Federated learning for log-based anomaly detection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Embedded Systems

by

Patrick Himler, BSc
Registration Number 0925574

to the Faculty of Electrical Engineering and Information Technology

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.-Ing. Tanja Zseby
Assistance: Dr.Dr. Florian Skopik
Dr. Max Landauer
Dr. Markus Wurzenberger

Vienna, 31st October, 2022

Erklärung zur Verfassung der Arbeit

Patrick Himler, BSc

Hiermit erkläre ich, dass die vorliegende Arbeit gemäß dem Code of Conduct – Regeln zur Sicherung guter wissenschaftlicher Praxis (in der aktuellen Fassung des jeweiligen Mitteilungsblattes der TU Wien), insbesondere ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel, angefertigt wurde. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, 31. Oktober 2022

Kurzfassung

Die Erkennung von Anomalien ist ein sehr wichtiger Bereich, um unerwünschtes Verhalten und Angriffe auf Computersysteme zuverlässig zu erkennen. Für eine präzise Anomaly Detection (AD) können Logdaten verwendet werden, die eine bedeutende Quelle für Informationen zu überwachten Systemen darstellen. Angesichts der schieren Menge an Logdaten, die heute zur Verfügung stehen, wird Machine Learning (ML) und dessen Weiterentwicklung Deep Learning (DL) seit Jahren zur Erstellung von Modellen für die AD eingesetzt. Insbesondere bei der Verarbeitung von komplexen Logdaten ist DL oft in der Lage eine bessere Leistung als ML zu erreichen. Ähnlich wie bei ML können DL Modelle unterteilt werden in supervised, unsupervised und semi-supervised Ansätze. Semi-supervised Ansätze trainieren ihre Modelle nur mit anomalie-freien Logdaten. Durch die große Menge an Logdaten entstehen Probleme bei der Übertragung an eine zentrale Stelle, an der Modellberechnungen stattfinden können. Federated Learning (FL) versucht dieses Problem zu überwinden, indem lokale Modelle gleichzeitig auf Endgeräten gelernt und Verzerrungen, die aufgrund mangelnder Heterogenität der Trainingsdaten auftreten können, durch den Austausch von Modellparametern einzudämmen, um schließlich zu einem konvergierenden globalen Modell zu gelangen. Die lokale Betrachtung der Logdaten trägt dem Datenschutz und rechtlichen Bedenken Rechnung. Dies könnte eine stärkere Koordinierung und Zusammenarbeit zwischen Forschern und Cybersecurity-Unternehmen usw. in Zukunft möglich machen. Derzeit gibt es nur wenige wissenschaftliche Veröffentlichungen über logdaten-basierte AD, die FL verwenden. Es ist notwendig zu untersuchen, ob der Einsatz von FL für AD ein praktisch einsetzbares Anwendungsgebiet ist. Die Grundlage für diese Masterarbeit ist ein zentralisierter Ansatz unter Verwendung eines Autoencoder (AE) und eines Long Short-Term Memory (LSTM) Modells für AD. Das AE Modell wird von Grund auf implementiert und für das LSTM Modell wurde eine moderne Open Source Implementierung namens LogDeep angepasst. Nach der Validierung der Ergebnisse mit anderen wissenschaftlichen Publikationen, übertragen wir die Modelle in eine FL Umgebung. Für die Auswertung verwenden wir einen Hadoop Distributed File System (HDFS) Datensatz, der in der aktuellen Forschung weit verbreitet ist, und einen Audit Datensatz, der vom Austrian Institute of Technology (AIT) zur Verfügung gestellt wird. Die ersten Ergebnisse zeigen, dass das AE Modell nicht für den untersuchten Audit-Datensatz geeignet ist. Die Ergebnisse für das LSTM Modell und den HDFS Datensatz zeigen, dass FL fast die gleichen Ergebnisse in Bezug auf die Metriken Accuracy, Precision, Recall und F1-Score, liefert wie ein zentraler Ansatz. Wobei wir im

Zuge der Implementierung von LogDeep auf einen Vorverarbeitungschritt gestoßen sind ohne den die guten Ergebnisse aus der Literatur nicht repliziert werden konnte. Der Audit Datensatz war aufgrund der inkludierten Labels nicht uneingeschränkt für die gewählten Modelle nutzbar. Die Implementierung von FL bietet den Vorteil, dass die Modelle trotz einer ungleichen Verteilung der Logdaten konvergieren. Außerdem können die Ergebnisse durch die Variation einzelner LSTM Modellparameter erheblich verbessert werden. Dennoch können durch Einschränkungen bei den verwendeten Datensätzen keine generellen Aussagen getroffen werden. Es ist weitere wissenschaftliche Forschung notwendig, um die FL Ansätze zu optimieren und den Bedarf an Rechenressourcen zu reduzieren.

Abstract

Anomaly Detection (AD) is a very important area to reliably detect unwanted behavior and attacks on computer systems. Log data is a rich source of information about systems under investigation and thus provides a suitable input for accurate AD. With the sheer amount of log data available today, Machine Learning (ML) and its further development Deep Learning (DL) is used for years to create models for AD. Especially when processing complex log data, DL is often able to achieve better performance than ML. Similar to ML, DL models can be divided into supervised, unsupervised and semi-supervised approaches. Semi-supervised approaches train their models only with benign log data. With the large quantity of log data, issues arise with the transfer of log data to a central entity where model computation can be done. Federated Learning (FL) tries to overcome this problem, by learning local models simultaneously on edge devices and overcome biases due to a lack of heterogeneity in training data through exchange of model parameters and finally arrive at a converging global model. Processing log data locally takes privacy and legal concerns into account and this could make more coordination and collaboration between researchers, cyber security companies, etc., feasible in the future. Currently, there are only few scientific publications on log-based AD which use FL. It is necessary to investigate whether the implementation of FL for AD is a practical field of application. The basis of this master thesis is a centralized approach using an Autoencoder (AE) and a Long Short-Term Memory (LSTM) model for AD. The AE model is implemented from scratch and for the LSTM model a state of the art open source implementation called LogDeep is adapted. After validating the results with other scientific publications, we transfer the models into a FL environment. For the evaluation, we use a Hadoop Distributed File System (HDFS) data set, which is well studied in current research, and an Audit data set provided by the Austrian Institute of Technology (AIT). The initial results show that AE is not suitable for the Audit data set under investigation. The results for a LSTM model show that FL yields almost the same results in terms of the metrics Accuracy, Precision, Recall, and F1-Score as a centralized approach. In the course of implementing LogDeep, we encountered a preprocessing step without which the good results from the literature could not be replicated. The Audit data set was not fully usable for the selected models due to the included labels. Implementing FL gives the advantage of converging models despite heterogeneous distribution of log data. Furthermore, by varying individual LSTM model parameters, the results can be greatly improved. However, due to limitations in the data sets used, no general statements can

be made. Further scientific research will be necessary to optimize FL approaches and reduce computational resource requirements.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Anomaly Detection (AD) has been an actively researched field for decades in various domains. Abnormalities, deviants, or outliers from normal expected behavior are referred to as anomalies and the goal of anomaly detection is to find those instances as efficiently and quickly as possible [CC19]. The application areas range from cyber security, health applications to risk management and many more. In this master thesis, we investigate the detection of anomalies in the cyber security domain using log data analysis.

In recent years, cyber attacks have become more sophisticated and the attack surface has tremendously increased because of nowadays complex computer systems and networks. The variety of cyber attacks is broad and ranges from obtaining unauthorized access into systems and data breaches to the destruction of critical infrastructures. This causes a slow but steady rise of awareness in non-security companies for security countermeasures. AD is widely used in Intrusion Detection Systems (IDS) to automatically detect and classify intrusions, attacks, or violations of security policies in infrastructures at network-level and host-level [VAS$^+$19].

Conventional signature-based IDS, using patterns of already known attacks and behavior have become insufficient and the need for more flexibility and adaptability is immanent. As a result, we see a shift towards anomaly based IDS, which use various data sources like textual log data and network traffic data to reliably discover deviations from a desired system's behavior. This approach supports reaching the goals of minimal maintenance, human interaction, and delay in response time [WSSF18].

Earlier work shows a development of applying Machine Learning (ML) in AD [CBK09]. Especially Deep Learning (DL) as a branch of ML, whose performance is remarkable has become a research hot spot [LOSW22]. DL can discover the essential differences between normal and abnormal data with high accuracy [LL19]. This strengthens monitoring systems, by creating a single model for benign behavior and threat detection. DL can support security experts to react quickly and effectively to known and unknown attacks and to gain a broad overview of the threat landscape. Therefore, security experts can expand their arsenal of security countermeasures and combine tools to keep up with the accelerating pace of cyber threats [PRT+18]. But it soon became apparent that DL for AD approaches also incorporate disadvantages, such as that the massive amount of data generated by edge devices must be communicated to a central node, for example, a data center, where the model is generated and trained. This introduces a single-point of failure. Those approaches are also often plagued by a delay in response time, where a fast analysis would be required. Furthermore, the transfer of data over the internet to a central node itself raises privacy concerns [RTTM20].

One promising solution to overcome those disadvantages is a distributed paradigm called Federated Learning (FL). In FL, training of DL models for AD is solved by a federation of participating devices, often referred to as clients, coordinated by a central server. This decentralized technique leaves the data distributed on the clients. With this data, each client trains a local DL model and sends only updates of the local models to a central server. The central server conducts the training of the global DL model by aggregating locally-computed updates and manages the communication of the global model back to the clients with the overall goal of a fast converging global DL model for AD [MMR+17]. This aggregation maximizes the usage of limited information of each client because every client can benefit from the updated global model [NMM+19].

With implementing FL for AD, we want to find out if it is possible to overcome limitations of centralized approaches. For example, achieving better adaptability of local DL models on each client without retraining models from scratch when the observed system is in a rapidly changing environment. It also appears fruitful to utilize log data for training DL models. Log data give a more detailed look for behavioural understanding of a system than network traffic data and is useful for troubleshooting. Contrarily to many available state of the art AD systems, which process network traffic data, a paradigm shift towards log data should be considered [WSSF18]. There are currently not many approaches in scientific research, combining DL methods for AD utilizing Audit log data and FL. Therefore, this master thesis aims to close this research gap further.

1.2 Research Questions

The thesis aims to explore two open source data sets (Hadoop Distributed File System (HDFS) [XHF+09] and Audit [LSW+21]) and to compare two DL algorithms, namely Autoencoder (AE) and Long Short-Term Memory (LSTM), with respect to their capability detecting anomalies. Also their disadvantages and limitations will be identified. An already existing centralized open source implementation of DL models for AD that has been applied to HDFS is adapted for the Audit data set. To test whether it is worthwhile implementing FL for AD, we conduct comparable simulation experiments of FL- and centralized AD. A multitude of research questions arise when trying to integrate FL models into AD. Those which are addressed in this thesis are the following:

- R1: Which DL algorithm, AE or LSTM, is more suitable with regard to Accuracy, Precision, Recall and F1-Score for the investigated data sets in the domain of AD?
- R2: Can a FL AD implementation reach the same Recall as a centralized DL AD for a well-studied HDFS and an Audit data set?
- R3: How fast do DL models for AD converge with regard to the Recall when FL is used in comparison to a centralized DL AD approach?
- R4: What are the open challenges for integrating FL for AD?

In this master thesis, we consider suitability using metrics that are commonly used for AD. Namely, the metrics are Accuracy, Precision, Recall, and F1-Score. Those are described in more detail in Sect. 4.8. For convergence we examine whether a FL implementation achieves the same Recall value as a central implementation, and if so, within what period of time.

1.3 Approach

First, we choose two open source data sets containing log data. The HDFS [XHF+09] data set is widely used in research for evaluating DL models in the AD domain and the Audit [LSW+21] data set is provided by the Austrian Institute of Technology (AIT). For the AE algorithm we build a detector from scratch and for the LSTM detector we use an open source implementation of DeepLog, which is a state of the art DL approach for AD. We verify published metrics for the HDFS data set with LogDeep, which is an open source re-implementation of DeepLog and uses LSTM as DL model, and adapt this re-implementation with the capability to process Audit data sets. By tuning DL model parameters, we show by which degree metric values change. With a working centralized foundation we proceed our research and implement a FL approach for LogDeep, in order to compare the FL approach with the centralized solution.

1.4 Contribution

The main contributions of this work are:

- In depth analysis of the Audit data set and comparison with the HDFS data set.
- Stand-alone AE implementation for Audit data set.
- Analysis of existing open source implementation of LSTM models for AD and selection of the most suitable for aforementioned data sets. It turned out that LogDeep [DA20] is best suited for the two data sets because of its documentation and ease of adaptation.

- Experiments with LogDeep utilizing the HDFS data set to confirm results from [DLZS17] and provide a basis for comparison with the FL implementation. The results could be reproduced, but shortcomings in the implementation based on simplified assumptions were identified.
- Adaption of LogDeep for Audit data set.
- Experiments with LogDeep for Audit data set to check if similar results as with HDFS data set can be achieved.
- Implementation of LogDeep in flower framework and experiments with evenly and unevenly split HDFS and Audit data sets.

1.5 Structure

In Chapt. 2 we provide background information about how ML, DL and log data are used in the field of AD, as well as a an introduction of two examined DL algorithms, which would be AE and LSTM. Furthermore, we explain FL and its basic features. The current state of the art research of DL and FL in the AD domain will be presented in Chapt. 3. A description of the data sets, data preprocessing, implementation approaches and used metrics can be found in Chapt. 4. To make this work reproducible, a detailed description of all experiments, which were carried out, can be found in Chapt. 5. In Chapt. 6 we interpret the results of the experiments. Finally, in Chapt. 7 and Chapt. 8, we will go into detailed review of the right and wrong assumptions we made at the start of this master thesis and also point out directions and considerations, which should be investigated further in future research.

1.6 Support

This work was partly funded by the EU project PANDORA (SI2.835928).

CHAPTER 2

Background

In this section we provide an overview of general concepts which act as foundation of this master thesis, namely: AD, DL, FL and log data.

2.1 Anomaly detection

AD describes the procedure to find instances in data sets that deviate from the expected behavior. Those deviations are often called anomalies or outliers. Anomalies can be caused by various things such as malicious activities and also wrong usage of a system. Many of the AD systems in use are tailored to one specific application [CBK09].

The main advantage over conventional signature-based approaches lies in increased flexibility to detect novel and previously unknown attacks. It is also necessary to consider an automatic analysis due to the huge amount of data. Some anomaly based detection approaches apply machine learning to first learn a system's normal behavior and then use the divergence to detect anomalies [BB15][LZXS07][XHF$^+$09].

Basically, anomalies can be classified into three categories, which would be:

- Point Anomalies: Those anomalies usually stand out from the rest of the data set because they deviate significantly. The vast majority of research focused on this kind of anomalies, because these are the simplest type of anomalies to detect [CBK09]. In this master thesis, we try to detect point anomalies with an AE approach.

- Contextual Anomalies: If parts of the data set are anomalies only in a certain context but not individually, then we speak of contextual anomalies. To detect contextual anomalies, we need to consider both contextual attributes, like passed time between data instances and behavioral attributes such as occurrences of specific data instances. Anomalies can be detected when we combine both, i.e. values of behavioral attributes in a specific context [SWJR07]. Most commonly explored are time-series data sets. The predestined DL algorithm here is LSTM [KMDH19].

- Group Anomalies: Individual data instances in isolation appear as normal but if observed collectively they can be detected as anomalies. Group anomalies can be seen as point anomalies which appear over and over again [CC19][CBK09]. Because it always depends on the data set which kind of anomalies can be investigated, we have chosen the two previous categories for this master thesis.

AD techniques can be classified based on the availability of labels. Labels indicate whether a respective data instance is benign or malicious. The three categories are [LOSW22][CBK09]:

- Supervised: A fully labeled training set containing both benign and anomalous data is required. A common approach is to build a predictive model for both data classes. Then unseen data instances are tested with the model to determine which class they belong to. At first glance, it is easy to create a model like this but it has two major disadvantages. First, usually data sets contain fewer anomalies than benign data which leads to an imbalanced class distribution. Second, it is not trivial and thus challenging to label the anomaly class.
- Semi-supervised: The prerequisite is that the training set only contains benign data. After training, the model is used to identify anomalies in test data. This technique assumes availability of benign training data sets which can be challenging in some application domains.
- Unsupervised: The system learns independently to distinguish between benign and anomalous data without the prerequisite of a fully labeled training set. This approach makes use of the intrinsic properties of data instances. In principle, it is often the case that data sets under investigation have fewer anomalies than benign data, which would speak for an unsupervised approach. The trained model should be robust to those few anomalies. If this assumption does not apply it will lead to a high false alarm rate. But what we see in recent research is that a semi-supervised approach is often more suitable, because it is often simpler to gather benign data then modeling anomalies.

It is worth noting that both semi-supervised and unsupervised can never match the performance of supervised AD, e.g., binary classifier. Nevertheless, we have chosen a semi-supervised approach for this master thesis, because it is a realistic scenario with regard to real world applications to first learn a model in an anomaly-free area with benign data and then switch to live operation.

2.2 Machine Learning

The main reason to use machine learning algorithms for AD is to create a baseline for a system under observation with minimal human interaction and keeping it up to date. Therefore, machine learning can be used on two fronts [SWL21]:

1. First, to learn which data and features are useful and relevant to detect anomalies.
2. Second, to create and maintain a model which can distinguish between normal and malicious behaviour and reveal hidden relations.

