will be trained. The hardware used in the course of this work restricts the options for the activation function in LSTM networks, with *tanh* being the only one supported for fast execution [27].

IV.*c* Loss Function

As this paper focuses on creating a prototype to find anomalies and not on optimal loss functions, the research on the best practise and most used loss functions is limited to the two most basic metrics, the *mean squared error* (MSE) and the *mean absolute error* (MAE). The MSE has the advantage of punishing big errors more severely, while the MAE is more lenient. On the other hand, the MSE is more susceptible to exploding and vanishing than the MAE due to the squared nature of the MSE. In contrast to MSE, the leniency of the MAE leads to slower convergence of the training process, resulting in more time consuming experiments. As training time is not an issue in this work, MAE is chosen for its better stability.

IV.*d* Evaluation

The test set used in this paper consists of the log data of six consecutive days. On two of these days, several attacks were conducted under supervision, see section VI, and the respective logs can therefore be labelled. To evaluate the performance of a prototype, its output is compared using the four variables shown in fig. 9. True positives (TP) are values that are actual anomalies and are correctly predicted like that by the model. Anomalies that are not detected by the model are called false negatives (FN). True negatives (TN) are log entries that are correctly labeled as not being anomalies, while false positives (FP) are legitimate logs that are incorrectly labeled as anomalies. The primary objective set for the prototype is to minimize FN. The secondary objective is to minimize FP. In technical terms, these objectives are called *recall* and *precision*. *Recall* in this context measures the ratio of logs correctly labelled anomalous by the

prototype (TP) to the total number of effectively anomalous logs, see eq. (5). *Precision* on the other hand, measures the ratio of TP to the total number of logs predicted anomalous by the prototype, see eq. (6). The *F1-score* is a metric which takes precision and recall into consideration, see eq. (7). It is commonly used to compare models over various data sets, especially in multi-class categorization problems, as it combines precision and recall into a single number. All of these metrics yield values between 0 and 1, where 1 is the desired value.

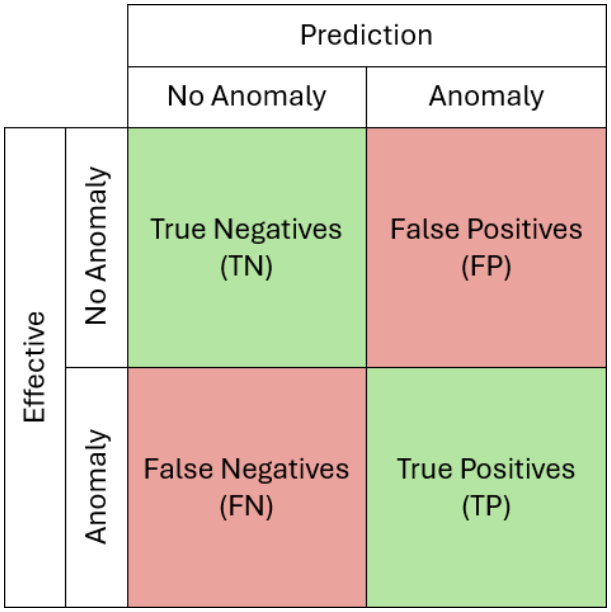

Figure 9: General concept of a confusion matrix.

$$\text{recall} = \frac{\text{TP}}{\text{TP}+\text{FN}} \tag{5}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP}+\text{FP}} \tag{6}$$

$$\text{F1-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision}+\text{recall}} \tag{7}$$

The metric to focus on depends on the type of detection that is being sought. For anomaly detection, it is crucial to identify as many effective anomalies as possible, which implies a high *recall*. *Precision* indicates how many of the normal logs are actually flagged as anomalies. Improving *precision* is also important as a second goal because marking too many logs as anomalies increases the manual effort to review them. The *F1-score* ensures that while aiming for high recall, precision does not drop to unacceptable levels. It also provides a single value to compare the performance of various models in the market. However, for this use case, *recall* remains the most critical metric.

V IMPLEMENTATION

This chapter describes the implementation of the preprocessing of the logs, the atoencoder prototypes, and the test setup. The language used for the effective implementation is python, with the frameworks of *numpy* [*28*], *pandas* [*29*], *keras* [*30*], and *tensorflow* [31]. For plotting, *matplotlib.pyplot* [32] is used.

First, the training data is imported into the python kernel as a *pandas.DataFrame* from multiple CSV files. It is then sorted by the field 'SOURCE_TS', before the preprocessing begins. After preprocessing, the autoencoder is defined and trained as described in section V.*b*. The trained autoencoder model is then used to detect anomalies and puts the according log entries into an output file. With the output generated from a test data set, the performance of the autoencoder can be evaluated as detailed in section V.*d*.

V.*a* PREPROCESSING

This section goes over each attribute of the logs and how it is processed in detail. Some of the original attributes are converted to one or more processed attributes with different names.

V.*a.1* SOURCE_TS

The attribute 'SOURCE_TS' contains a number of seventeen digits, representing a timestamp in the format *YYYYMMDDHHmmSSsss*. Since interpreting this directly as an integer number leads to various problems like missing values between 60 and 99 in the digits for minutes and seconds, this original attribute is converted to ten processed attributes. The first of these is called 'weekday', and it contains a float value between 0 and 1, where 0 is Monday morning at 12 a.m. and 0.994 is Sunday evening at 23:00. The calculation or lookup which weekday a certain date represents, is handled by the python library *datetime*. The second processed attribute is called 'is_holiday'. It contains a 1 if the date represents a saturday, a sunday, or a holiday in the region of the customer. For this paper, the list of holidays is manually compiled and hard-coded into the model. The third to tenth processed attributes are a bucket-representation of the time of day, in overlapping four hour steps. As an example, the attribute 'h0-3' contains a float value from 0 to 1, where 0 means 00:00:00.000 and 1 would be 04:00:00:000. Therefore, a log entry with 'SOURCE_TS' 20240327005959000 would get 'weekday' = 0.2857, 'is_holiday' = 0, 'h0-3' = 0, 'h3-6' = 0.2499, 'h6-9' = 'h9-12' = ... = 'h18-21' = 0, and 'h21-0' = 0.9999.