In the past, machine learning algorithms such as Decision Tree (DT), Support Vector Machine (SVM) and Principal Component Analysis (PCA) were mainly used for log-based AD. But it became clear that those algorithms are not good at analyzing concealed relationships of some features. Nowadays, DL algorithms are being used to overcome this limitation [YKD20].

2.3 Deep Learning

DL is a further development of classical ML which shows good performance and flexibility, especially when data sets become larger and the structure of data becomes more complex [CC19]. The complexity lies in the fact that anomalies often show clear abnormal characteristics in low dimensional space that are virtually not noticeable in higher dimensional space. Another reason why DL has become so important is the necessary shift to semi-supervised and unsupervised AD [PSCH22]. It simply does not correspond to reality to assume availability of completely labeled data sets. The choice of the DL architecture depends usually on the type of input data. A widespread assumption is that AE are preferred for image data sets and LSTM are good for processing sequential data [CC19].

2.3.1 Autoencoder

AE is one of the well studied DL algortihms which can be used for unsupervised and semi-supervised AD [ITM21]. An AE consists of an encoder and decoder. At first, input data gets compressed and mapped to a low-dimensional feature space by the encoder. As a second step, the decoder tries to recover the input data from the encoded data. Through a bottleneck network architecture an attempt is made to retain information that is necessary to reconstruct the input data. Finally, input data is compared with the output data and a reconstruction error is determined [YKD20][PSCH22]. Anomalies should be harder to reconstruct from compressed space than benign data which lead to a large reconstruction error [CC19]. This large reconstruction error can then be used as an indicator for anomalies. The mathematical expression for the encoder and decoder can be found in Eq. 2.1 and 2.2 [ITM21]. A basic structure of an AE can be seen in Fig. 2.1.

$$Encoder\ Stage: \mathbf{H} = G(\mathbf{x} \cdot \alpha + \mathbf{b}) \tag{2.1}$$

$$Decoder\ Stage: \mathbf{y} = \mathbf{H} \cdot \beta \tag{2.2}$$

where:

$\mathbf{H}$ = Hidden-layer matrix

G = Activation function

$\mathbf{x}$ = Input

α = Input weight matrix

$\mathbf{b}$ = Bias vector

$\mathbf{y}$ = Ouput

β = Output weight matrix

β is trained so that an input is reconstructed as correctly as possible.

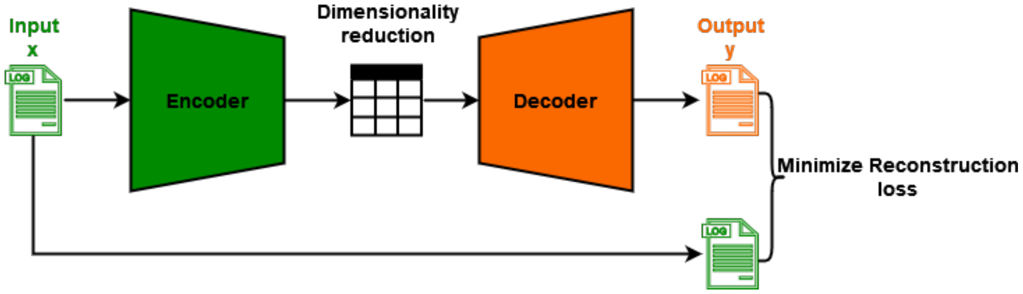


Figure 2.1: Basic design of an AE based on [ITM21].

After training we calculate the Root Mean Squared Error (RMSE) as shown in Eq. 2.3.

$$RMSE(w^i_{-1}) = \|w^i_{-1} - \widetilde{w}^i_{-1}\|^2 \tag{2.3}$$

where:

w^i_{-1} = Original data

$\widetilde{w}^i_{-1}$ = Reconstructed data

In conclusion, AE is a straightforward approach for many different types of data sets which is the biggest advantage. In the AD domain, AE can be used to detect point anomalies. But the learned model can be biased and therefore unsuitable for AD, if the training data set contains anomalies or infrequent regularities [PSCH22].

2.3.2 RNN & LSTM

A Recurrent Neural Network (RNN) is an artificial neural network which can learn sequential or temporal interrelationships. To achieve that, RNNs have the ability to memorize and store a previous output of a learning node and return it as an input to a subsequent learning node [DLZS17][YKD20]. In principle, an input layer represents features at one point in time t where the number of dimensions of this layer corresponds to the feature size of the input data [HXY15]. The output layer gives a probability function for predicted labels at time t. The equation for hidden state can be found in Eq. 2.4 and for the output in Eq. 2.5 [HXY15].

$$\mathbf{h}_t = f(\mathbf{Ux}(t) + \mathbf{Wh}(t-1)) \tag{2.4}$$

$$\mathbf{y}_t = g(\mathbf{Vh}(t)) \tag{2.5}$$

where:

$\mathbf{h}_t$ = Hidden state

$\mathbf{x}_t$ = Input

$\mathbf{y}_t$ = Output

$\mathbf{U}$,$\mathbf{W}$,$\mathbf{V}$ = Connection weights

f, g = Activation functions

The authors of [VSP17] claim that LSTM as a variant of RNNs algorithm is one of the most important approaches to represent long-range temporal dependencies in log sequences of arbitrary length. For AD, this means that we use past information to predict whether future events are benign or anomalous. This past information is especially important when we want to detect contextual and group anomalies. Unlike RNN, LSTM replaces hidden layers with special memory cells [HXY15]. The mathematical expressions for a basic LSTM cell can be found in Eq. 2.6,2.7,2.8,2.9,2.10 and 2.11 [SH21]. A basic structure of a LSTM cell can be seen in Fig. 2.2.

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{x}_t + \mathbf{R}_f\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.6}$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_g\mathbf{x}_t + \mathbf{R}_g\mathbf{h}_{t-1} + \mathbf{b}_g) \tag{2.7}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{x}_t + \mathbf{R}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.8}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \tag{2.9}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o x_t + \mathbf{R}_o\mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.10}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{2.11}$$

where:

$\mathbf{f}_t$ = Forget gate

$\mathbf{g}_t$ = Memory cell

$\mathbf{i}_t$ = Input gate

$\mathbf{o}_t$ = Output gate

$\mathbf{c}_t$ = Cell state

$\mathbf{h}_t$ = Hidden state

$\mathbf{W}$ = Input Weight matrices

$\mathbf{R}$ = Recurrent Weight matrices

$\mathbf{b}$ = Bias vectors

σ = Sigmoid activation function

$\odot$ = Pointwise product

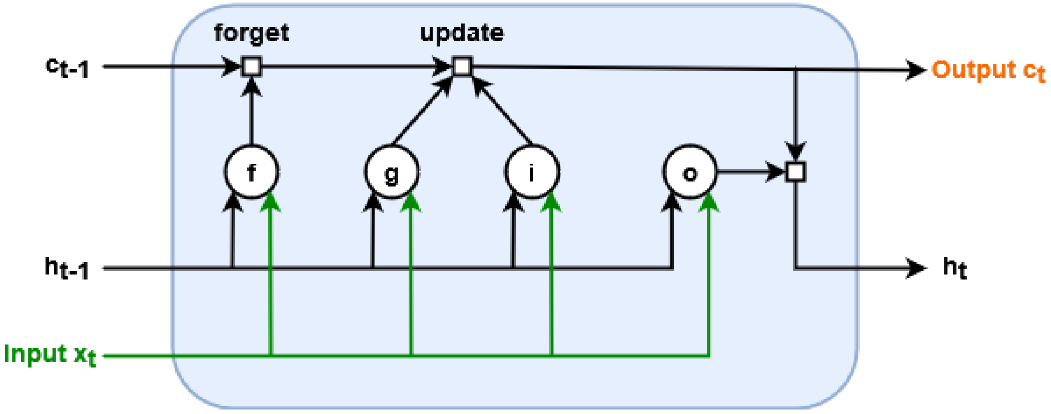


Figure 2.2: LSTM cell based on [SH21].

The LSTM model is composed of several LSTM cells. Each cell stores the state of the input. In addition, cells also receive the state of the preceding cell. This way, historical information is passed on. An exemplary structure of several interconnected LSTM cells can be seen in Fig. 2.3.

2.4 Federated Learning

Both ML and DL can nowadays benefit from the sheer volume of data. However, with this large quantity there arise issues with transferring this data to a central server. FL tries to avoid this problem by having the training data never leave local devices. A shared DL model is learned through locally aggregated updates. Because the DL model is learned on local devices, it also reacts better to environmental changes over time [ITM21]. The

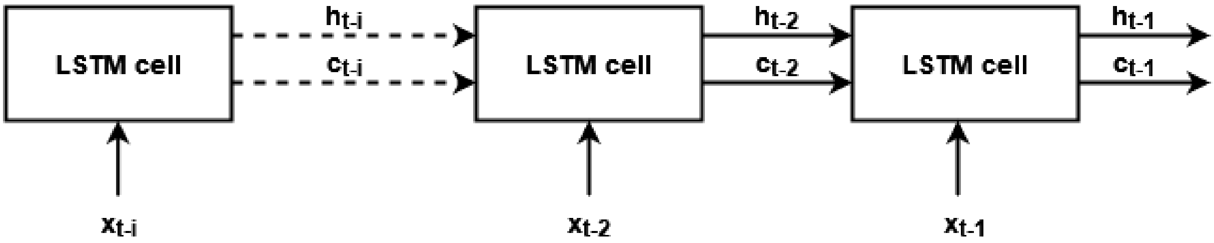


Figure 2.3: Interconnected LSTM cells based on [DLZS17].

localization of data also takes privacy and legal concerns into account. This could make it feasible in the future that there is even a secure data exchange between organizations that currently do not cooperate with each other. Coordination and collaboration will become very important in the future, especially in the cyber security domain. FL was first introduced in 2016, by researchers of Google [MMR+17]. The term FL origins from the fact that the model learning task is performed by a loose federation of devices called clients which are coordinated by a central server. Each client trains a local DL model only with a local training data set and sends updates for a global model to a central server. It must be mentioned that the central server that manages the training and distribution of the global model must be trusted. Another reason why it is interesting to deal with FL is to reduce communication costs. Nowadays, devices like smartphones have fast processors and Graphics Processing Units (GPUs). Assuming that local data sets are small, the described hardware can simply take over computation of DL models instead of many devices taking up bandwidth unnecessarily by sending data to a central server, thus reducing communication costs [MMR+17]. A final statement on the saving of computational resources is difficult, because it depends strongly on the other load of a client and which energy resource, e.g., battery, is available.

Basically, FL consists of the following steps [LFL20]:

1. All participating clients get a generic global model from a central server for local training.
2. Each client learns a local DL model for themselves with local data.
3. After the learning phase, clients send back their local parameter updates to the central server for aggregation.
4. The central server averages those updates and sends back an updated version of the global model.
5. Those processes are repeated until desired performance is achieved. An iteration of those processes is called a round.

The process described here is shown in Fig. 2.4.

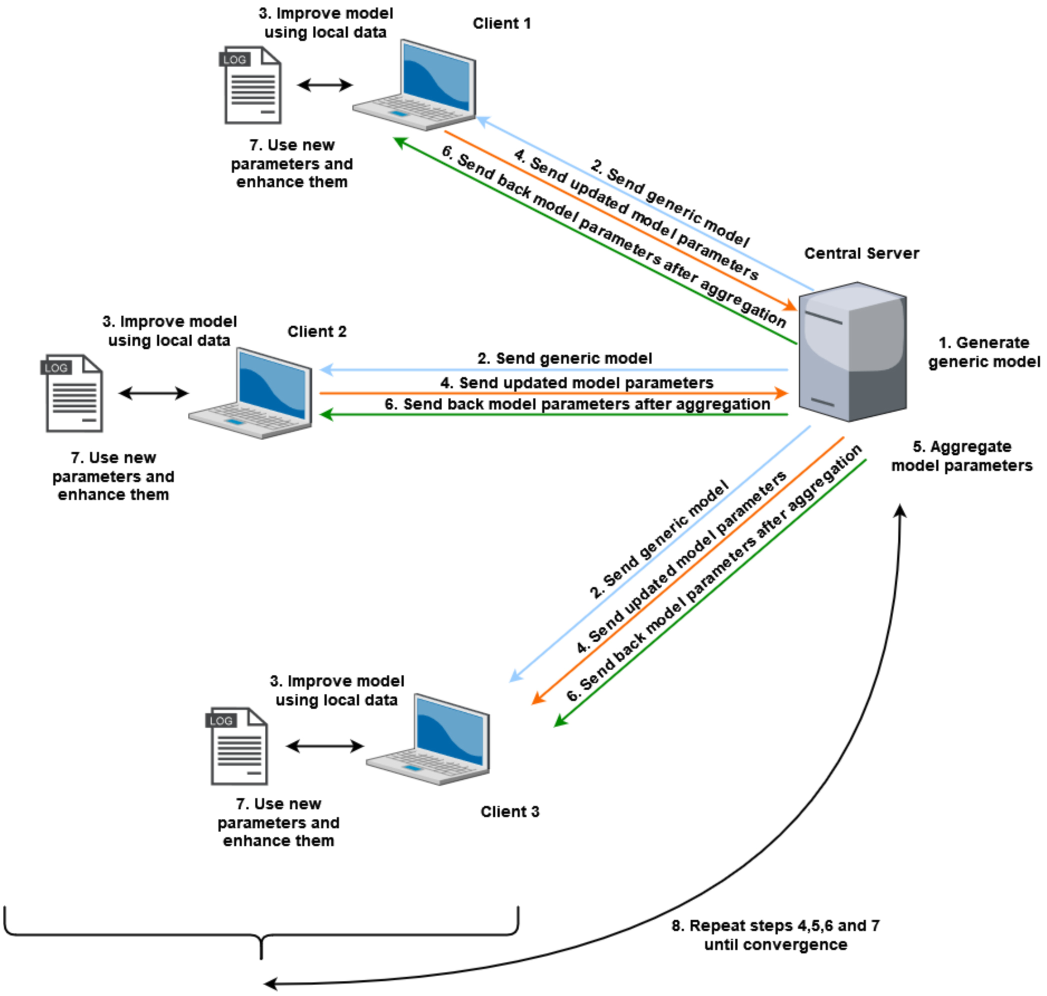

3. Improve model using local data
Client 1
LOG
7. Use new parameters and enhance them
2. Send generic model
4. Send updated model parameters
6. Send back model parameters after aggregation
Central Server
3. Improve model using local data
Client 2
LOG
2. Send generic model
4. Send updated model parameters
6. Send back model parameters after aggregation
1. Generate generic model
7. Use new parameters and enhance them
5. Aggregate model parameters
2. Send generic model
4. Send updated model parameters
6. Send back model parameters after aggregation
3. Improve model using local data
LOG
7. Use new parameters and enhance them
Client 3
8. Repeat steps 4,5,6 and 7 until convergence

Figure 2.4: Illustrative FL scenario based on [RTTM20].

2.4.1 Categorization of Federated Learning

The authors of [YLCT19] proposed the following categorization of FL approaches based on how data is distributed among participating clients in feature and sample space, where a sample is a single entry in a data set and a feature is a measurable value of such entry [Bis06].

- Horizontal FL: The feature space is the same but the sample space is different. An example would be two regional banks which of course have different customers (=samples) but the business transactions (=features) are very similar.

- Vertical FL: Unlike horizontal FL, it is the other way around, i.e. in both data sets there are identical samples that can be recognized by an identifier but have different features. For example, two different companies have the same customers (=samples) with the same residential address, but the companies provide different services (=features) to these customers.

- Federated transfer learning: Data sets on the clients differ in both sample and feature space. There is only a small overlap, e.g., in the feature space, and this allows conclusions to be drawn about the entire sample- and feature space. This approach is still at the beginning of a development but should be mentioned here for the sake of completeness.

In this master thesis, we use the horizontal approach for AD. The logs of the data sets are recorded centrally. They have the same feature space and are divided into parts for a number of clients for experiments. So each client on its own can make a local AD system and collaboratively learn from others simultaneously. Concretely, local biases that arise due to lack of heterogeneity in the training data can be overcome [LPBA22]. But existing FL approaches for AD are still missing a uniform structure and are far from complete, especially when utilizing log data.

2.5 Log data

Since the dawn of programming, logging was used to record program variable values, trace execution, report runtime statistics, and even print out full sentence messages designed to be read by a human. In that sense, log data is a valuable source to help debug system failures and perform root cause analysis. Naturally that is the reason why log data is used for AD. But, we have to keep in mind that it is not always sufficient to look at the presence, absence or frequency of a single type of log data to pinpoint a problem. Often the problem reveals itself only when we look at correlations or relative frequencies. The important information and interrelationships is buried in millions of log lines. So logs often indicate sources of problems, but have become to large to be examined manually. Automatic intrusion detection performed by ML and DL sounds promising [XHF+09].

Usually, log-based DL for AD consists of three steps: log parsing, feature extraction and AD. Log parsing allows us to turn unstructured log lines into structured data. During feature extraction, this structured data is transformed into numerical feature vectors, because DL processes only numerical representations of log lines. Finally, a DL algorithm is applied to classify between benign and malicious log lines [HZHL16].

The majority of scientific research in the field of AD does not focus on log data. A large number of scientific publications investigate AD based on network traffic. Network features, such as source and destination, ports, TCP flags, protocols, and packet lengths are examined [LPBA22].