V.*a.2* SERVER

Since the attribute 'SERVER' contains either *jane00100* or *jane00120*, it is processed to contain 0 and 1 instead of *jane00100* or *jane00120* respectively.

V.*a.3* HTTP_HOST

The attribute 'HTTP_HOST' can contain one of six different values, these are *'10.10.100.101', '10.10.101.101', 'jane-prod.intranet.customer.com', 's100-jane-prod.intranet.customer.com', 's102-jane-prod.intranet.customer.com'*, and *'sv243554.zit.customer.com'*. This

is due to the way the customer accesses the application *Janereporting*, and the architecture using the failover handler mentioned in fig. 2. To decrease the risk of putting ordering into these values, one-hot encoding is used. Thus, the original attribute is converted to six processed attributes, each representing one of the possible values.

V.*a.4* METHOD

The web server is accessed over HTTP, therefore the requests are always made using a HTTP-method. This method is contained in the attribute 'METHOD'. The application *Janereporting* requires either a POST or a GET request, therefore the value 'GET' is mapped to 0 and the value 'POST' is mapped to 1.

V.*a.5* AGENT

In the attribute 'AGENT', the information about what kind of browser or tool was used to create the request. This can either be any browser like *Mozilla Firefox, Microsoft Edge*, or *Google Chrome*, or it can be a command-line tool like *wget*. A design choice was made to remove the information about the exact version of the browsers used, as it can change with every update any of the manufacturers release and it has no real impact on the usage of the application. Also, whether a customer uses *Chrome*, *Safari*, or another browser is ignored and the only information left in the attribute 'AGENT' is therefore whether or not a command-line tool was used. This results in the processed attribute 'IS_WGET', containing a 1 if a command-line tool was used, and a 0 otherwise.

V.*a.6* RESPONSE_CODE

The 'RESPONSE_CODE' attribute refers to the HTTP response code, where for example *200* stands for *OK*. The values found in the training set are *200, 302, 304, 301, 500, 408*. Therefore, and to reduce confusions about similar codes like *301* and *302*, the processed attributes *'2xx', '200', '3xx', '304', '4xx'*, and *'5xx'* are computed. The special cases of *200* and *304* take precedence over *2xx* and *3xx*, respectively.

V.*a.7* LOG_FILE

The web server used writes into two separate log files, depending on whether or not the request was made over HTTPS. Hence, the original attribute 'LOG_FILE' is converted to the processed attribute 'IS_HTTPS', containing either a 1 or a 0.

V.*a.8* SIZE

The attribute 'SIZE' contains the information about how many bytes are sent in the response to the request. To better understand what sizes are common within the training data set, the values of 'SIZE' were plotted over a subset containing logs from six consecutive days. These plots are shown in figs. 10 and 11. On the left side, in fig. 10, the unfiltered sizes are shown. As there are a few extrem outliers, the *normal* values are compressed to almost one apparent size. Therefore, a second plot, see fig. 11, contains only sizes smaller than 400000 bytes. In this second plot, two features are relevant for decisions in processing the attribute 'SOURCE_TS'. Firstly, the frequency of points in the region between 220000 and 325000 bytes shows clear signs of normal working

hours. Secondly, there are six distinguishable lines that grow over time and reset every day. For reference purposes, this kind of plot as shown in fig. 11 was created with several different weeks of data, all showing similar features about work-days, working hours, and holidays. These other plots are not shown in this paper as they are not further relevant. Also, in fig. 11, there are four specific values on the y-axis, at 20000, 105000, 220000, and at 325000 bytes. These separate the values into five regions. For processing, these five regions are used as buckets in order to bucketize the values. Therefore, the processed attributes corresponding to the size are *'<20k'*, *'20-15k'*, *'105-220k'*, *'220-325k'*, and *'>325k'*. Within these buckets, the values range from 0 to 1, where 0 is the lower and 1 the upper boundary.

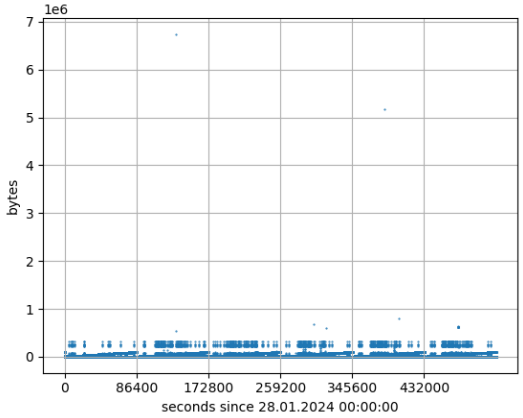

Figure 10: A plot of the size attribute of logs over six days from Sunday, 28.01.2024 vs. their timestamp.

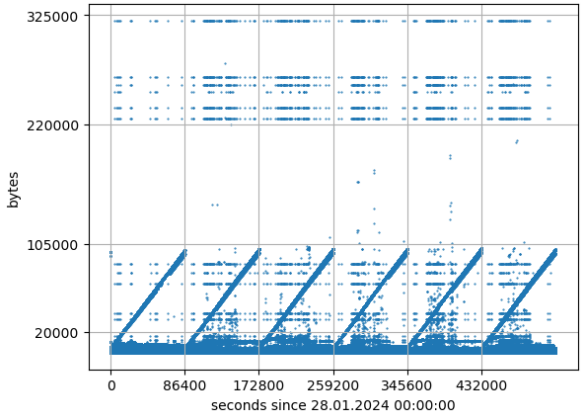

Figure 11: The same data as shown in fig. 10, filtered to only contain values smaller than 400000 bytes.

V.*a.9* PATH

The web server offers several different servlets that can be accessed in the context of using *Janereporting*. These are accessed over the respective context paths. A value count over the training data revealed that ten of these are used frequently, and all others comparatively rarely. This lead to the decision to categorize logs into eleven categories, one for each of the ten most-used context paths, and one for all others. Thus, the processed attributes are *'OUTPUT_SERVLET'*, *'REPORT_SERVICE'*, *'MISC_SERVICE'*, *'USER_SERVLET'*, *'NEWS_SERVLET'*, *'MENU_SERVLET'*, *'CONFIG_SERVLET'*, *'TIME_SERVICE'*, *'JANEREPORTING'*, *'JANEREPORTING/'*, and *'PATH_UNCATEGORIZED'*, and one-hot encoding is used to assign values.

V.*a.10* REFERER

The 'REFERER' attribute contains information about *from where* the user got to requesting a resource. This can mean for example that the user clicked on a link shown on the landing page, to request a document. Then, the 'PATH' attribute would show the path to the document, whereas the 'REFERER' would be the path to the landing page. Both, the 'PATH' and the 'REFERER', can contain a *jsessionId*. A *jsessionId* is used to identify sessions on application level. The session is tied to a user, but a user is not necessarily tied to a session, as a new session is created every time the user reconnects without sending the necessary cookies in the request. To condense this

information into float values, the processed attributes *'JID_IN_REF'*, *'JID_IN_PATH'*, *'JID_R_SEEN'*, *'JID_P_SEEN'*, *'JID_DIFFERENT'* contain information about this *jsessionIds*. The first two, *'JID_IN_REF'* and *'JID_IN_PATH'*, are set to 1 if a *jsessionId* is present in 'REFERER' or in 'PATH', respectively, and 0 otherwise. The next two, *'JID_R_SEEN'* and *'JID_P_SEEN'*, contain a 1 if a *jsessionId* is present in 'REFERER' or 'PATH', respectively, and has been encountered before and 0 otherwise. The last attribute, *'JID_DIFFERENT'*, is 1 if the *jsessionId* in 'REFERER' matches the *jsessionId* in 'PATH' and 0 otherwise.

V.*b* AUTOENCODER

The correct choice of hyperparameters such as the number of layers or their size is done by multiple experiments, observing the training process more closely over few epochs.

> "As for any machine learning study, the choice of the hyperparameters is guided by experimental tests carried out by analyzing the outcome [...] with respect to the validation set." [15]

With the *keras* [30] library, the code to implement an LSTM reduces to the following:

```
1  import keras
2  from keras.layers import LSTM,Dense,RepeatVector,TimeDistributed
3  from keras.models import Sequential
4  autoencoder = Sequential()
5  autoencoder.add(LSTM(70, activation='tanh',
   ↪  input_shape=(seq_size,n_features), return_sequences=True))
6  autoencoder.add(LSTM(42, activation='tanh', return_sequences=True))
7  autoencoder.add(LSTM(6, activation='tanh', return_sequences=False))
8  autoencoder.add(RepeatVector(seq_size))
9  autoencoder.add(LSTM(42, activation='tanh', return_sequences=True))
10 autoencoder.add(LSTM(70, activation='tanh', return_sequences=True))
11 autoencoder.add(TimeDistributed(Dense(n_features)))
12 autoencoder.compile(optimizer='adam', loss='mean_absolute_error',
   ↪  loss_weights=weights)
```

The variable *weights* used on line 12 is defined as follows:

```
1  weights = np.ones(n_features, dtype='float')
2  columnsWeighted = {'IS_WGET':1.1, "<20k":1.2 ,"20-105k":1.2,
   ↪  "105-220k":1.2, "220-325k":1.2, ">325":1.2}
3  for col,weight in columnsWeighted.items():
4          index = df.columns.get_loc(col)
5          weights[index] = weight
```

Here, *n_features* is the number of features after preprocessing. The weights are used to guide the model during training, by weighing the impact of deviations in the respective columns in the loss function. The definition of the autoencoder with LSTM layers is then complete, and the training process can be started. The calculation of the loss after each step, as well as the backpropagation are handled by the framework. To start the training process, the method *fit*() is called on the autoencoder object, as shown in the following code snippet:

```
autoencoder.fit(
  x=train_d,
  y=train_d,
  validation_data=(val_d,val_d),
  batch_size=batch_size,
  epochs=60
)
```

To train the autoencoder, more than 14.5 million log records were collected over nine weeks. The tenth week, containing 1.7 million log records, is used as the validation set. The model is not trained on the validation set, it is used as a reference to prevent or notice overtraining. These log records are preprocessed as described in section V.*a*. The variable *train_d* represents the *DataFrame* containing the preprocessed logs, and is used as the input x as well as the reference-output y. Note here, that the autoencoder tries to replicate its input, therefore, the output should yield the same as the original input. The parameter *validation_data* needs two *DataFrame*s as well. Again, the output should be the same as the input. The validation set is used to ensure the model is not overtraining, and in this case it consists of one week worth of log data that is not included in the training set. Furthermore, an overview of the training data indicated that a maximum of 71 log records have identical timestamps, therefore the sequence size is set to 100, containing at least two different timestamps. As a consequence, the *batch_size* is set to 200, so that two sequences can be trained in one batch. Additionally, a *shift_window_size* [33] is defined to allow the model to shift the sequencing window over the batch. As can be seen in fig. 12, the loss stagnates over many epochs, and after around 50 epochs drops again. This gave reason to use 60 epochs for training.