CHAPTER 3

State of the art

There are currently many scientific publications on how DL can support AD. But as mentioned before, research publications dealing with log data, AD, DL and FL are rare. In principle, the majority of research in the field of DL and AD utilizes network traffic data sets and neglect log data sets. The reason for this is the simpler processing of numerical data sets where the parsing step is left out. The sheer variety makes log data processing not trivial. Also FL is still a relatively new research topic. Those are the main reasons that there are still only few publications in this field. Nevertheless, we would like to present a few research results worth mentioning in the individual domains in this chapter.

3.1 Deep Learning for anomaly detection

In this section we list three state of the art approaches that use DL for AD.

3.1.1 DeepLog

The authors of [DLZS17] have developed an approach called DeepLog that uses an LSTM as DL model that processes log lines as sequences. The model thereby learns log patterns during normal execution and detects anomalies when log patterns deviate from this learned model. Even though this approach was published in 2017, it still has great significance to this day. As a starting point of development, the authors describe that log entries in most cases have fixed patterns and also follow grammar rules. But they also state that it is still difficult to make a generalization about interesting features for different data sets. To evaluate their implementation the authors use the HDFS data set [XHF$^+$09] and the OpenStack data set [DLZS17]. The OpenStack data set contains administrative logs of virtual machine instances. DeepLog works in a semi-supervised way, therefore only anomaly-free sequences are used for training the LSTM model. First, the

Spell parser extracts so-called log keys (=constant part) and parameter values (=variable part) from each log line. This way two separate AD systems can be set up. First, DeepLog checks if the log key to be examined is a known one. In case it is, it can then check if parameter values indicate anomalies. First, the paper describes the process for log keys. In principle, one can assume that the number of different log keys is constant, no matter how log lines are recorded for a given data set. This assumption indicates that the authors do not take into account new log lines, that have a new structure. The log producing system forms a sequence of the process to be observed with a finite number of log keys. The sequence in which log keys occur can be used to detect so-called execution path anomalies. This type of anomalies correspond to the contextual anomalies explained above. The LSTM model uses a set of log keys with a fixed size, called window size, and uses the gathered knowledge to predict which log key should follow. After training the LSTM model, DeepLog outputs possible candidates that were predicted with their respective probabilities. In contrast to this, parameter values are used to find irregularities in log lines with the same log keys. A matrix is built up where each column corresponds to a log key and the corresponding parameter values are entered in the rows. For the evaluation of the matrix, an LSTM model can also be used. The individual parameter values are used as input in the order in which they occur and an attempt is made to generate a prediction for the following parameter value based on this existing history. The structure of the DeepLog architecture described here is shown in Fig. 3.1.

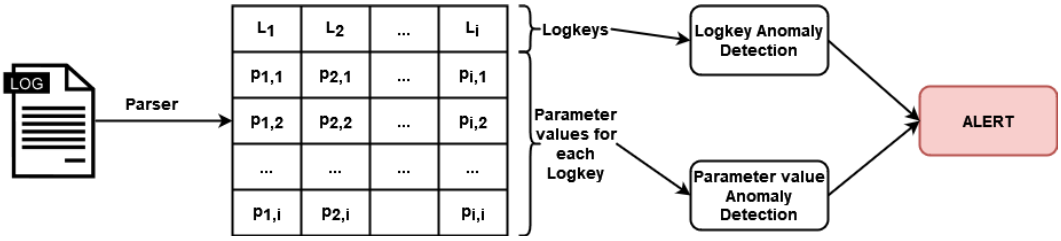


Figure 3.1: DeepLog architecture based on [DLZS17].

The two most important input parameters DeepLog needs are length of the window under consideration and number of top candidates for prediction. The choice of these parameters depends on the problem to be considered. For example, if we choose the window size too large we get a wide view in the past and a better overall picture, but we also have to expect performance losses resulting in long training time. The choice for the number of candidates results in a trade off between AD rate (=true positives) and false alarm rate (=false positives). For benchmarking the authors used PCA- and Invariant Mining (IM) approaches, as classical ML methods, to compare it with their DeepLog implementation. The breakdown of the HDFS dataset for evaluation can be seen in Tab. 3.1 and resulting metrics for DeepLog compared to approaches using PCA and IM are shown in Fig. 3.2.

Train (normal) sequences	Test normal sequences	Test abnormal sequences
4,855	553,366	15,200

Table 3.1: Splitting of HDFS data set used in [DLZS17].

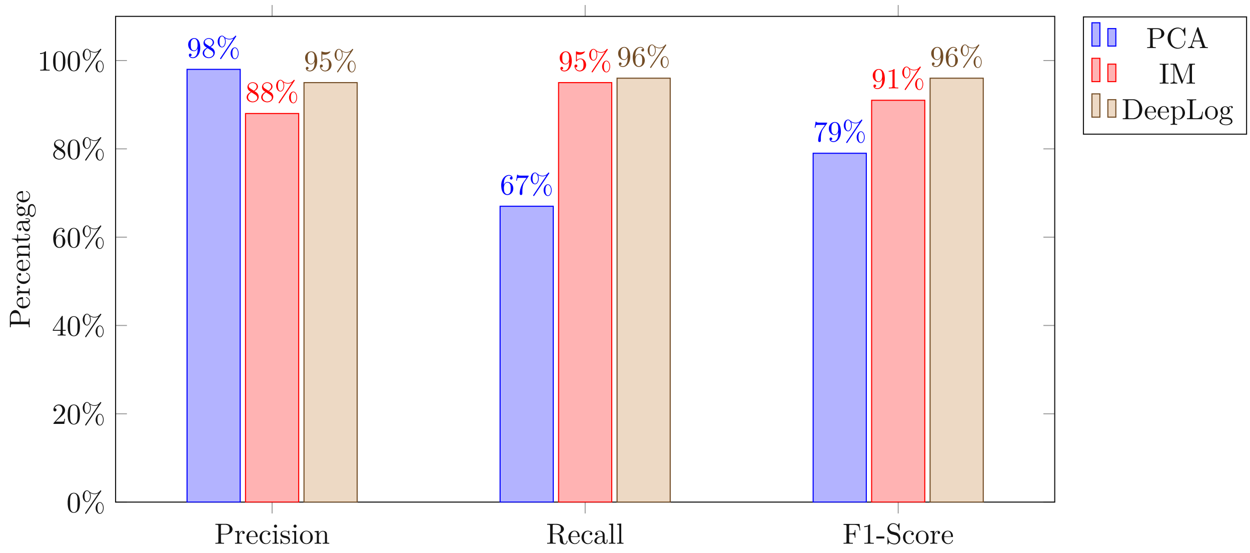


Figure 3.2: Evaluatution results of [DLZS17].

It is evident that DeepLog delivers the best results in comparison. This is proven by an F1-Score of 96%. A shortcoming of this publication is that the authors only briefly mention that they use the Keras and TensorFlow library to implement DeepLog, but did not go into more detail or published open source code. In summary, DeepLog was the first framework to detect contextual anomalies in log data using DL. Another reason for its relevance to date is the release of open source re-implementations by various developers [LOSW22].

3.1.2 LogAnomaly

LogAnomaly follows a similar approach as DeepLog. The authors of [MLZ^{+}19] claim that if we just look at log keys rigidly you get many false alarms, because log lines are much more complex. Therefore they analyze not only log keys but also semantics of the logs. For this they use a method they call template2vec. This method extracts semantics including synonyms and antonyms. They use LSTM as DL model to predict consecutive logs and HDFS as data set for evaluation. In addition, they also consider the Blue Gene/L (BGL) data set [OS07], which consists of logs from a supercomputer and was manually labeled. The authors designate LogAnomaly as an unsupervised approach, but they use labels of the data sets as groundtruth for evaluation. According to the experiments performed, LogAnomaly performs better than DeepLog based on the evaluation metrics for the datasets investigated. However, a described case study in this paper shows that DeepLog triggered an alarm faster than LogAnomaly in a single anomaly case scenario. Unfortunately, we were not able to inspect the authors' template2vec method more closely using code examples, but re-implementations are available.

3.1.3 LogRobust

In the paper [ZXL$^+$19], the authors draw attention to instability of log data. This is caused by the evolution of the logging process itself and of processing noise in log data. Evolution is based on constant development of software and associated changes in source code and logging statements. The introduction of noise already happens during data collection. In distributed systems, logs can be lost or duplicated during transfer to a central entity. But logs can also be misinterpreted during parsing. All this reduces the accuracy of an AD system. To counteract this, the authors of LogRobust rely on semantic vectors. Not unlike LogAnomaly, semantic properties of log lines are extracted. But in contrast to previously discussed approaches, LogRobust can also process new log lines if they are similar to known ones. To test this, slightly modified log lines are inserted in the HDFS data set to be examined. It is important to note that training of the model is still performed with unmodified log lines i.e. the original HDFS training data set. LogRobust is basically very similar to DeepLog and LogAnomaly. It also uses log sequences as input for a LSTM. First difference is that it is a supervised approach and uses 6000 benign log sequences and 6000 malicious log sequences which are chosen randomly from the HDFS data set to train the model. As expected due to the supervised approach metrics after training are better, for testing without modified log lines, than for the other two approaches shown above. The metrics for the test data set (562855 benign log lines and 10838 malicious log lines) result to: Precision=98% Recall=100% F1-Score=99%. Further Evaluation shows that LogRobust still achieves good metrics even with high injection rates of modified log lines in the test data. For example, the F1-Score decreases to only 89% at 20% injection rate compared to no injection. Again, the authors have not published the source code for their work, but re-implementations are available.

3.1.4 Deep Learning for Anomaly Detection in Log Data: A Survey

Only recently mid 2022, [LOSW22] published a survey of how DL is used in the area of AD with a focus on log data. The paper focuses on explaining aspects of different DL architectures and neglects a quantitative comparison between different approaches, published up to that point. At the beginning of the paper, the authors explain how they conducted the literature search and what their reasons were for including or rejecting certain papers in their survey. Their review identifies Convolutional Neural Networks (CNN), RNN (e.g., LSTM and Gated Recurrent Unit (GRU)), and AE as the most commonly used DL architectures in this scientific domain. Furthermore, the authors confirm the observation, that some publications claim unsupervised approaches when in fact they are semi-supervised approaches. A clear definition would be desirable in this area in any case. They confirm the assumption that for a large part of the evaluations only four data sets are used, namely: HDFS [XHF+09],BGL [OS07],Thunderbird [OS07] and OpenStack [DLZS17]. Based on number of citations but also how often it was used as a benchmark, the authors confirm the enormous importance of [DLZS17].

3.1.5 Selection of open source state of the art implementations

As discussed above, it is unfortunately often the case that authors of scientific papers only rarely publish their source code and thus results cannot be verified. Nevertheless, some developers make the effort and re-implement presented architectures and publish them. Tab. 3.2 shows a selection of such open source implementations.

3.2 Federated learning for anomaly detection

The authors of [GWZ+21] published one of the few papers that deal with AD, FL and log data. A central point of this research is that it deals with the issue of transmission of updates between clients and servers. It has been found that attackers can intercept communicated gradient updates and thus violate the privacy protection of local data. Therefore it is advantageous if this communication is encrypted. However, encryption always leads to a computation overhead which is proportional to the size of gradient updates. The authors therefore propose a lightweight FL method for AD called FLOGCNN. Lightweight is obtained by reducing the model parameters. As a result the AD model uses one-dimensional convolution with very few parameters. For the evaluation of the presented approach the authors use the HDFS data set and Thunderbird data set [OS07]. Thunderbird is an publically available data set of logs collected from a Thunderbird supercomputer system at Sandia National Labs in Albuquerque. The data set distinguishes between alert and non-alert messages. This is a labeled data set which can also be used for supervised approaches. For FLOGCNN two roles are defined. A server aggregates gradient updates and distributed participants, which are here called log owner. The server randomly selects a number of log owners and sends initial model parameters to them. Log owners learn independently and send updates back to the server. For evaluation purposes, the authors compare their implementation with a centralized LogRobust implementation. For both implementations the source code is not publicly available. The author's implementation achieves slightly worse results for the metrics than LogRobust. As a positive result the authors state a lower training time. A limitation of this approach is that it only shows how to deal with HDFS and Thunderbird logs. These are data sets with few features and little variation. Therefore, feature reduction is easy. It is questionable whether the presented architecture will perform as well, when other log data sets are used.

Paper	Year	Model	Dataset	Github
[DLZS17]	2017	LSTM	HDFS	github.com/donglee-afar/logdeep
[vEAS+22]	2022	LSTM/GRU	HDFS	github.com/Thijsvanede/DeepLog
[CLG+21]	2021	LSTM/AE	HDFS	github.com/logpai/deep-loglizer
[MLZ+19]	2019	CNN	HDFS	github.com/WraySmith/log-anomaly
[GYW21]	2021	LSTM	Thunderbird/BGL/HDFS	github.com/HelenGuohx/logbert

Table 3.2: A selection of open source state of the art implementations.

CHAPTER 4

Methodology

In this chapter, we provide an introduction to the data sets used. The individual features of the audit data set we use are discussed and reasons are given as to which features are selected. The preprocessing of log lines is an important step to prepare data as input for DL networks. DeepLog which was described in the previous chapter is adapted for this master thesis. We decided to first implement DeepLog in a centralized way and verify it with existing results. This is followed by a decentralized FL implementation and here we also explain our chosen FL aggregation algorithm. The FL implementation is based on the flower framework, the basics of which are briefly explained. At the end of this chapter we will discuss metrics that are used to make our implementations comparable to others.

4.1 Data sets

Many AD systems are tailored to a specific environment to detect cyber attacks. Benchmarking AD for real-world applications is nontrivial. To make valuable statements about AD they require data sets collected in realistic system environments [LSW+21]. Some data sets become standards for evaluation over time such as the HDFS [XHF+09] data set we used in this thesis. However, we should keep in mind that most data sets become outdated over time.

4.1.1 HDFS data set

The HDFS is a file system designed for storing large files, batch processing and to run on commodity hardware. Because of to the popularity of HDFS, it has been widely studied in recent years and is used in this thesis for providing reproducible comparison with state of the art anomaly based IDS. The data set was generated in a private cloud environment (Amazon's Elastic Compute Cloud) using benchmark workloads and is described in detail in [XHF+09]. It consists of 11.2 million system log entries. It was manually labeled by Hadoop domain experts, to identify anomalies. 2.9% of all system log entries were labeled as malicious by those experts. The raw HDFS logs are semi-structured and consist of a header- and a content part. The log data are sliced into sequences according to block_ID's. Then each trace associated with a specific block_ID is assigned a ground truth label: normal/anomaly. In this master thesis we use the HDFS data set to test our DeepLog implementation and to compare it with published results of [DLZS17].

It is interesting to mention that through the labeling work of Hadoop domain experts a bug has been uncovered, which was hidden in HDFS for a long time. The bug involves a relatively rare code path. This code path attempts to delete a replica of a block that no longer exists and could only be discovered because abnormal execution paths were analyzed. The Hadoop developers stated that this bug was hard to find, because it was no single error message and only shows when we look at a sequence of log lines. This incident shows an additional motivation to deal with abnormal execution paths [XHF+09].

4.1.2 Audit data set

This data set was provided by researchers of the AIT and is also publicly available. How to set up a testbed, to generate the Audit data set, is shown in [LSW+21]. They published four testbeds, where they log users accessing a webmail platform and online store in a time period of six days. Within this time period one multistep attack and one complex vulnerability exploit were launched. The exact procedure of the multistep attack and tools used are described in Fig. 4.1. The testbed is designed in such way that logs are collected by a Linux Audit daemon. Those logs are interesting, because they provide low-level syscall information and show accesses of files and paths on the host. Audit log lines include types such as SYSCALL-, CWD-, PATH- or PROCTITLE type. Each individual entry consists of at least one log line of type SYSCALL, but can also be extended by log lines of other types. The type of each log line is specified at the beginning of the log line in the type field. Type SYSCALL indicates that a log line was triggered by a system call to the kernel. Type CWD does not occur in the dataset provided by the AIT. Type PATH specifies all paths that a system call gets as arguments. Finally, Type PROCTITLE entry gives the complete command-line in hexadecimal notation. In this master thesis we concentrate only on the entries of type SYSCALL, because they contain all relevant information and the remaining type fields do not always occur. A detailed description of all individual features of a SYSCALL Audit log line is described in Tab. 4.1.

In principle, it appears to be challenging to analyze Audit log, because of their sheer quantity [SWL21]. As mentioned before, the most difficult task related to data set creation is to correctly label log lines. For this data set the authors used time-based labels and line-based labels. Time-based labeling uses time stamps, which are parsed with each log line. An attack is labeled malicious, when the log line occurrence lies within the time frame of the attack stage. Problems with this approach arise, if normal log lines interleave between log lines of an attack. So the second approach called line-based labeling tries to overcome this shortcoming by labeling attack steps in an idle system. It turns out that most attacks generate ordered log sequences (e.g., webshell upload) or repeating log lines (e.g., scans). In this thesis, we look at an Audit data set with line-based labels.