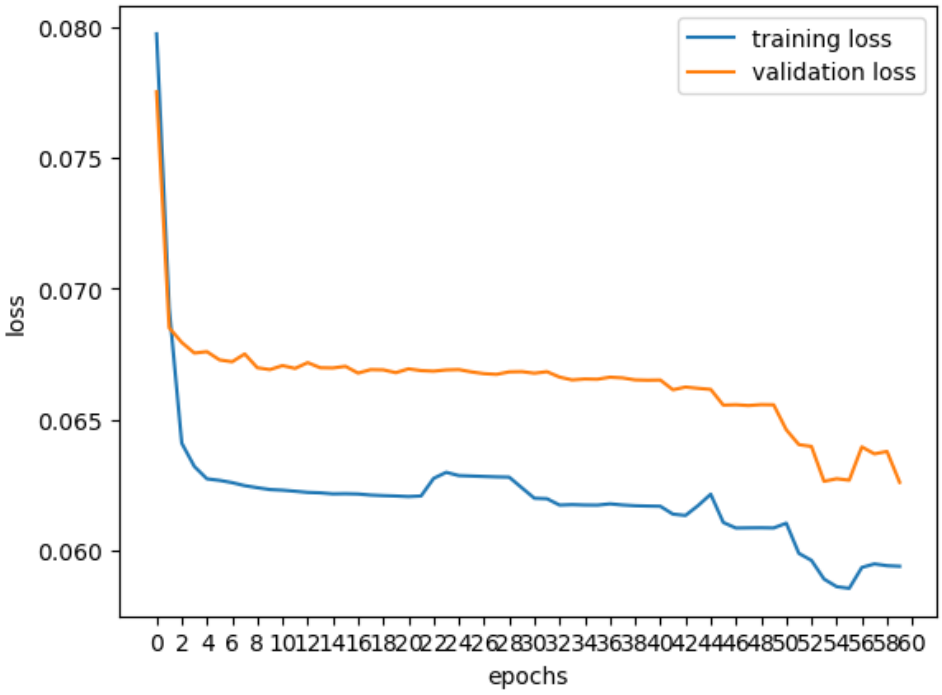

Figure 12: The history graph of the training of prototype P_B. The loss over the training set is plotted in blue, the loss over the validation set is plotted in yellow at the end of each epoch. One can see the loss reducing until about epoch 55.

V.*c* Output of the Prototype

With the trained autoencoder model from section V.*b*, the collected log data can be processed. For processing, the data must first be preprocessed, resulting in a *DataFrame* x which can then be input into the autoencoder. The output of the autoencoder itself is a *DataFrame* y with the same dimensions as x. The number of columns in x and y is *n_features* as defined in section V.*b*. Then, the mean absolute error is calculated over each row. To clarify this in mathematical terms, see eq. (8).

$$mae_i = \frac{1}{\text{n\_features}} \sum_{j=0}^{\text{n\_features}} \left| x_{i,j} - y_{i,j} \right| \tag{8}$$

Equation (8) defines mae as a vector, or an array, where each entry corresponds to the mean absolute error in reproducing one log entry. Afterwards, a manually set parameter is used as a threshold for the decision whether or not an entry is to be labelled anomalous. What value was chosen for the threshold is described in section VII. With this threshold and the array mae, the original *DataFrame* can be extended by a column *'ERROR_ABOVE_THRESHOLD'*. This columns contains the value of the corresponding entry in mae minus the threshold. The output of the prototype is therefore the original CSV file extended by the column *'ERROR_ABOVE_THRESHOLD'* filtered to contain only logs with a positive value in this new column.

V.*d* Evaluation

With the output described in section V.*c*, the metrics of recall, precision, and F1-score can be calculated by comparing the output to the effective values. This calculation is done with the package *sklearn.metrics*, more precisely with its methods *classification_report* and *confusion_matrix*. To get those metrics, all logs which are from attacks in section VI.*a* are marked *True* in the column *'IS_ANOMALY'*. To optimize the prototypes, the threshold parameter is defined variably, and plots are created to show recall, precision, and F1-score for different values, see figs. 15 to 17. The best threshold is then defined by comparing these values.

VI TEST SETUP

The anomaly detection model is trained exclusively on unlabeled data consisting only of legitimate logs, ensuring there are no anomalies. Additional testing is required to evaluate the model's performance. This chapter details how the testing was done.

VI.*a* ATTACKS

To test the model, anomalous data is needed. Therefore, some attacks were carried out to collect the anomalies. The log attributes of the data covered in this paper do not contain data about users, nor which machine was responsible for a request via an identifier such as IP addresses. This excludes attacks such as unauthenticated and unauthorized access from the use cases. DoS and brute-force attacks are not primarily tied to user data and are therefore valid use cases for evaluating performance of an anomaly detection model.

VI.*a.1* EXECUTING DOS ATTACK

In order to perform a DoS attack, a specific URL was requested, creating a load on the application for 1.5 seconds for each request. The requests were made using Gatling [4]. The attack was started at 50 users per second and increased to 150 users per second over a period of one minute, as shown in fig. 13.

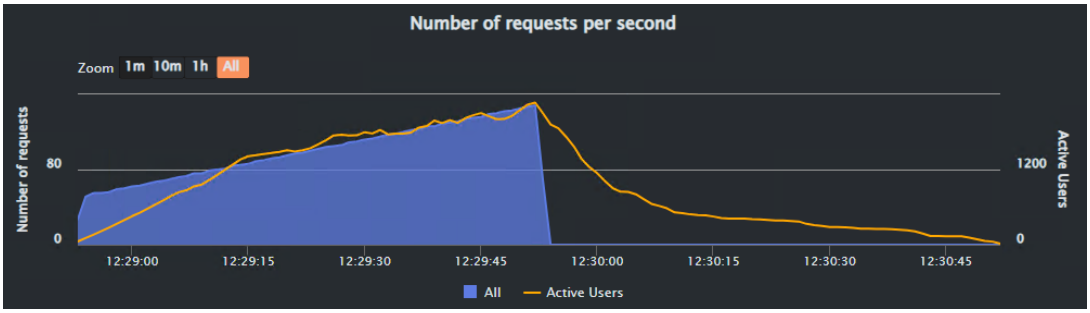

Figure 13: Graph from Gatling [4] of the number of requests for the DoS attack.