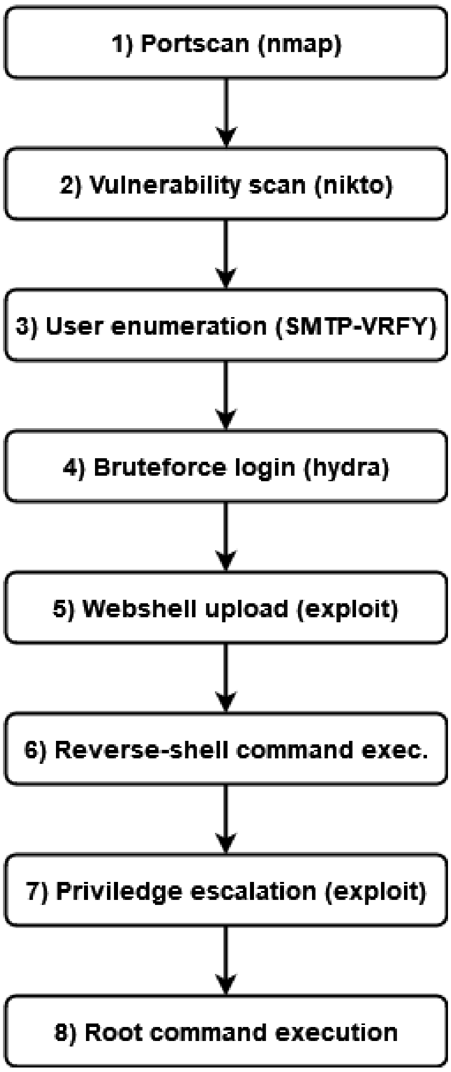

1) Portscan (nmap)
2) Vulnerability scan (nikto)
3) User enumeration (SMTP-VRFY)
4) Bruteforce login (hydra)
5) Webshell upload (exploit)
6) Reverse-shell command exec.
7) Priviledge escalation (exploit)
8) Root command execution

Figure 4.1: Procedure of multistep attack based on [LSW$^+$21].

Feature	Description
msg	This field contains a timestamp and a unique ID. Multitple log line can share the same timestamp and ID if they were generated as part of the same Audit event. The timestamp is given in unix time format and must be converted for further processing.
arch	This field contains information about the CPU architecture of the system.
syscall	This syscall field records the type of the system call that was sent to the kernel.
success	This field records whether the system call recorded in that particular event succeeded or failed.
exit	This field contains a value that specifies the exit code returned by the system call. This value varies for different system call.
a0-a3	Those fields record the first four arguments, encoded in hexadecimal notation, of the system call in this event.
items	This field contains the number of PATH auxiliary records that follow the syscall record.
ppid	This field records the Parent Process ID (PPID).
pid	This field records the Process ID (PID)
auid	This field records the Audit user ID, that is the login uid. This ID is assigned to a user upon login and is inherited by every process even when the user identity changes.
uid	This field records the user ID of the user who started the analyzed process.
gid	This field records the group ID of the user who started the analyzed process.
euid	This field records the effective user ID of the user who started the analyzed process.
suid	This field records the set user ID of the user who started the analyzed process.
fsuid	This field records the file system user ID of the user who started the analyzed process.
egid	This field field records the effective group ID of the user who started the analyzed process.
sgid	This field records the set group ID of the user who started the analyzed process.
fsgid	This field records the file system group ID of the user who started the analyzed process.
tty	This field records the terminal from which the analyzed process was invoked.
ses	This field records the session ID of the session from which the analyzed process was invoked.
comm	This field records the command-line name of the command that was used to invoke the analyzed process.
exe	This field records the path to the executable that was used to invoke the analyzed process.
key	This field records the administrator-defined string associated with the rule that generated this event in the Audit log.

Table 4.1: Features of audit log line [RH].

4.2 Audit data set exploration

The choice of a suitable DL algorithm depends on the way input data are processed. Input data usually can be classified into sequential (e.g., Audit log lines with the same process ID) or non-sequential data (e.g., single Audit log lines). In this master thesis we try to identify both point anomalies and context anomalies. In order to be able to feed previously described data sets into a DL model, some preprocessing steps are required. One of the biggest challenges is to select appropriate features for DL. In principle, there are approaches that also use DL for feature extraction. However, research from paper [HWZ$^+$21] shows that these approaches currently achieve worse values for the Recall than DeepLog. That is why we have performed a detailed analysis of the raw Audit logs manually. The considerations for selection of a feature for the AE training can be seen in Tab. 4.2 and the correlations between features are shown in Fig. 4.2. We used the Spearman's rank correlation coefficient to measure monotonic correlation between two features. The values lie between -1 and +1, -1 indicating total negative monotonic correlation, 0 indicating no monotonic correlation and 1 indicating total positive monotonic correlation.

4.3 Audit data set preprocessing

To make the features described above accessible from raw logs we use a parser. With log parsing we extract event templates from unstructured raw log data. Each log message consists of a constant event template and a variable part, where values can differ over time. The goal is to separate constant- from variable parts and form a log event. In [DLZS17] they use an effective methodology to extract so called log keys. Those log keys represent the constant part of a print statement in given log entries. In this master thesis we used a state of the art parser called Spell [DL16], which utilizes a Longest Common Subsequence (LCS) based method. The LCS method is based on the assumption that constant part takes a majority part of the whole log lines and the variable part only takes a small portion. If two log lines are produced by the same log printing statement and only differ in the dynamical variable part, the LCS of the two log lines is very likely to be constant and thus an event template. In a log line all occurring words are so-called tokens separated by delimiters. These delimiters depend on the format of the log line. The tokens are used for the construction of the LCS [DL16]. An example of the output after preprocessing is shown in Fig. 4.3.

4.4 Feature extraction

After data preprocessing, the next step is feature extraction. This is needed because DL models require numerical feature vectors as input. An important aspect is to know the range of considered features. The simplest approach is one hot encoding, where each feature value is assigned a numeric value. But the one hot encoding method is not sufficient when the numerical representation is too large for a high number of different

Feature	Useful?	Explanatory note
msg	no	The unique ID is only used for concatenating log lines with the corresponding label. The timestamp could be used for frequency analysis for certain log lines, but is not used in this master thesis.
arch	no	Could be useful but this training data set was recorded on one system and therefore the value is constant.
syscall	yes	This feature is very important because the system call is the fundamental interface between an application and the Linux kernel. It has a high correlation with items, comm and exe
success	yes	Whether a system call has succeeded or failed is important for classification and detection of anomalies.
exit	yes	The return value of successful system calls is helpful for identification of anomalies. The values are skewed and have a high correlation with ses
a0-a3	no	The hexadecimal representation of the first four arguments of a system call is not important, because we look at the decimal representation. Those fields have a high cardinality.
items	no	Because we do not look at the PATH records, this feature is not important. It has a high correlation with syscall.
ppid	no	Because the value of this feature is not unique and can be reassigned it is removed. It has a high correlation with uid, gid, euid, suid, fsuid, egid, sgid, fsgid
pid	no	Same as ppid.
auid	no	Same as ppid.
uid	no	Same as ppid.
gid	no	Same as ppid.
euid	no	Same as ppid.
suid	no	Same as ppid.
fsuid	no	Same as ppid.
egid	no	Same as ppid.
sgid	no	Same as ppid.
fsgid	no	Same as ppid.
tty	no	Constant value.
ses	no	97.9% of all audit log lines have the same value. It is not helpful for the analysis. It has a high correlation with items, comm and exe.
comm	yes	For AD it is very important to know, which commands have been entered. The values have a high cardinality and have a high correlation with syscall and exe.
exe	yes	To know the path of the executable is of great importance. The values have a high cardinality and have a high correlation with syscall.
key	no	Constant value.

Table 4.2: Analyzed Features of Audit logs.

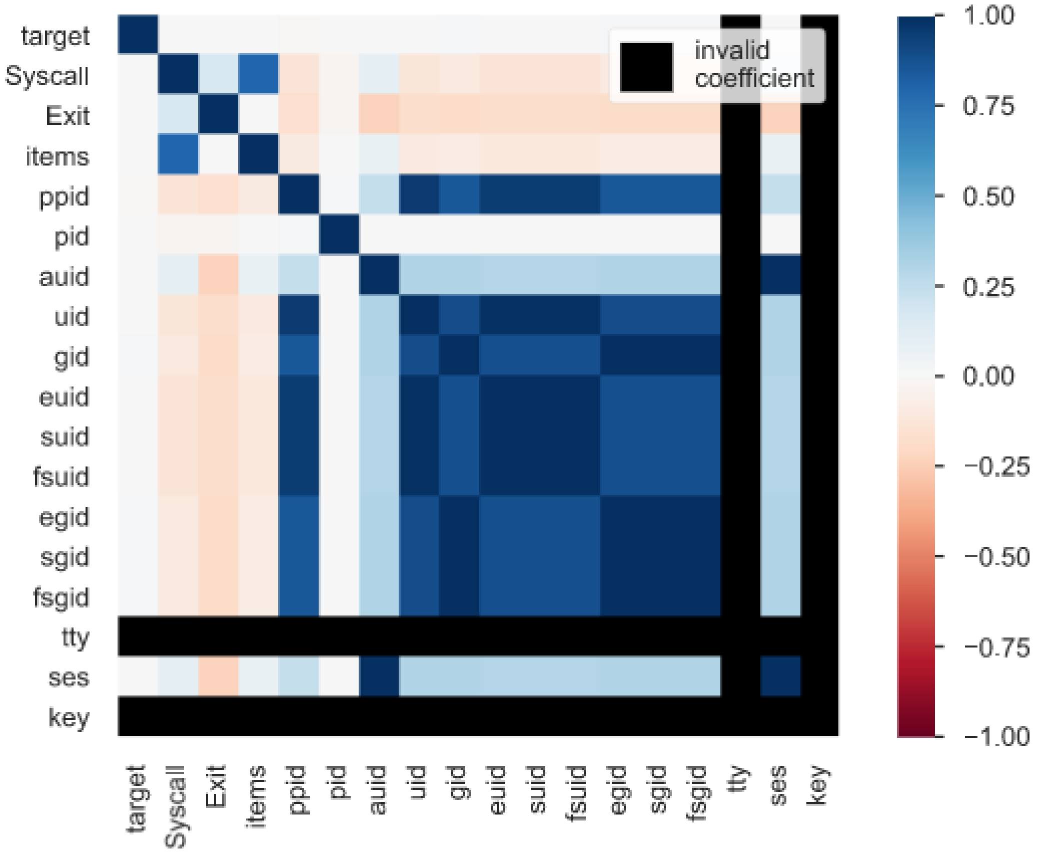
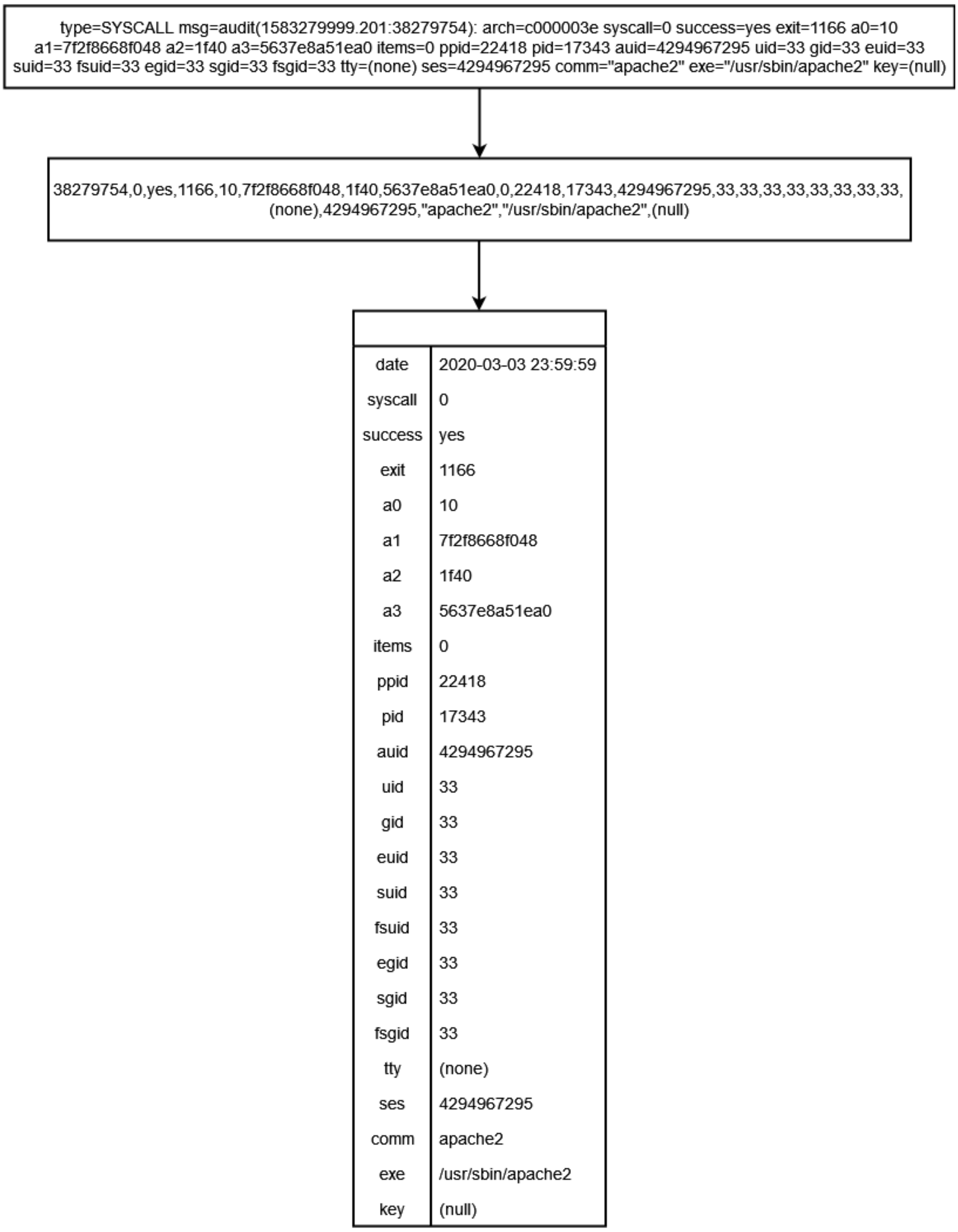


Figure 4.2: Spearman's rank correlation for features of Audit data set.

values, which can lead to many new feature dimensions and therefore to memory and computation problems. For our selected features in the AE implementation, one hot encoding is sufficient. An example after one hot encoding for 3 features of 5 log lines is shown in Fig. 4.4 where syscall has 15 distinct numeric values, success has 2 distinct string values, and exe has 69 distinct string values. The distinct string values are mapped to distinct numeric values.

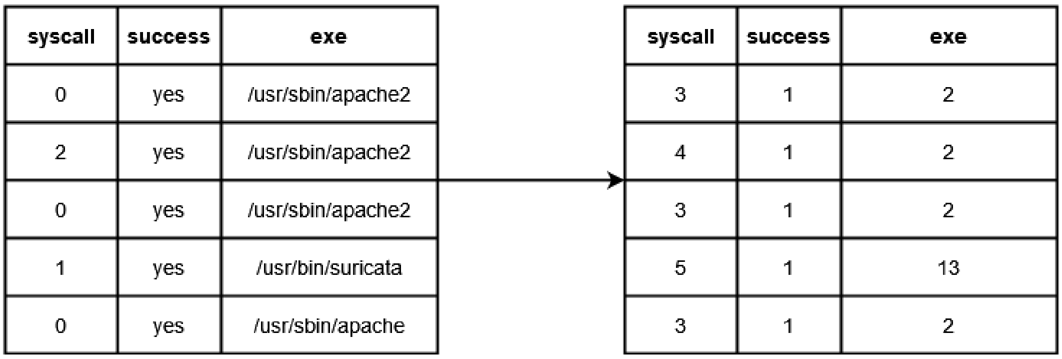


```
type=SYSCALL msg=audit(1583279999.201:38279754): arch=c000003e syscall=0 success=yes exit=1166 a0=10
a1=7f2f8668f048 a2=1f40 a3=5637e8a51ea0 items=0 ppid=22418 pid=17343 auid=4294967295 uid=33 gid=33 euid=33
suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr/sbin/apache2" key=(null)
```

```
38279754,0,yes,1166,10,7f2f8668f048,1f40,5637e8a51ea0,0,22418,17343,4294967295,33,33,33,33,33,33,33,33,
(none),4294967295,"apache2","/usr/sbin/apache2",(null)
```

date	2020-03-03 23:59:59
syscall	0
success	yes
exit	1166
a0	10
a1	7f2f8668f048
a2	1f40
a3	5637e8a51ea0
items	0
ppid	22418
pid	17343
auid	4294967295
uid	33
gid	33
euid	33
suid	33
fsuid	33
egid	33
sgid	33
fsgid	33
tty	(none)
ses	4294967295
comm	apache2
exe	/usr/sbin/apache2
key	(null)

Figure 4.3: Parsing for one single line.



syscall	success	exe
0	yes	/usr/sbin/apache2
2	yes	/usr/sbin/apache2
0	yes	/usr/sbin/apache2
1	yes	/usr/bin/suricata
0	yes	/usr/sbin/apache

syscall	success	exe
3	1	2
4	1	2
3	1	2
5	1	13
3	1	2

Figure 4.4: One hot encoding for 3 features of 5 log lines.

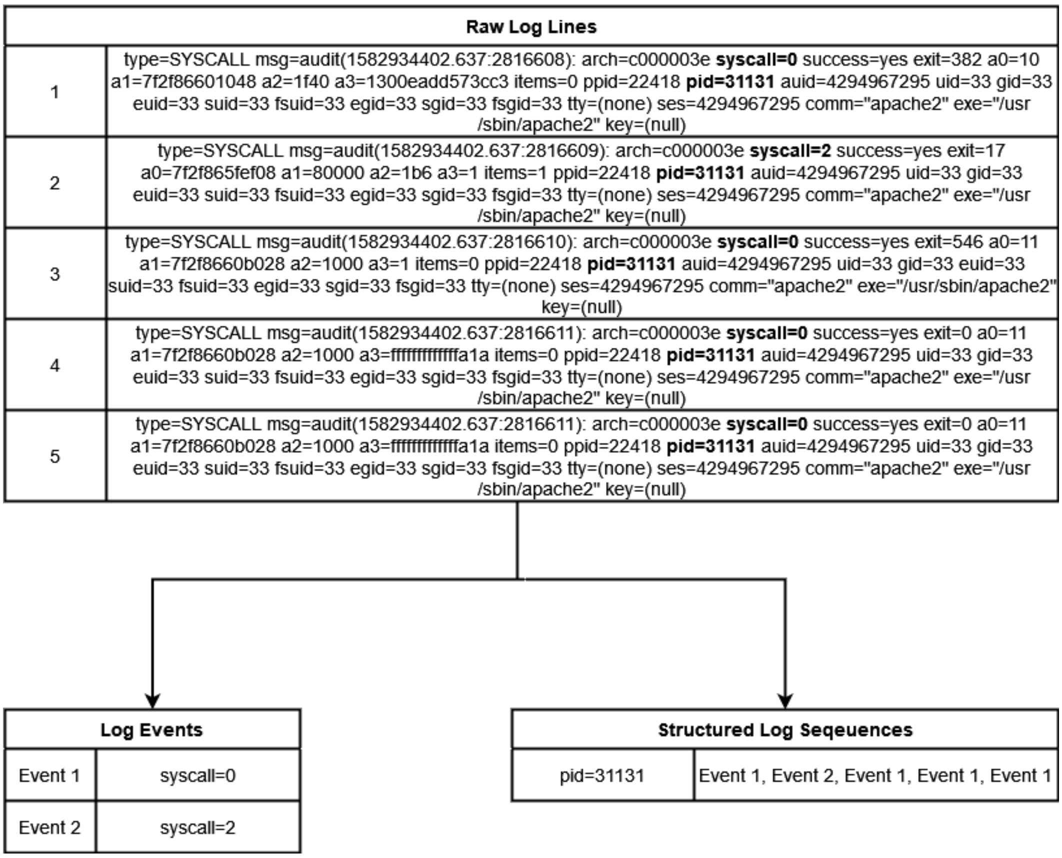
If we want to detect unusual patterns that span over multiple log lines we need to group these log lines together. These event groups can then be processed by an LSTM. The groups are divided into windows and can be based on time or sessions. Time-based grouping is again divided into sliding time - and fixed time windows. For sliding time windows, a duration is specified and pushed over all data with a static step size. For each step all log lines are grouped together that are between a start and end time of a time window. By moving the window further, an overlapping of log lines is possible. On the other hand for fixed windows, the step size is the same as window size. Log lines are always considered in exactly one window size and overlapping is not possible. Fixed time windows give a less fine-grained view on log lines but each log line is also only considered once. In contrast, session windows group log lines by session identifiers. This way, program flows can be represented very well, which may be processed at different times. It can also be used to distinguish between events that may be taking place in parallel. Unfortunately, not all types of log data have session identifiers [LOSW22].

For HDFS log data "block_id" serves as session identifier, which marks a particular execution sequence and groups together particular log lines. For the processing of Audit log lines we use a combination of sliding and session window. This way we can use the advantages of both. As session identifier we can use the pid feature. Fig. 4.5 shows how we get from unstructured Audit log lines to structured Audit log sequences.

For each log sequence an event count vector is created which counts how often a certain event occurs in a log sequence. These event count vectors are in turn combined into a matrix called event count matrix. The event count matrix is used as input for training an LSTM network. After training, the LSTM network is used to determine whether a new incoming log sequence is normal or malicious.

4.5 Autoencoder implementation

The implementation of an AE was done using Python 3.9.13 including the libraries Pandas 0.25.2, Numpy 1.20.3, Keras 2.3.1 and Scikit-learn 0.23.1. For each experiment we have removed other features, which is described in 5.1.1 and 5.1.2, to keep dimensions small



Raw Log Lines	
1	type=SYSCALL msg=audit(1582934402.637:2816608): arch=c000003e **syscall=0** success=yes exit=382 a0=10 a1=7f2f86601048 a2=1f40 a3=1300eadd573cc3 items=0 ppid=22418 **pid=31131** auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr /sbin/apache2" key=(null)
2	type=SYSCALL msg=audit(1582934402.637:2816609): arch=c000003e **syscall=2** success=yes exit=17 a0=7f2f865fef08 a1=80000 a2=1b6 a3=1 items=1 ppid=22418 **pid=31131** auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr /sbin/apache2" key=(null)
3	type=SYSCALL msg=audit(1582934402.637:2816610): arch=c000003e **syscall=0** success=yes exit=546 a0=11 a1=7f2f8660b028 a2=1000 a3=1 items=0 ppid=22418 **pid=31131** auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr/sbin/apache2" key=(null)
4	type=SYSCALL msg=audit(1582934402.637:2816611): arch=c000003e **syscall=0** success=yes exit=0 a0=11 a1=7f2f8660b028 a2=1000 a3=fffffffffffffa1a items=0 ppid=22418 **pid=31131** auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr /sbin/apache2" key=(null)
5	type=SYSCALL msg=audit(1582934402.637:2816611): arch=c000003e **syscall=0** success=yes exit=0 a0=11 a1=7f2f8660b028 a2=1000 a3=fffffffffffffa1a items=0 ppid=22418 **pid=31131** auid=4294967295 uid=33 gid=33 euid=33 suid=33 fsuid=33 egid=33 sgid=33 fsgid=33 tty=(none) ses=4294967295 comm="apache2" exe="/usr /sbin/apache2" key=(null)

Log Events	
Event 1	syscall=0
Event 2	syscall=2

Structured Log Seqeuences	
pid=31131	Event 1, Event 2, Event 1, Event 1, Event 1

Figure 4.5: From unstructured log lines to structured log sequences.

from the start and thus calculation of models small and also to eliminate non-meaningful features from the start. After we have redacted the data set to the features we want to look at, we use the Keras class Tokenizer which allows to vectorize occuring text, e.g., exe feature. As a result each text is turned into an integer. A list of tokenized log lines is passed to the AE as input. The model of the AE is built with so called dense layers. The dimension of those layers depends on the dimension of the input. Each dense layer works with Rectified Linear Unit (ReLu) as activation function. We have chosen ReLu as activation function, because of its computational performance [SH20]. As optimizer, the Adam optimizer was used and the loss function is calculated with the RMSE. The Adam optimizer is often used for tasks where sparse gradients are to be expected. The advantages of this optimizer are increased computational performance and low memory requirements [KB17]. To get fast results, experiments have shown that the model converges very fast, if a low number of epochs with 5 was selected. The batch size was chosen with 256, which is an average batch size for the size of the Audit data set. The test validation data was also randomly shuffled to eliminate temporal dependencies on the log lines. After the model has been trained, a prediction is made for each log line as to whether it is normal data or an anomaly. The RMSE error is calculated for this.

4.6 Deeplog implementation

A in depth research was performed. Common databases for open source, such as Github and Gitlab, were searched. A search string containing the terms AD, DL, and log data was used. To narrow down the results, we filtered by the number of ratings, the year of publication and the data sets used. After a thorough research, the open source implementation LogDeep of [DA20] stood out, because of its detailed documentation and ease of adaptation.. There are more open source implementations which can be found in Tab. 3.2. It combines the work of [DLZS17],[ZXL$^+$19] and [MLZ$^+$19] to a framework for AD utilizing log data and has already been adapted in other scientific work, such as [GYW21]. Because of the scientific relevance, we have focused on the DeepLog part from this implementation. As previously described, DeepLog uses LSTM as a DL model. The main adjustable parameters in the LogDeep implementation are:

- L: Number of layers of LSTM.
- α: Number of memory units in one LSTM block.
- Window size: Length of window under consideration.
- Candidates: Number of candidates that were predicted with their respective probabilities.

After the parameters have been set, LogDeep reads the training log sequences. The log sequence must have at least the length of the specified window size. After that, the read log sequences are fed to the LSTM. In principle, training of the DL model and AD, that uses the trained model, are separate. The reason for this is that after the training, the model is stored and can be transferred somewhere else in order to perform AD there as well. As with the AE, the Adam optimizer is used for the LSTM. The LSTM delivers candidates with a certain probability of which log key is next in the log sequence. Therefore we use the cross entropy as loss function here, which quantifies the difference between probability distributions. After training, LogDeep can be used for AD. For this purpose, benign and malicious test data are read in separately. The model calculates which log key comes next for each individual window and compares whether this matches the log keys in the test data. If the following log key is not contained in the proposed candidates, this log sequence is marked as anomaly.

4.7 Federated learning

4.7.1 Flower

In July 2020 flower was released [BTM$^+$20]. flower is a scaleable open source framework, in terms of number of clients. With this it is very convenient to implement existing DL setups in a federated setting and evaluate their convergence- and training time. One of the main advantages of flower is that it is programming language- and ML framework-agnostic by design. The flower core framework, as common in FL, consists of a server and client part. The flower server is further divided into three main components: Client Manager, FL loop and Strategy. The client manager is responsible for communication between connected clients. The strategy describes the aggregation algorithm of updates. The algorithm we have chosen will be discussed below in Sect. 4.7.2. The FL loop works as a central entity. It queries strategies, receives updates from connected clients, calculates global models and returns calculated models to clients via client manager. The client side only waits for instructions from the server and executes them. The client manager works with so called flower messages. These messages are based on Remote Procedure Call (RPC) streams. RPC is efficient for low bandwidth transmission, because of the binary serialization format. So the client manager works as a RPC server for sending and receiving flower messages [BTM+20]. The flower architecture described was summarized in Fig. 4.6.

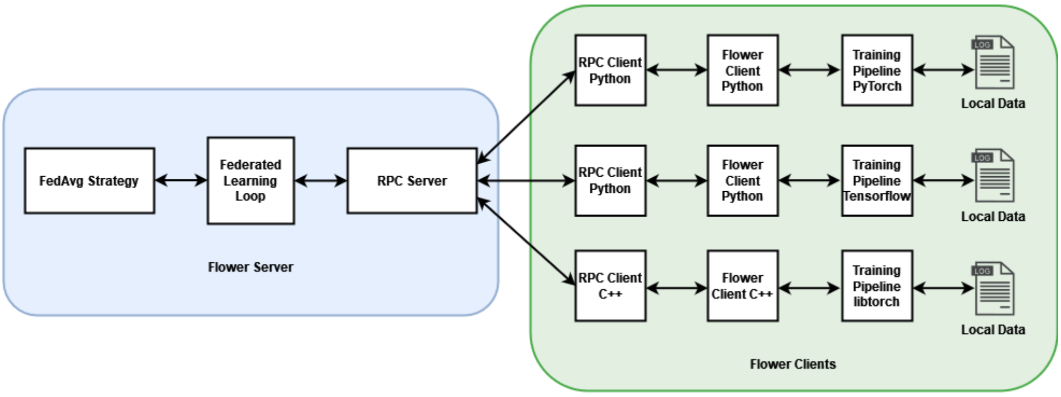


Figure 4.6: Flower architecture based on [BTM+20].

For this master thesis first a standalone LogDeep client is re-implemented and evaluated. Then we port this working and tested LogDeep implementation to the flower environment. Each client runs the same LogDeep implementation, but with different data sets. We mainly investigate the capability of FL, how clients learn DL models for AD with different sized data sets.

4.7.2 Federated Averaging

The Federated Averaging (FedAvg) aggregation algorithm assumes K participating clients, where each client has a number of n_k log lines as training data and a local model weight w_{t+1}^k. For the $(t+1)$-th round aggregation is given by Eq. 4.1.

$$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k \tag{4.1}$$

where the global model weight update is denoted with w_{t+1}. n indicates total number of log lines at K clients [LCL+19]. Besides FedAvg, there are more complex aggregation algorithms, which for example select only a part of total available clients K and which may differ from round to round. A listing and description of further aggregation algorithms can be found in [LPBA22]. In this master thesis we only consider FedAvg for simplicity.

4.8 Metrics

In this thesis binary classification is chosen to measure the effectiveness of the studied AD. True Positives (TP) are anomalous log lines or sequences correctly detected as anomalies. Benign log lines or sequences evaluated as normal on the other hand are marked as True Negatives (TN). False Positives (FP) are benign log lines or sequences which are wrongfully marked as anomalies. Finally, False Negatives (FN) are anomalies which were not detected as such. The most widely used metrics in this context are Accuracy, Precision, Recall, F1-Score and are briefly described below:

The Accuracy indicates the ratio of correctly classified log lines or sequences to total amount of input log lines or sequences.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.2}$$

The Precision indicates the number of log lines or sequences that are actually anomalies divided by all classified anomalous log lines or sequences.

$$Precision = \frac{TP}{TP + FP} \tag{4.3}$$

The Recall indicates the probability that an anomaly is detected as such.

$$Recall = \frac{TP}{TP + FN} \tag{4.4}$$

The F1-Score is the summed average of Precision and Recall.

$$F1 - Score = \frac{2TP}{2TP + FP + FN} \tag{4.5}$$

CHAPTER 5

Experiments

In this chapter, all experiments performed are listed and described in detail. Basically, DL requires a lot of processor power and Random Access Memory (RAM). However, nowadays, more and more edge devices have the required resources and researchers work on the resource-hungry requirements of DL. In the beginning experiments were performed on a laptop provided by the AIT. The resources it had were sufficient in the beginning. With time it became clear that simultaneous model training and other analysis brought this laptop to its limits. Thankfully the AIT provided another possibility to process simultaneous experiments faster. Resources were provided to realize OpenStack instances. The specifications of the laptop and the individual OpenStack instances can be found in Tab. 5.1. A central OpenStack instance was used to verify experiments which were previously performed with the laptop. The second type of OpenStack instance describes the specification of FL clients and servers.

	Laptop	**OpenStack (central)**	**OpenStack (FL)**
Processor	IntelCore i7-8665U	8 VCPU	2 VCPU
RAM	32GB	16GB	4GB
Operating System	Windows 10 20H2	Ubuntu 20.04	Ubuntu 20.04

Table 5.1: Specifications of the laptop and the individual OpenStack instances.

All experiments that were performed can be found in Tab. 5.2, where ✓ describes that a function (Padding, FL or Splitting) was used in the experiment. All experiments with a - describes that the function is not relevant for this experiment. Finally, experiments with ✓/✗ are comparative experiments where the function is switched on and off.

Section No.	Model	Dataset	Padding	FL	Splitting	Description
5.1.1	AE	Audit (~18 mil.)	-	-	-	3 features
5.1.2	AE	Audit (~18 mil.)	-	-	-	5 features
5.2.1	LSTM	HDFS	✓	-	-	Verification of published result of DeepLog [DLZS17]
5.2.2	LSTM	HDFS	✓/✗	-	-	Comparison padding and nopadding
5.2.3	LSTM	Audit (~18 mil.) Audit (2 mil.)	✓	-	-	Adapting 5.2.1 for Audit data set
5.2.4	LSTM	Audit (~18 mil.) Audit (2 mil.)	✓/✗	-	-	Comparison padding and nopadding
5.2.5	LSTM	Audit (~18 mil.)	✓	-	-	Change of metrics between 5 and 35 epochs
5.2.6	LSTM	Audit (~18 mil.)	✓	-	-	Change of metrics between 1 and 7 candidates
5.2.7	LSTM	Audit (~18 mil.)	✓	-	-	Change of metrics between 3 and 12 for length of window
5.3.1	LSTM	HDFS	✓	✓	even/uneven	Port the LogDeep implementation from Exp. 5.2.1 to 5 clients
5.3.2	LSTM	HDFS	✓	✓	even	Even splitting of training data for 5 clients According to Tab. 5.9
5.3.3	LSTM	HDFS	✓	✓	uneven	Uneven splitting of training data for 5 clients According to Tab. 5.10
5.3.4	LSTM	HDFS	✓	✓	uneven	Change of metrics between 300 and 3 epochs
5.3.5	LSTM	Audit (2 mil.)	✓	✓	even/uneven	Port the LogDeep implementation from Exp. 5.2.3 to 5 clients
5.3.6	LSTM	Audit (2 mil.)	✓	✓	even	Even splitting of training data for 5 clients According to Tab. 5.11
5.3.7	LSTM	Audit (2 mil.)	✓	✓	uneven	Uneven splitting of training data for 5 clients According to Tab. 5.12

Table 5.2: Table of Experiments.

5.1 Autoencoder

The baseline of the Audit [LSW$^+$21] data set consists of 18,428,737 log lines. The log lines are shuffled with a random state. This has the advantage that the training data set does not only include log lines for training that occur up to a certain point in time. The same applies to the test data set. 70% of the log lines were selected for the training data set and 30% for the test data set. This distribution was taken from [YB22]. The exact division into training and test data can be found in the following Tab. 5.3.