Figure 14 shows how many requests actually received a response. After about 20 seconds, you can see that most of the requests do not receive a response. In general of all 6000 requests, only 771 received a response. This represents a successful DoS attack where the service was mostly unavailable.

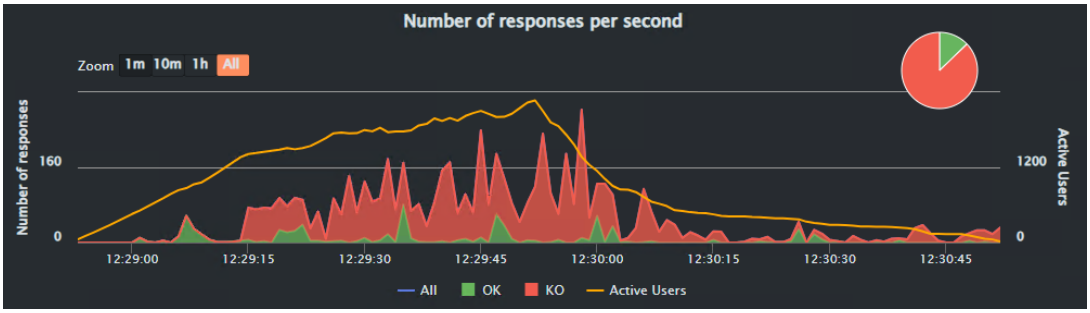

Figure 14: Graph from Gatling [4] of the number of responses from the server during the DoS attack.

VI.*a*.2 EXECUTING BRUTE FORCE ATTACK

The second attack performed on the server was a brute force attack against *Janereporting*. In this context, brute force involves systematically trying a large number of possible URL combinations to uncover hidden directories or files on a web server. This attack was carried out by a tool called *Gobuster* [6]. Gobuster used a word list containing of nearly 45000 potential directory and file names to query the server. In this attack simulation, due to the preprocessing that removes deeper paths, see section V.*a*.9, the attack was only targeted at a specific root directory. This resulted in most responses indicating that the resources were not found. Although nothing useful was found during the attack, the requests were executed and logs were generated for testing the anomaly detection model.

VI.*b* LABELING OF TEST DATA

To be able to evaluate the performance of the autoencoder, all logs which are representing an attack need to be labeled as effectively anomalous. This was done by addressing a specific path in the attack requests, so during testing, all these logs could be filtered and flagged as a true anomaly. The identification is done in the path, which is, according to the specification in section V.*a*.9, preprocessed to only contain information about the servlet used. Therefore, the identification does not impact the preprocessed *DataFrame*.

VII RESULTS

Three prototypes were composed in the course of this work. For referencing purposes, they are named P_A, P_B, and P_C. All of the prototypes are autoencoders consisting of one input layer of size 47, five LSTM layers, and one output layer of size 47. The input size is given by the number of features of the input *DataFrame*, called *n_features* in section V.*b*, after preprocessing. The output layer, by definition of autoencoders, must have the same size as the input. The five LSTM layers of P_A have the sizes 70, 42, 12, 42, 70 in this order, the five LSTM layers of P_B have sizes 70, 42, 6, 42, 70. Therefore, the difference lies only in the innermost layer with size 12 or 6, respectively, defining the size of the latent space. The smaller latent space of P_B forces it to generalize more during training. The third prototype, P_C, has hidden layers of sizes 60, 35, 5, 35, 60, resulting in only about 80% of trainable parameters compared to P_A and P_B. Also, the latent space is even smaller than that of P_B, ergo forcing it to generalize even more. To figure out the best threshold values, a graph is made for each prototype, showing their recall, precision, and F1-score with different values set as threshold.

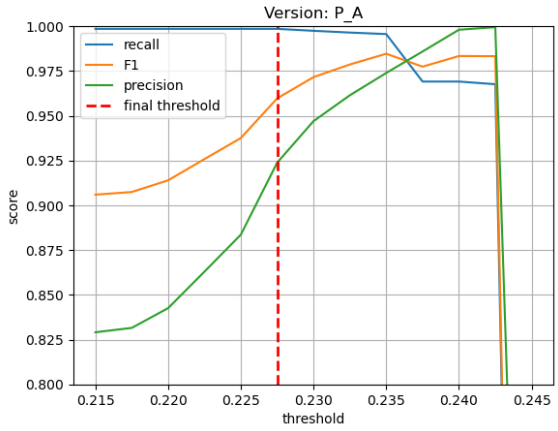

Figure 15: Performance of prototype P_A, measured by recall, precision, and F1 score with thresholds ranging from 0.215 up to 0.245.

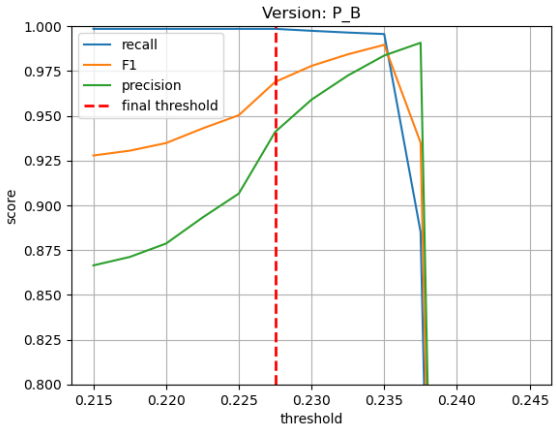

Figure 16: Performance of prototype P_B, measured by recall, precision, and F1 score with thresholds ranging from 0.215 up to 0.245.