Train (normal) log lines	Test normal log lines	Test abnormal log lines
12,895,886	5,526,809	6,041

Table 5.3: Splitting of Audit data set with ~18 million log lines.

5.1.1 Autoencoder with 3 features

The selected features for this experiment can be seen in Tab. 5.4. The msg field was removed because the unique ID is only used for concatenating log lines with the corresponding label and the timestamp could be used for frequency analysis for certain log lines, but is not used in this master thesis. Each log line is reduced to 3 features. All log lines have a value for the 3 features present. The selection of the features is based on the findings conducted by the Audit data set exploration in Sect. 4.2 and Tab. 4.2. If a log line is unique it gets a unique token. After tokenization we get 86 unique tokens. Due to the 3 selected features the input dimension of the AE is also 3. The input layer is followed by a dense layer with dimension 2 and the ReLu activation function. In the middle there is a dense layer with dimension 1 which is also called bottleneck layer. After the bottleneck layer the decoder comes with the first dense layer with dimension 2 and an output dense layer with dimension 3. The structure described is shown in Fig. 5.1.

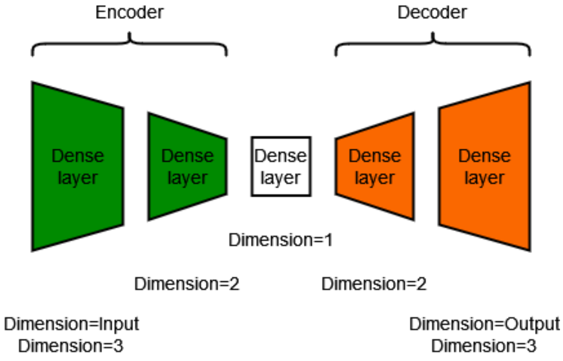


Figure 5.1: Structure of AE layers for 3 features.

Features selected	Features removed
syscall	msg
success	exit
exe	a0
	a1
	a2
	a3
	ppid
	pid
	auid
	uid
	gid
	euid
	suid
	fsuid
	egid
	sgid
	fsgid
	tty
	ses
	comm
	key

Table 5.4: Selected & removed Features for 5.1.1.

5.1.2 Autoencoder with 5 features

The selected features for this experiment can be seen in Tab. 5.5. The reasons for the selection of the 5 features can be taken from Tab. 4.2. Each log line is reduced to 5 features. After tokenization we get 13,740 unique tokens. Due to the 5 selected features the input dimension of the AE is also 5. The input layer is followed by a dense layer with dimension 3 and the ReLu activation function. This completes the encoder and is followed by the decoder with the first dense layer with dimension 3 and an output dense layer with dimension 5. The structure described is shown in Fig. 5.2.

5.2 LogDeep (LSTM)

The re-implementation of DeepLog called LogDeep [DA20] was used as baseline for all further experiments. In the first step the code base was analyzed. Since only the DeepLog part is relevant for this master thesis, the code had to be refactored. With the original code base only the HDFS data set can be processed. The code has been adapted to allow a user to choose if he wants to process a HDFS or an Audit data set. In our implementation the parameters for the selected data set can be set and padding, which

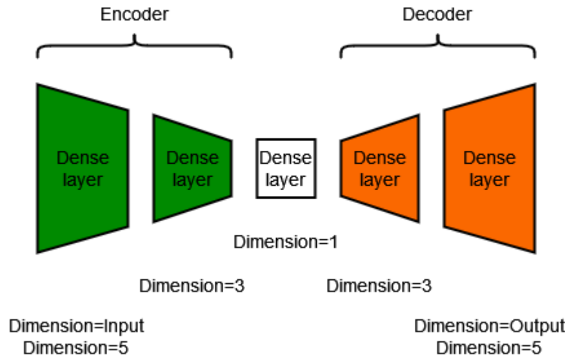


Figure 5.2: Structure of AE layers for 5 features.

Features selected	Features removed
syscall	msg
success	a0
exe	a1
exit	a2
comm	a3
	ppid
	pid
	auid
	uid
	gid
	euid
	suid
	fsuid
	egid
	sgid
	fsgid
	tty
	ses
	key

Table 5.5: Selected & removed Features for 5.1.2.

is described in Sect. 5.2.2 can be activated or deactivated. The following requirements for the current implementation apply: python>=3.6 and pytorch>=1.1.0.

5.2.1 LogDeep (LSTM) with HDFS data set

First we verified if the evaluation metrics of LogDeep correspond to those published in [DLZS17]. For this purpose we set the window size to 10. The hidden size, which defines the number of hidden states was set to 64. Further, the number of layers was set to 2, number of candidates to 9, batch size to 2,048, and number of epochs to 300. These values were also taken from [DLZS17]. As a basis for evaluation we used the freely available HDFS data set in parsed form. This way our implementation can be compared with other scientific publications and errors caused by parsing can be ignored. The number of log keys is 29 and the exact breakdown of the data set can be taken from Tab. 5.6.

Train (normal) sequences	Test normal sequences	Test abnormal sequences
4,855	553,366	16,838

Table 5.6: Splitting of HDFS data set.

5.2.2 LogDeep (LSTM) with HDFS data set and padding function

While analyzing the source code of LogDeep, we came across an interesting point. If we take a closer look at the HDFS data set, we see that train- and test normal log sequences consist of at least 10 consecutive log keys. However, shorter sequences also occur in test abnormal log sequences. This results in a problem for the window size, because the LSTM determines how far it stores past information. In order to be able to analyze these short sequences with the model, so-called padding is used. In this case, log sequences are filled with a log key, that is not yet in use, in order to achieve desired window size. This log key must be unquie and must not have occurred before, otherwise the data set would be corrupted. Based on this discovery, we conducted another experiment to show what effect padding has on LogDeep and the results can be found in Sect. 6.2.2. If padding is omitted, LogDeep cannot process and discards all test abnormal log sequences in the AD phase, whose number of log keys is less than the window size. The size of the training and test normal data set remains the same as in experiment 5.2, only the size of the test abnormal data set is reduced. The exact breakdown can be found in the Tab. 5.7.

Train (normal) sequences	Test normal sequences	Test abnormal sequences
4,855	553,366	10,647

Table 5.7: Splitting of HDFS data set without padding.

5.2.3 LogDeep (LSTM) with Audit data set

In order to get Audit log sequences, preprocessing and parsing had to be adapted for the Audit [LSW+21] data set. First, log lines were grouped based on the pid feature. This way, we get all log lines that belong to a process. With the Spell parser, these grouped log lines were parsed based on the syscall feature. Thus, each syscall that occurs in the data set corresponds to a log key. There are 268 log keys in total. The result of these steps are ordered log sequences that can be used as input for the LSTM. In order to make experiments and their results for the Audit data comparable with those for HDFS data, we decided to use padding in the first step. The baseline of the Audit data set consists of 18,428,737 log lines. For the Exp. 5.3.5,5.3.6, and 5.3.7 a reduced to 2 million log lines Audit data set is used in the FL environment due to performance limitation. In this experiment for a centralized approach we also compare if the reduction has an impact on the metrics before we conduct experiments in the FL environment. The division into training, test normal and test abnormal sequences can be seen in the Tab. 5.8. The LSTM was not fundamentally changed. We only had to adapt a few parameters for the Audit data set. The following remained the same: window size was set to 10, hidden size to 64, number of layers to 2, and batch size to 2,048. Only the number of candidates with 6 and number of epochs with 20 were changed. For verification the best values for the metrics are obtained with these parameter values. We have conducted further experiments which are described below.

	Train (normal) sequences	**Test normal sequences**	**Test abnormal sequences**
~18 mil.	33,634	8,409	90
2 mil.	4,142	1,036	90

Table 5.8: Splitting of Audit data set.

5.2.4 LogDeep (LSTM) with Audit data set and padding function

Comparative experiments to see what effect padding has on Audit data.

5.2.5 LogDeep (LSTM) with Audit data set: Number of epochs

For this experiment, only the number of epochs change. All other parameters remain the same. We investigate how metrics change between 5 and 35 epochs.

5.2.6 LogDeep (LSTM) with Audit data set: Number of candidates

For this experiment, only the number of candidates change. All other parameters remain the same. We investigate how metrics change between 1 and 7 candidates.

5.2.7 LogDeep (LSTM) with Audit data set: Length of window

For this experiment, only the window size changes. All other parameters remain the same. We investigate how metrics change between 3 and 12 for length of window.

5.3 Federated learning

For FL experiments we port the working LogDeep implementation to 5 clients. The number of clients was chosen due to resource limitations. First, experiments were started on a laptop, described above. Further on, the FL implementation was ported to an OpenStack environment. The server uses Federated Averaging as update aggregation methods and starts the FL process only when all 5 clients are available. The connection to clients happens on the laptop locally and for the OpenStack implementation via IP addresses of clients. We have chosen a scenario as a basis for FL, where all clients have different training data but the same normal and abnormal test data locally. At the start of the FL process, the server passes model parameters with the fit method to all the clients. All the clients receive parameters for their local model and start the first training phase. After completion of the first training phase, all clients send their current local model parameters as an update to the server with the method get_parameters and evaluate the local model with the normal and abnormal test data. The server aggregates all received updates and sends them back again. After that, all clients have the updated model parameters and the test data is evaluated again with them in the so-called evaluation phase. With this same level of knowledge, the clients start the next round and try again to improve the model with their local training data. The process described here can be repeated as often as desired and is shown for 2 rounds in Fig. 5.3.

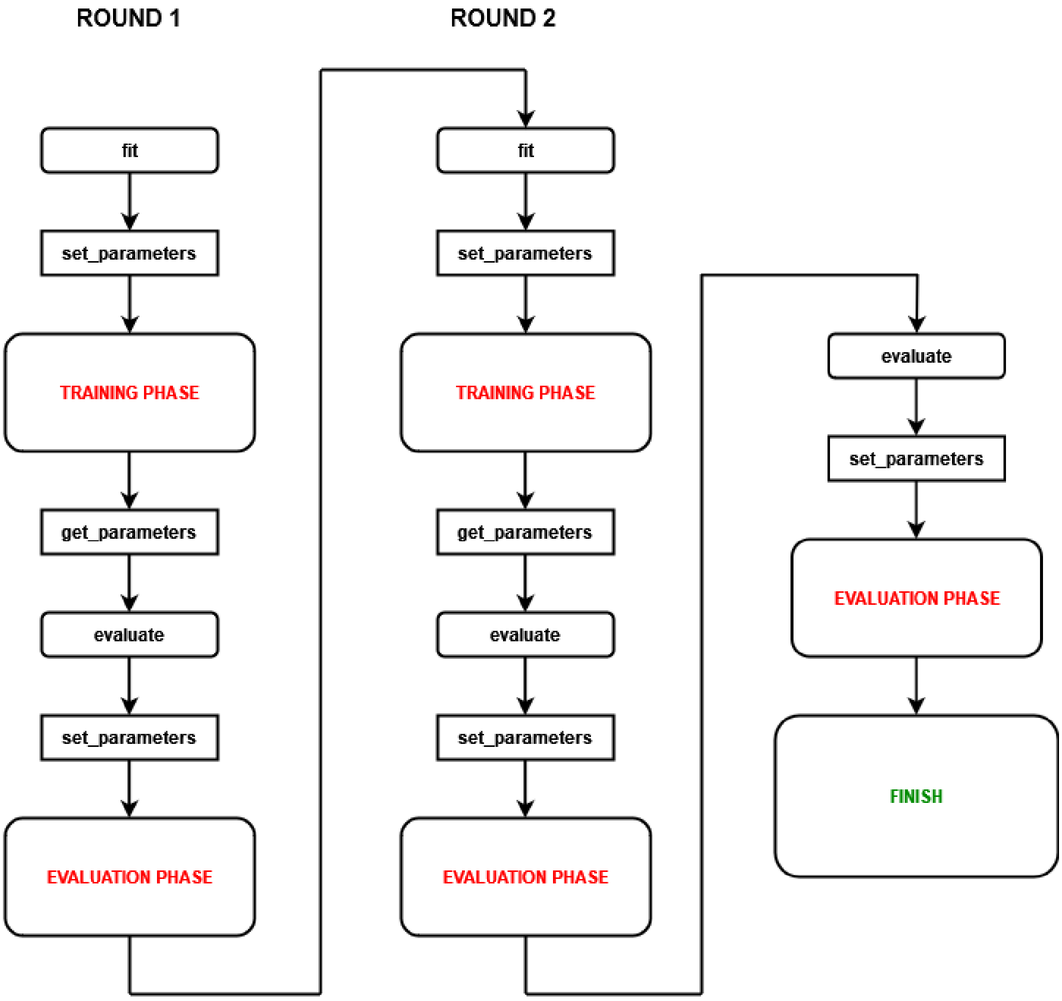


Figure 5.3: Flowchart of flower framework.

5.3.1 Federated learning with HDFS data set

The LogDeep client for HDFS data uses again window size 10, hidden size 64, number of layers 2, and number of candidates 9. The number of log keys is again 29. As a baseline for the first experiments the number of epochs is set to 300. Furthermore, experiments are conducted where the number of epochs is decreased to 3 and compared to the results for number of epochs of 300 in Sect. 5.3.4. Basically, we use padding to make the results comparable with the central implementations.

5.3.2 Federated learning with even splitted HDFS data set

To show advantages of FL we have decided on two illustrative scenarios. First to divide training data evenly among all clients and second to divide it by percentage such that clients get a different amount of training data. With even as well as with uneven splitting the sequences are shuffled. Otherwise it could be that certain sequences only appear at certain times, e.g., at night, which would only be seen by one client, if we keep the temporal order of sequences. The even distribution is described in Tab. 5.9.

Client No.	**Train (normal) sequences**	**Test normal sequences**	**Test abnormal sequences**
1	971 (20%)	553,366	16,838
2	971 (20%)	553,366	16,838
3	971 (20%)	553,366	16,838
4	971 (20%)	553,366	16,838
5	971 (20%)	553,366	16,838

Table 5.9: Even splitting of HDFS data set for 5 FL clients.

5.3.3 Federated learning with uneven splitted HDFS data set

We have selected the following uneven distribution key for the percentage distribution: Client 1 receives 5%, Client 2 receives 20%, Client 3 receives 20%, Client 4 receives 5% and Client 5 receives 50%. In our opinion, this distribution could describe a realistic scenario, where one client keeps a lot of log data locally and can use it for model training and other clients keep much less log data locally but can learn from other participating clients. The uneven distribution is described in Tab. 5.10.

Client No.	Train (normal) sequences	Test normal sequences	Test abnormal sequences
1	243 (5%)	553,366	16,838
2	971 (20%)	553,366	16,838
3	971 (20%)	553,366	16,838
4	243 (5%)	553,366	16,838
5	2,427 (50%)	553,366	16,838

Table 5.10: Uneven splitting of HDFS data set for 5 FL clients.

5.3.4 Federated learning with HDFS data set: Number of epochs

For this series of experiments we vary the number of epochs for the uneven distributed HDFS training data set. It should be shown that by FL reducing the number of epochs and thus shortening training time still produces acceptable results. For this experiment, we reduce the number of epochs from 300 to 3. A total of 3 FL rounds are run.

5.3.5 Federated learning with Audit data set

Again, the LogDeep implementation, which was tested centrally, is ported to 5 clients. Due to computational limitations we use the Audit data set, with 2 million log lines and the splitting which is shown in Tab. 5.8.

5.3.6 Federated learning with even splitted Audit data set

In this section the Audit data set is split even and exact splitting can be taken from Tab. 5.11. The sequences are shuffled again during splitting. It should be shown how the metrics develop over 3 FL rounds.

Client No.	Train (normal) sequences	Test normal sequences	Test abnormal sequences
1	828 (20%)	1,036	90
2	828 (20%)	1,036	90
3	828 (20%)	1,036	90
4	828 (20%)	1,036	90
5	828 (20%)	1,036	90

Table 5.11: Even splitting of Audit data set for 5 FL clients.

5.3.7 Federated learning with uneven splitted Audit data set

In this section the Audit data set is split uneven and exact splitting can be taken from Tab. 5.12. The sequences are shuffled again during splitting.

Client No.	**Train (normal) sequences**	**Test normal sequences**	**Test abnormal sequences**
1	207 (5%)	1,036	90
2	828 (20%)	1,036	90
3	828 (20%)	1,036	90
4	207 (5%)	1,036	90
5	2,071 (50%)	1,036	90

Table 5.12: Uneven splitting of Audit data set for 5 FL clients.

CHAPTER 6

Results

In this chapter we present results for the experiments described in the previous chapter. Like in the previous chapters, this chapter is divided into results for centralized approach (AE and LogDeep (LSTM)) experiments and FL experiments.

6.1 Results of Autoencoder experiments

This section presents the results retrieved from the AE experiments.