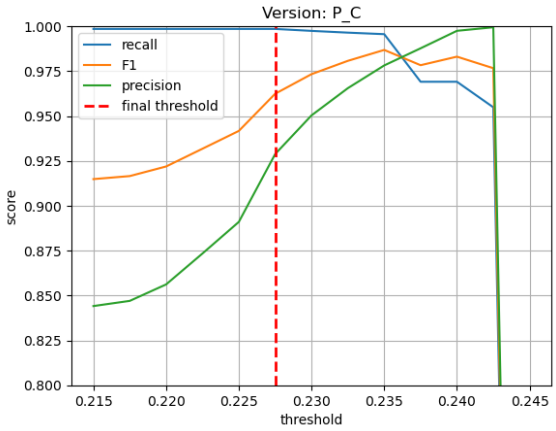

Figure 17: Performance of prototype P_C, measured by recall, precision, and F1 score with thresholds ranging from 0.215 up to 0.245.

As can be seen in figs. 15 to 17, the optimal threshold for all three prototypes is 0.2275. In this case, optimal means that a higher value would result in a smaller recall value, which would work against the primary objective, and a lower value would result in a smaller precision, working against the secondary objective. In figs. 18 to 20, the confusion matrices of the three prototypes with a threshold of 0.2275 are shown. The values for true-positives, true-negatives, false-positives, and false-negatives allow for the calculation of recall, precision, and F1-score. Table 2 shows the scores of P_A, P_B, and P_C with the threshold 0.2275 in more detail. The second prototype, P_B, is highlighted because it achieves a higher precision and F1-score than P_A and P_C.

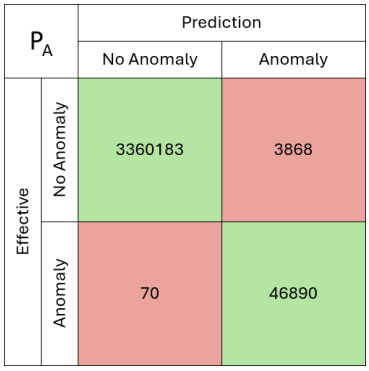

Figure 18: Confusion matrix of prototype P_A.

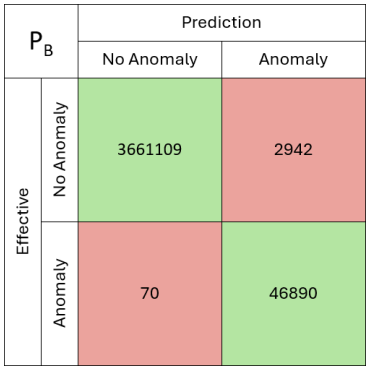

Figure 19: Confusion matrix of prototype P_B.

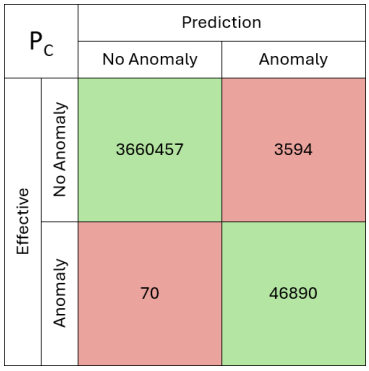

Figure 20: Confusion matrix of prototype P_C.

Prototype	Recall	Precision	F1-Score
P_A	0.9985	0.9238	0.9597
$\boldsymbol{P_B}$	**0.9985**	**0.9410**	**0.9689**
P_C	0.9985	0.9288	0.9624

Table 2: The metrics of the performances of the three prototypes P_A, P_B, and P_C with the threshold of 0.2275.

Figure 21 shows the mean absolute error produced in regenerating each of the test set logs. The logs of normal usage are in blue, the known anomalies generated by the attacks described in section VI.*a* are marked in dark red. Additionally, the threshold of 0.2275 is shown as a dashed red line. On day 10 of the test set, the DoS attack section VI.*a.1* was conducted, the marked data points correspond to this attack. Similarly, on day 11, the brute-force attack section VI.*a.2* was carried out and the data is marked accordingly. Comparing fig. 19 and fig. 21, one can see the 70 log records that were falsely labelled negative below the threshold. On the other hand, there are approximately 3000 blue dots above the threshold, misclassified as anomalies.

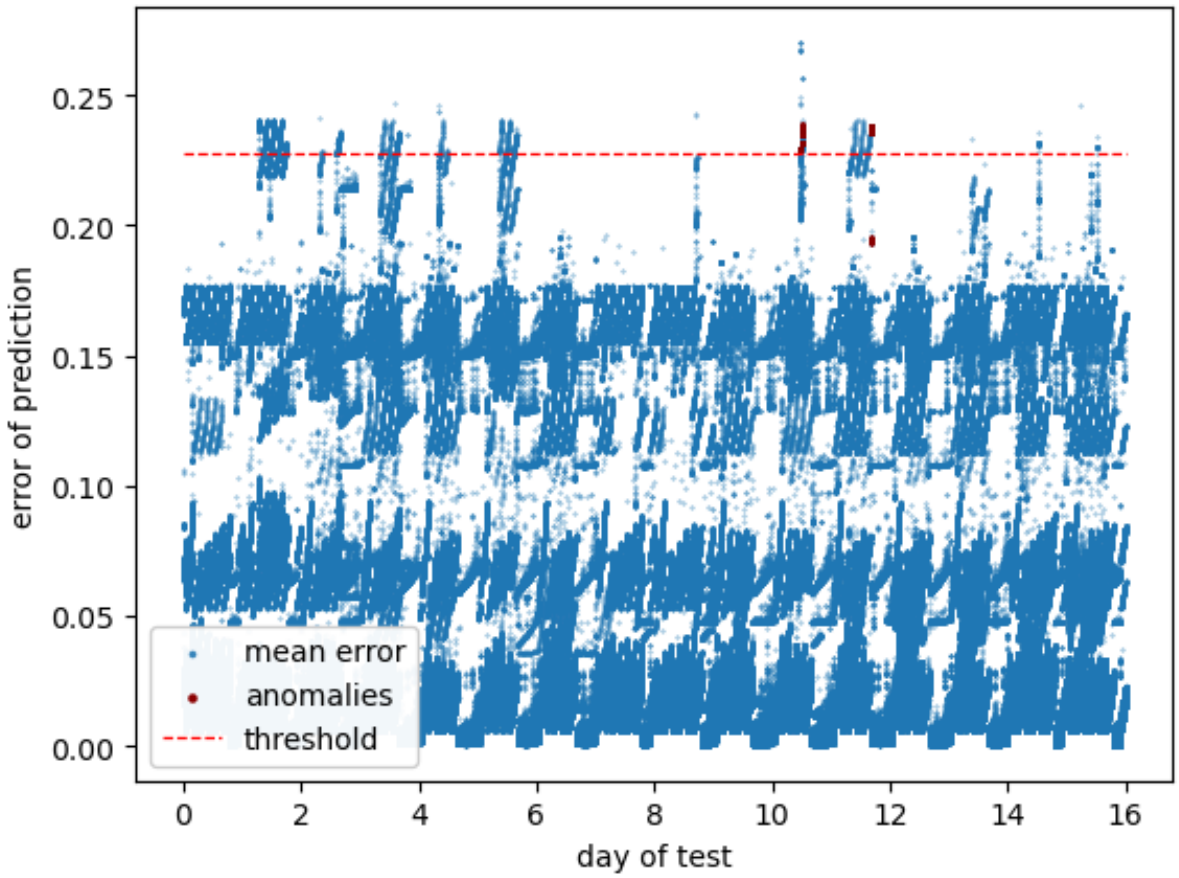

Figure 21: The error the prototype P_B produces in regenerating the test set. The dark red dots mark the anomalies created with the attacks described in section VI.*a*. The dashed red line represents the optimal threshold for this prototype.

After development, the resulting prototype is available in the ZHAW GitHub repository [34] as a Python application. Instructions for its use are provided in the corresponding README file [35]. The prototype can read in multiple log files simultaneously, detect anomalies as described earlier in this section, and output a single merged file of suspicious logs. Input files may look like table 3, while the corresponding output file looks like table 4. The merged output file consists of the detected anomalous logs, excluding the logs that were not detected as anomalies. In addition, it features a new column that indicates the error above the anomaly detection threshold.

SOURCE_TS	METHOD	RESPONSE_CODE	...
20240512063722000	GET	200	...
20240512064921000	POST	200	...

SOURCE_TS	METHOD	RESPONSE_CODE	...
20240512155515000	POST	404	...
20240512155517000	POST	200	...
20240513145114000	GET	404	...

Table 3: An example of two input CSV files.

SOURCE_TS	METHOD	RESPONSE_CODE	...	ERROR_ABOVE_THRESHOLD
20240512063722000	GET	200	...	0.0142
20240512064921000	POST	200	...	0.0024
20240513145114000	GET	404	...	0.0156

Table 4: An example of an output CSV file with the additional column of *ERROR_ABOVE_THRESHOLD* only consisting of anomalous detected logs.

VIII DISCUSSION

The prototype P_B shows promising performance values as shown in table 2, with 99.85% recall, 94.1% precision and an F1-score of 96.89%. This result is consistent with similar work, achieving a recall of about 96% to 99%, precision of about 93% to 98%, and F1-score of about 94% to 98%, see [15, 36, 37]. The comparison is not entirely accurate, as there are significant discrepancies in the data sets used for training and evaluation. The manual preprocessing does not allow training on different data sets without Further investigation is required, evaluating the prototype on bigger test sets. The attacks described in section VI.*a* both generate many logs within short times, which might result in misleading scores. On the other hand, the false positives are anomalous, even though they are harmless. These harmless anomalies may arise partly from application functionalities that are rarely used. This distinction needs further investigation. Also, another test set should be used to see the performance after defining the threshold on the first test set. The way the threshold is optimized shown in figs. 15 to 17 only optimizes it for this specific test set. To better test the prototype, more work should go into creating different anomalous logs under controlled circumstances as well. Also, further optimization can be conducted in preprocessing. Several features contain redundant information and could be condensed with techniques like binary representation instead of one-hot-encoding. The bucketing of the attribute *'SIZE'* as described in section V.*a.8* currently does not overlap, resulting in the same values for the size 0 and for example 20000. Additionally, the weighting of the different attributes to the mean absolute error should be further investigated as well. Moreover, the loss function itself can be reconsidered. An autoencoder with less trainable parameters could benefit from the mean squared error, with less risk of a loss explosion as mentioned in section IV.*c*.

To improve the reliability of the results found, the training data set should be rechecked with more knowledge about the application. Some functionalities of the application *Janereporting* are used very rarely, and therefore seem anomalous to anyone without expertise, although they are legitimate. Even though being used rarely, such an action can lead to several log entries over the course of a few seconds. Additionally, collecting more precise timestamps could improve the prototype, as the resolution in seconds does not always allow for correct ordering. Similarly, a strict userID or effective IP-addresses would enable the model to detect more types of attacks. Other fields with valuable information would be geolocation or the system's timezone.

This work offers an insight into the exact preprocessing of web access log records into floating point values and binary representations. The prototype trained over the course of this work can be used in a program that can successfully be used to identify anomalies.

IX Conclusion

This work developed and evaluated a prototype utilizing an LSTM autoencoder to detect anomalies in web access logs. In business environments, many applications generate a considerable amount of logs per day, where anomalies can indicate potential attacks. This amount of logs exceeds the practical limits of manual analysis. The prototype assists in identifying genuine anomalies, while keeping the workload of manual verification within manageable limits.