6.1.1 Results of Autoencoder experiments with 3 features

This section presents the results retrieved from the Exp. 5.1.1. After 5 epochs of training the model has an Accuracy of 95% for the test data. For AD we calculate the RMSE. The RMSE for anomalies should be high, because the trained model should reconstruct these log lines poorly. Unfortunately this is not the case. The values of the RMSE can be seen in Fig. 6.1. The explanation for those poor results is as follows:

1. If you discard duplicates of the ~18 million Audit log lines with 3 features that comprise the training and normal test data set, 335 unique Audit log lines remain. The same is done for the abnormal test data set, leaving 120 unique Audit log lines. The low number of unique log lines results from the low number of distinct values of the 3 features considered as discussed in the Sect. 4.4. Syscall has 15 distinct numeric values, success has 2 distinct string values, and exe has 69 distinct string values. If we compare the unique Audit log lines with each other, we see that 83 unique abnormal Audit log lines (=69%) occur in the training and test data set in exactly the same way. An example of identical log lines with 3 features and different labels can be seen in Fig. 6.2.These identical log lines were sometimes labeled as benign but also sometimes as malicious when the Audit data set was

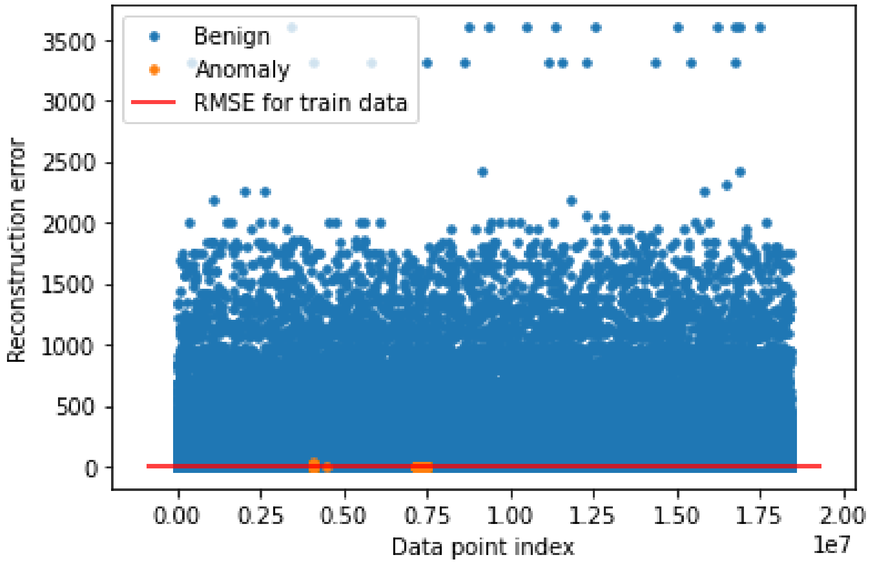


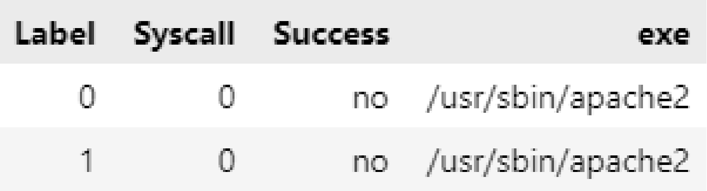
Figure 6.1: RMSE for ~18 million Audit log lines with 3 features.

recorded. They were labeled as malicious if they occurred within a time when an attack was active. So the difference in how such a log line was labeled is based on a different context which makes an analysis of individual log lines very difficult. With this limitation in mind, the AE does exactly what it is supposed to do. It does not detect the identical log lines as an anomaly if it was already classified as benign in the training. The remaining 37 abnormal Audit log lines do not occur exactly identically.

2. The averaged RMSE can be taken from Tab. 6.1. An important addition to the averaged RMSE for train- and test normal data and Fig. 6.1 is that 14,686,862 (=79%) of the train- and test normal log lines are even below the averaged RMSE for abnormal test data. However, there are outliers that also increase the values for the averaged RMSE. The Audit data set is also very imbalanced. As Tab. 5.3 shows, there are ~18 million benign log lines compared to 6041 malicious log lines. Thus no threshold can be set to reliably distinguish test abnormal log lines (=anomalies) from train and test normal log lines. This indicates that the small number of features and their conditional variability prevent application for AD.

Averaged RMSE for train data	2.79
Averaged RMSE for normal test data	2.78
Averaged RMSE for abnormal test data	2.20

Table 6.1: Averaged RMSE for experiment 5.1.1.



Label	Syscall	Success	exe
0	0	no	/usr/sbin/apache2
1	0	no	/usr/sbin/apache2

Figure 6.2: An example of identical log lines with 3 features and different labels.

6.1.2 Results of Autoencoder experiments with 5 features

This section presents the results retrieved from the Exp. 5.1.2. After 5 epochs of training the model has an accuracy of 64.84% for the test data. The results of the calculated RMSE can be seen in Fig. 6.3. The explanation for the poor results is as follows:

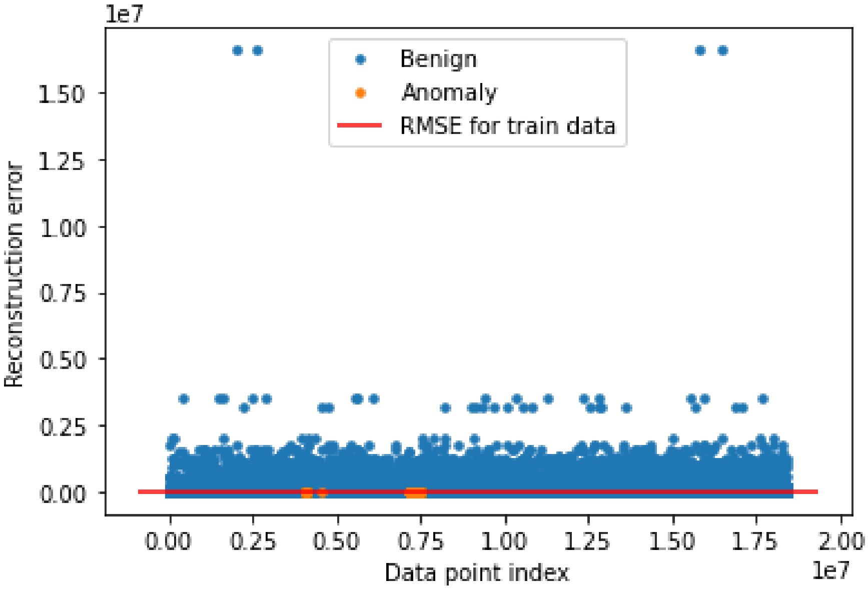


Figure 6.3: RMSE for ~18 million Audit log lines with 5 features.

1. If you discard the duplicates of the ~18 million Audit log lines with 5 features, that comprise the training and normal test data set, 27,671 unique Audit log lines remain. The same is done for the abnormal test data set, leaving 654 unique Audit log lines. If we compare the unique Audit log lines with each other, we see that 293 unique abnormal Audit log lines (=44%) occur in the training and test data set in exactly the same way. The percentages are lower than in 6.1.1 but still too high. Thus, the AE cannot detect those anomalies at all because it has learned them as benign in the training phase. The remaining 361 abnormal Audit log lines do not occur exactly identically.

2. The averaged RMSE can be taken from Tab. 6.2. An important addition to the averaged RMSE for train- and test normal data and Fig. 6.3 is that 8,656,001 (=48%) of the train- and test normal log lines are even below the averaged RMSE for abnormal test data. However, there are outliers that also increase the values for the averaged RMSE. The Audit data set is also very imbalanced. As Tab. 5.3 shows, there are ~18 million benign log lines compared to 6041 malicious log lines. The RMSE has generally increased and unfortunately, even with 5 features, it is still not possible to make accurate statements for AD.

Averaged RMSE for train data	16.87
Averaged RMSE for normal test data	16.78
Averaged RMSE for abnormal test data	6.27

Table 6.2: Averaged RMSE for experiment 5.1.2.

Changing the dimension number to 2 for the dense layer before the bottleneck did not improve the performance of the model. Increasing features for the AE did not bring any improvement either.

In summary, it was planned to use the AE to analyze single log lines and detect point anomalies. As the results in Sect. 6.1.1 and 6.1.2 have shown, the Audit data set is unsuitable for AD with an AE. In the data set, individual log lines are labeled both as benign and malicious even though their features are identical. The AE cannot detect an anomaly that it actually learned in training as benign log lines. At this point we decided to change the approach to LSTM since it does not only consider single log lines and by looking at sequences, we expect better performance.

6.2 Results of LogDeep (LSTM) experiments

This section presents the results retrieved from the LogDeep experiments for the HDFS and Audit data sets.

6.2.1 Results of LogDeep (LSTM) experiments with HDFS data set

The results of Exp. 5.2.1 show that the re-implementation achieves the same metrics as in [DLZS17]. The comparison is listed in Tab. 6.3.

	Accuracy	**Precision**	**Recall**	**F1-Score**
DeepLog [DLZS17]	-	95%	96%	96%
LogDeep [DA20]	99.76%	95.63%	96.35%	95.99%

Table 6.3: Comparison between DeepLog and LogDeep.

A subsequent experiment was conducted to see if changes to the number of epochs had a large effect on metrics. The results in Tab. 6.4, and Fig. 6.4 and 6.5 show that LogDeep is very stable with respect to changes in the number of epochs. But the very best metrics overall are at an epoch count of 300. If the number of epochs is increased further, an over-fitting of the model occurs and the metrics deteriorate again.

Epochs	Accuracy	Precision	Recall	F1-Score
5	99,54%	87,15%	99,17%	92,77%
10	99,54%	87,16%	99,17%	92,78%
20	99,52%	86,77%	99,16%	92,55%
50	99,50%	89,53%	94,32%	91,86%
100	99,53%	90,32%	94,51%	92,37%
200	99,62%	94,25%	93,02%	93,63%
300	**99,76%**	**95,63%**	**96,35%**	**95,99%**
350	99,65%	95,87%	92,19%	93,99%
390	99,65%	95,53%	92,61%	94,05%
450	99,65%	95,55%	92,63%	94,06%
550	99,65%	95,63%	92,60%	94,09%

Table 6.4: Variation of number of epochs for LogDeep with HDFS data.

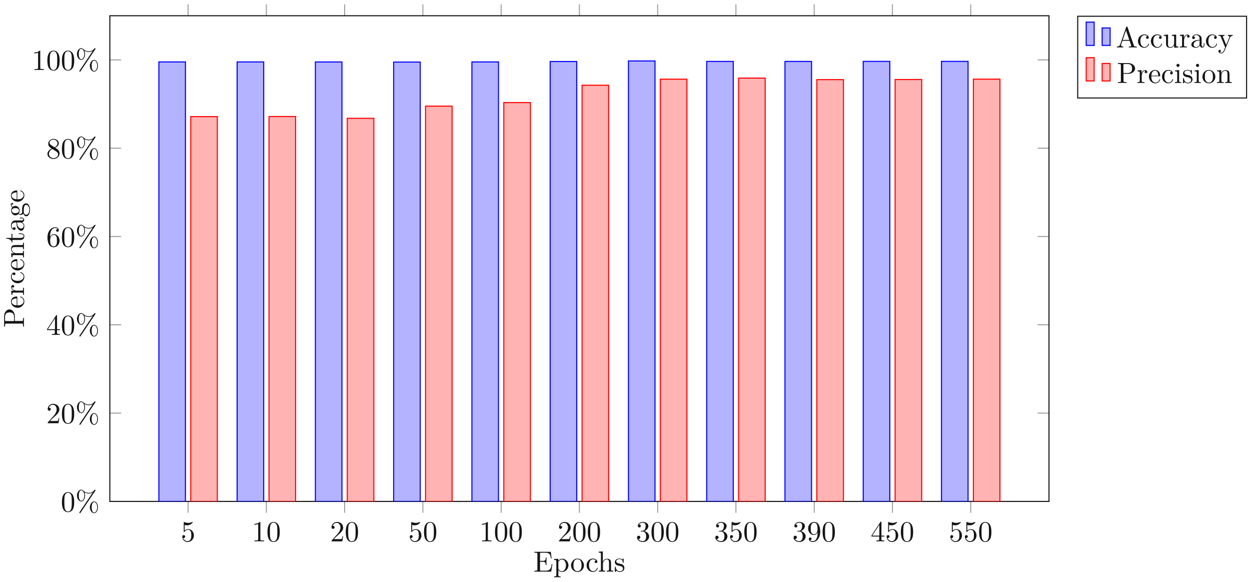


Figure 6.4: Effect on Accuracy and Precision for changing epoch number for LogDeep with HDFS data.

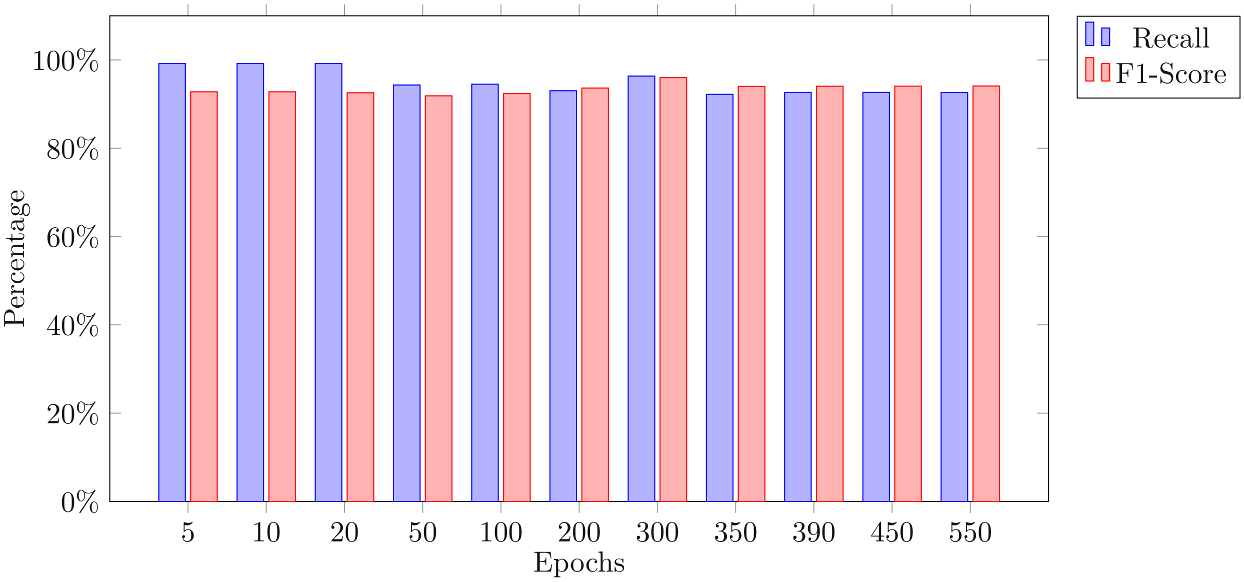


Figure 6.5: Effect on Recall and F1-Score for changing epoch number for LogDeep with HDFS data.

6.2.2 Results of LogDeep (LSTM) experiments with HDFS data set and padding

This section presents the results retrieved from the Exp. 5.2.2. The effects of padding can be analyzed well using the HDFS data set. Since only the abnormal test data set contains short log sequences smaller than the window size, padding is only applied here. The splitting of the data set and the reduction of the analyzed abnormal test data is shown again in Tab. 6.5. By appending a unique log key, the log sequence is artificially extended in order to be processed with the LSTM. In principle, this results in a trade off. On one hand, artificial lengthening gives better results for the metrics. By attaching a unique log key to the abnormal test data set the AD is very simple and it is like attaching an anomaly label. Therefore the results are not really useful due to padding. On the other hand, without padding, log sequences that are too short would simply be discarded and not fed into the AD process. The small amount of abnormal test data compared to the normal test data indicates an imbalanced data set. The results of this experiment can be seen in Tab. 6.6 and Fig. 6.6.

	Train (normal) sequences	**Test normal sequences**	**Test abnormal sequences**
padding	4,855	553,366	16,838
nopadding	4,855	553,366	10,647

Table 6.5: Splitting of HDFS data set with padding and without padding.

	Accuracy	**Precision**	**Recall**	**F1-Score**
padding	99.66%	95.13%	95.81%	94.45%
nopadding	99.66%	93.07%	89.03%	91.00%

Table 6.6: Comparison between padding and nopadding for LogDeep with HDFS data.