The reliability of the prototype is highlighted by the high recall and precision values it achieves on the test data, surpassing many models from similar projects. The tests show, that the prototype is able to detect attacks, even though they were never included in the training. This capability is crucial for enhancing the security of web applications. The verbose manual preprocessing ensures a robust foundation for anomaly detection, particularly for the application *Janereporting*, making early identification of attacks possible and allowing for prompt countermeasures.

Future work could build on the detailed feature selection and preprocessing methods presented. Expanding the dataset to include logs from diverse environments could improve the model's ability to generalize. Further optimizations in preprocessing, such as using binary representations instead of one-hot encoding and improving the bucketing of attributes like 'SIZE', could also enhance the model's performance. Revisiting the loss function and incorporating more precise timestamps, user identifiers, and geolocation data could provide additional improvements, making the prototype more robust and accurate in detecting anomalies.

References

[1] Next Stride AG, "Jane," accessed 2024-06-07. [Online]. Available: https://nextstride.com/de/jane/

[2] Y. Li and Q. Liu, "A comprehensive review study of cyber-attacks and cyber security; emerging trends and recent developments," *Energy Reports*, vol. 7, pp. 8176–8186, 2021.

[3] X. Zhang, M. Xu, G. Da, and P. Zhao, "Ensuring confidentiality and availability of sensitive data over a network system under cyber threats," *Reliability Engineering & System Safety*, vol. 214, p. 107697, 2021.

[4] Gatling, "Gatling (3.11.2)," accessed 2024-05-12. [Online]. Available: https://github.com/gatling/gatling

[5] Gordon Lyon, "nmap," accessed 2024-06-06. [Online]. Available: https://nmap.org

[6] Gobuster, "Gobuster (3.6.0)," accessed 2024-05-12. [Online]. Available: https://github.com/OJ/gobuster

[7] A. S. B. Singh, Y. Yusof, and Y. Nathan, "Eagle: Gui-based penetration testing tool for scanning and enumeration," in *2021 14th International Conference on Developments in eSystems Engineering (DeSE)*. IEEE, 2021, pp. 97–101.

[8] Next Stride AG, "Next Stride AG," 2024, accessed 2024-06-07. [Online]. Available: https://nextstride.com/de/

[9] M. Cieliebak, "Machine learning and data mining - neural networks – part 1," University Lecture, PDF, 2022.

[10] ——, "Machine learning and data mining - neural networks – part 2," University Lecture, PDF, 2022.

[11] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015, vol. 25.

[12] P. Kose, "Dawn of neural networks — explainable ai visualization (part 5)," 2022, accessed 2024-05-29. [Online]. Available: https://medium.com/deepviz/explainable-ai-and-visual-interpretability-dawn-of-neural-networks-part-5-b302e7d85650

[13] K. Gurney, *An introduction to neural networks*. CRC press, 2018.

[14] S. Flores. (2019) Variational autoencoders are beautiful. Accessed 2024-05-29. [Online]. Available: https://www.compthree.com/blog/autoencoder/

[15] M. Catillo, A. Pecchia, and U. Villano, "Autolog: Anomaly detection by deep autoencoding of system logs," *Expert Systems with Applications*, vol. 191, p. 116263, 2022.

[16] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[18] B. Lindemann, B. Maschler, N. Sahlab, and M. Weyrich, "A survey on anomaly detection for technical systems using lstm networks," *Computers in Industry*, vol. 131, p. 103498, 2021.

[19] aishwarya.27, "Introduction to recurrent neural network," 2023, accessed 2024-05-30. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-recurrent-neural-network/

[20] D. Kwon, K. Natarajan, S. C. Suh, H. Kim, and J. Kim, "An empirical study on network anomaly detection using convolutional neural networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1595–1598.

[21] V.-H. Le and H. Zhang, "Log-based anomaly detection without log parsing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 492–504.

[22] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 207–218.

[23] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*. IEEE, 2018, pp. 151–158.

[24] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 1285–1298.

[25] A. B. Nassif, M. A. Talib, Q. Nasir, and F. M. Dakalbab, "Machine learning for anomaly detection: A systematic review," *Ieee Access*, vol. 9, pp. 78 658–78 700, 2021.

[26] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer networks*, vol. 51, no. 12, 2007-8.

[27] Tensorflow, "tf.keras.layers.lstm," 2024, accessed 2024-06-05. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM#:~:text=Based%20on%20available,the%20outermost%20context.

[28] Numpy, "Numpy (1.23.4)," accessed 2024-06-06. [Online]. Available: https://numpy.org/

[29] T. pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020, accessed 2024-06-06. [Online]. Available: https://zenodo.org/records/7037953

[30] F. Chollet *et al.*, "Keras (2.6.0)," accessed 2024-06-06. [Online]. Available: https://keras.io

[31] Tensorflow, "Tensorflow (2.6.0)," accessed 2024-06-06. [Online]. Available: https://doi.org/10.5281/zenodo.5181671

[32] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[33] Tensorflow, "tf.data.dataset," 2024, accessed 2024-06-06. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#shift

[34] M. Koch, O. Wiedler, "Anomaly detection model prototype," accessed 2024-06-07. [Online]. Available: https://github.zhaw.ch/student-theses-traa/BA-log-analyzis-Wiedler-Koch

[35] ——, "Readme for anomaly detection model prototype," accessed 2024-06-07. [Online]. Available: https://github.zhaw.ch/student-theses-traa/BA-log-analyzis-Wiedler-Koch/blob/main/README.md

[36] H. Zenati, M. Romain, C.-S. Foo, B. Lecouat, and V. Chandrasekhar, "Adversarially learned anomaly detection," in *2018 IEEE International conference on data mining (ICDM)*. IEEE, 2018, pp. 727–736.

[37] J. Wang, Y. Tang, S. He, C. Zhao, P. K. Sharma, O. Alfarraj, and A. Tolba, "Logevent2vec: Logevent-to-vector based anomaly detection for large-scale logs in internet of things," *Sensors.*, vol. 20, no. 9, 2020-5-26.