6.2.3 Results of LogDeep (LSTM) experiments with Audit data set

This section presents the results retrieved from the Exp. 5.2.3. The results for a centralized LogDeep approach with ~18 million and 2 million Audit log sequences can be taken from Tab. 6.7. The reason why the metrics for Audit log sequences are so poor is similar to the AE. When analyzing the ~18 million Audit log sequences, we see that 39.56% of the abnormal Audit log sequences also occur in the training set and 28.67% of the abnormal Audit log sequence occur in the normal test set. And for the 2 million Audit log sequences, we see that 39.56% of the abnormal Audit log sequence also occur in the training set and 26.37% of the abnormal Audit log sequence occur in the normal test set. However, we have extended the analysis to not only look at identical log sequences but also to check if abnormal Audit log sequences occur at some arbitrary point in a training or test sequences. This is an important difference to the AE. An Audit log sequence is built up on individual log keys, e.g., syscalls. A practical example could be:

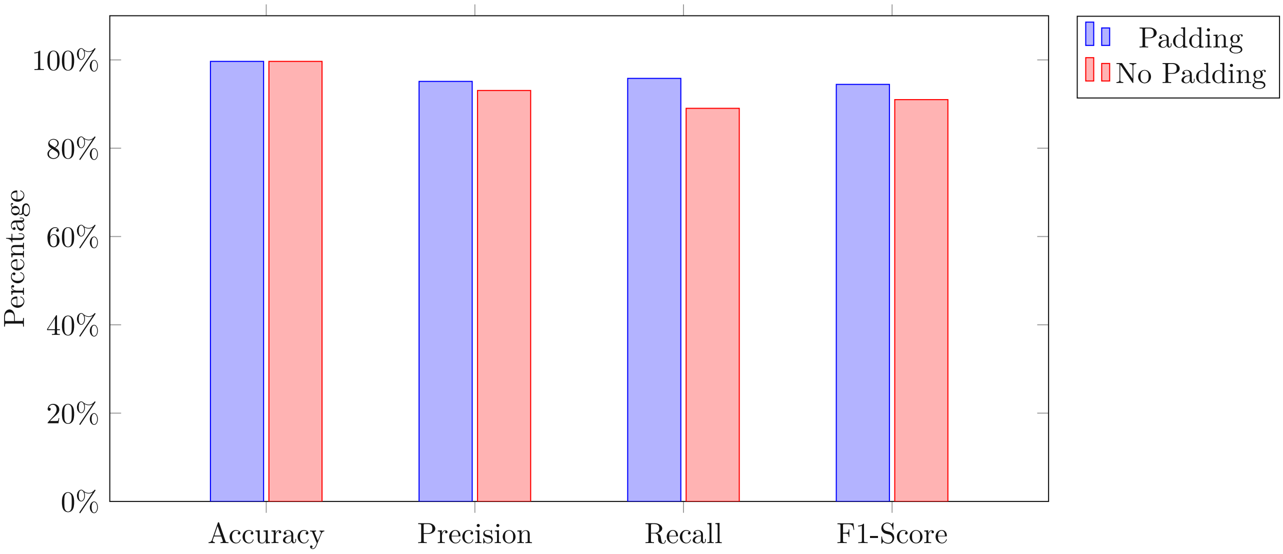


Figure 6.6: Comparison between padding and nopadding for LogDeep with HDFS data.

If we log the behavior of a user, the sequence of syscalls which are executed, is not clear from the start or can also occur in a different order. To continue the example one can assume an attacker, who behaves like a normal user up to a certain point. Thus, the beginning of an Audit log sequence can look exactly the same in the benign as well as in the anomaly case. This makes it difficult to distinguish between benign and malicious log sequences and leads to poor Recall scores and this would be the most important value for accurate AD. One recognizes a limitation here due to the recorded features of Audit logs. Another important point for the Audit data set is that it is an imbalanced data set. As Tab. 5.8 shows we have only 90 malicious log sequences. The low number compared to train and test normal log sequences is the reason for the high Accuracy values.

	Accuracy	**Precision**	**Recall**	**F1-Score**
~18 mil.	99.04%	68.00%	18.88%	29.56%
2 mil.	94.23%	80.49%	36.68%	50.38%

Table 6.7: Comparison between ~18 million and 2 million Audit log sequences with padding.

6.2.4 Results of LogDeep (LSTM) experiments with Audit data set and padding function

This section presents the results retrieved from the Exp. 5.2.4. If the 18 million Audit log sequence data set is not padded, the number of sequences for training is reduced from 33634 to 32822, the number of sequences for test normal from 8409 to 8169 and the number of sequences for test abnormal from 90 to 44. And if the 2 million Audit log sequence data set is not padded, the number of sequences for training is reduced from 4142 to 4030, the number of sequences for test normal from 1036 to 996 and the number of sequences for test abnormal from 90 to 44. The exact breakdown is listed again in Tab. 6.8 and the results for both reduced data sets can be seen in Tab. 6.9. It can be seen that some metrics for both data sets are degraded by nopadding. But the Recall seems to increase. The reason for this is that we have sequences in the abnormal data set that are detected regardless of whether they are padded or not. Those sequences are very long sequences and stand out from the rest. Nopadding reduces the number of processable anomalies by half. This reduction increases the Recall despite the same detection. Nopadding is also not an option because too short log sequences are simply omitted. In our opinion the effect of padding for Audit data is also not as poor as for the HDFS data, because the same unique log key is appended to the train, test normal and test abnormal data sets. After this experiment we decided to use only 2 million of the ~18 million log lines for training and test normal to get results in a certain time and because the analysis, of how many malicious log sequences appear in the training and test normal data set, show similar results. But all anomalies were extracted from the whole data set.

	Train (normal) sequences	**Test normal sequences**	**Test abnormal sequences**
~18 mil. padding	33,634	8,409	90
~18 mil. nopadding	32,822	8,169	44
2 mil. padding	4,142	1,036	90
2 mil. nopadding	4,030	996	44

Table 6.8: Splitting of Audit data set for ~18 million and 2 million Audit log sequences with padding and without padding.

	Accuracy	**Precision**	**Recall**	**F1-Score**
~18 mil. nopadding	99.53%	58.07%	40.91%	48.00%
2 mil. nopadding	95.67%	47.62%	22.7%	30.77%

Table 6.9: Comparison between ~18 million and 2 million Audit log sequences without padding.

6.2.5 Results of LogDeep (LSTM) experiments with Audit data set: Number of epochs

This section presents the results retrieved from the Exp. 5.2.5. The best results of the metrics for AD are achieved after 20 epochs. Before that the model has not learned enough and after that the metrics stagnate or decrease again which can be due to overfitting. Again, we see very poor values for the Recall. This is related to the identical log sequences which originate from the identical log lines in the train, test normal, and test abnormal data set as described in Sect. 6.2.3. The detailed results can be found in Tab. 6.10, and Fig. 6.7 and 6.8.

Epochs	Accuracy	Precision	Recall	F1-Score
5	81.97%	24.20%	58.88%	34.30%
10	91.39%	43.14%	24.44%	31.21%
15	92.72%	61.11%	24.44%	34.92%
20	**94.23%**	**80.49%**	**36.68%**	**50.38%**
25	94.05%	84.84%	31.15%	45.52%
30	94.13%	85.29%	32.22%	46.77%
35	93.60%	80.00%	26.66%	40.00%

Table 6.10: Variation of number of epochs for LogDeep with Audit data.

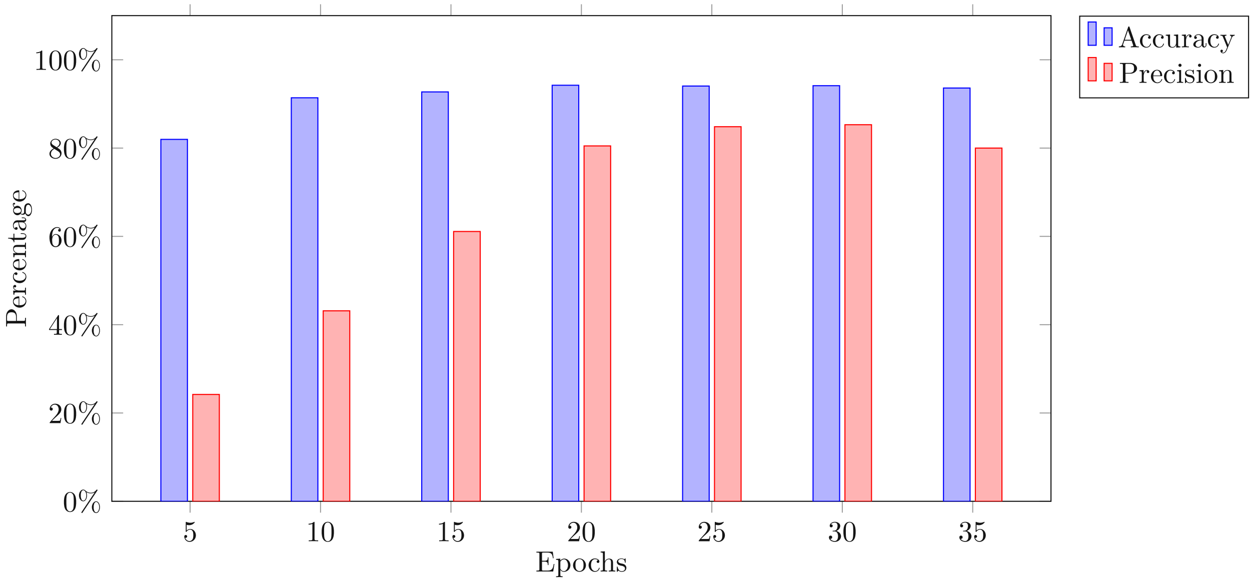


Figure 6.7: Effect on Accuracy and Precision for changing epoch number for LogDeep with Audit data.

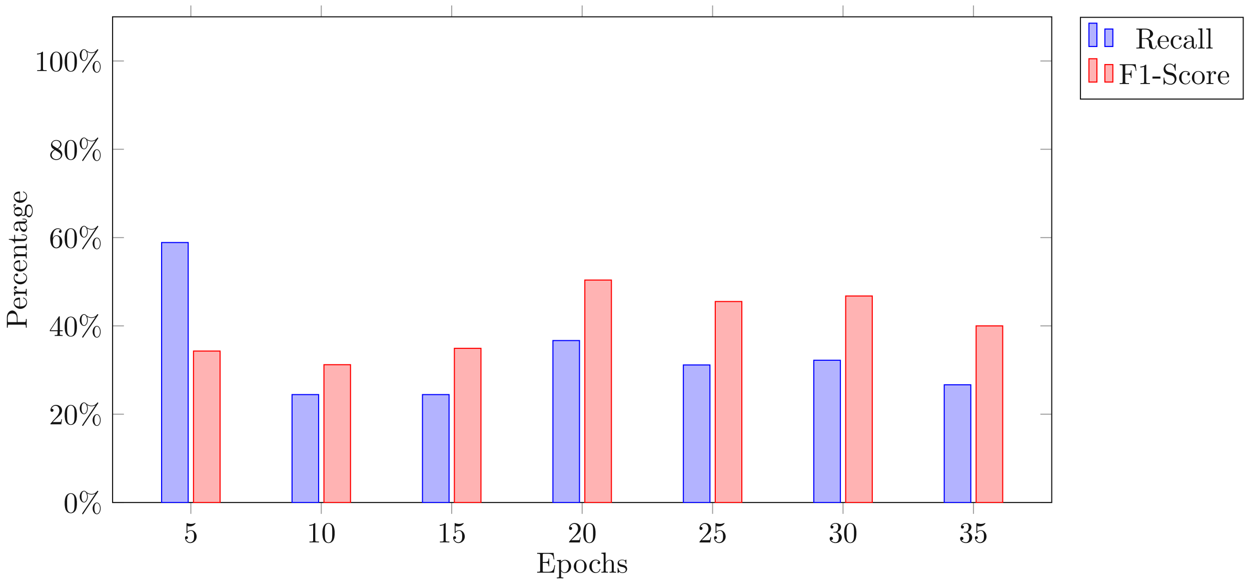


Figure 6.8: Effect on Recall and F1-Score for changing epoch number for LogDeep with Audit data.

6.2.6 Results of LogDeep (LSTM) experiments with Audit data set: Number of candidates

This section presents the results retrieved from the Exp. 5.2.6. The Recall is best if only one candidate is used. As the Recall indicates the probability that an anomaly is detected this would be desirable, but when we look at the Precision when using just one candidate it shows that we must expect many false positives. As a middle way between high detection of anomalies and less false positive alarms, the number of candidates with 6 is suitable. The detailed results can be found in Tab. 6.11, and Fig. 6.9 and 6.10.

Number of candidates	Accuracy	Precision	Recall	F1-Score
1	29.31%	8.66%	82.22%	15.68%
2	82.59%	29.29%	83.33%	45.35%
3	89.07%	38.92%	64.44%	48.54%
4	90.41%	42.23%	54.46%	47.57%
5	94.22%	74.52%	42.22%	53.90%
6	**94.23%**	**80.49%**	**36.68%**	**50.38%**
7	92.95%	82.35%	15.56%	26.17%

Table 6.11: Variation of number of candidates for LogDeep with Audit data.

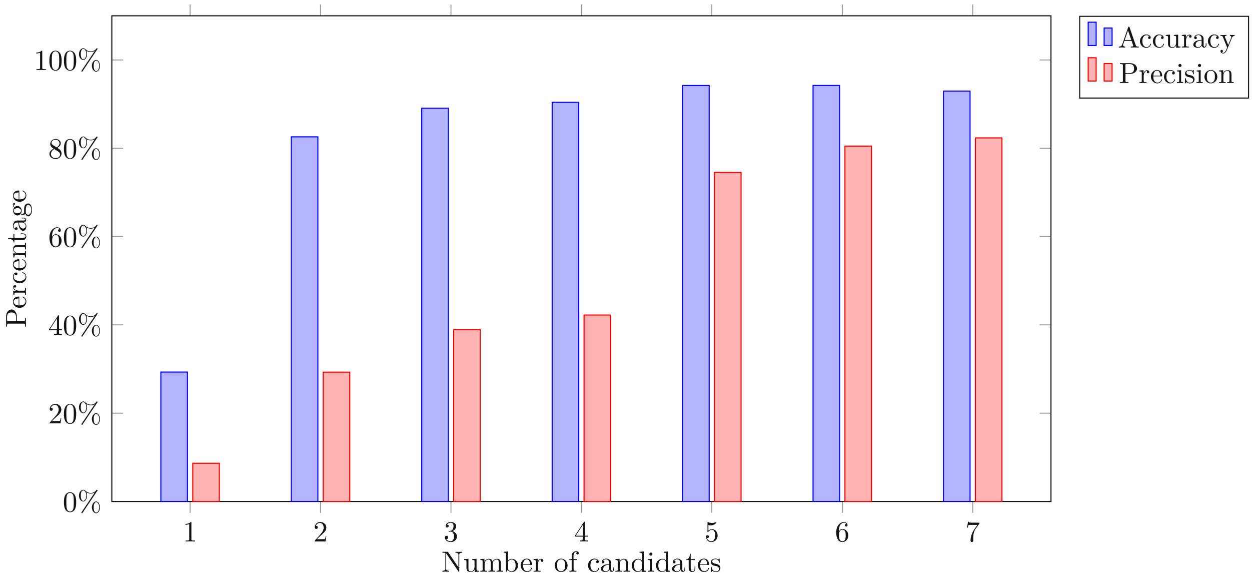


Figure 6.9: Effect on Accuracy and Precision for changing number of candidates for LogDeep with Audit data.

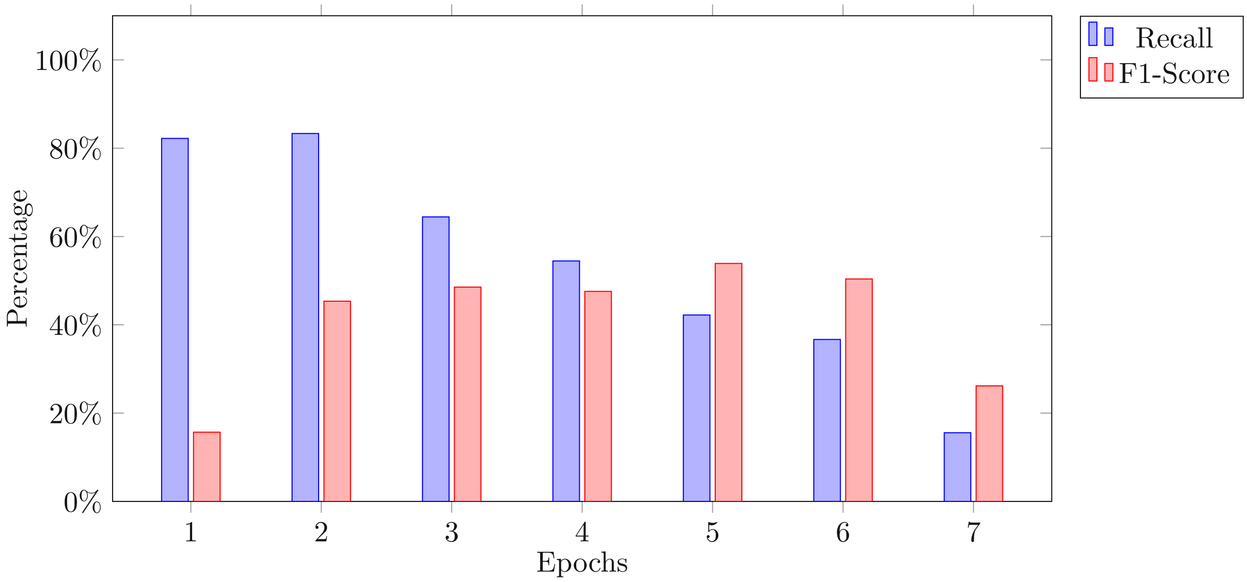


Figure 6.10: Effect on Recall and F1-Score for changing candidates number for LogDeep with Audit data.

6.2.7 Results of LogDeep (LSTM) experiments with Audit data set: Length of window

This section presents the results retrieved from the Exp. 5.2.7. For the window size, the best metrics are obtained with a length of 10. The detailed results can be found in Tab. 6.12, and Fig. 6.11 and 6.12. This value is a good middle ground between too short window sizes, which would result in a too short view into the past, and too long window sizes, which would increase the training time without improving the metrics much.

Window size	Accuracy	Precision	Recall	F1-Score
3	86.41%	32.97%	67.78%	44.36%
4	91.12%	45.97%	63.33%	53.27%
5	92.18%	50.90%	62.22%	56.00%
6	91.65%	48.15%	57.77%	52.52%
7	91.38%	46.73%	55.55%	50.76%
8	93.69%	65.07%	45.55%	53.22%
9	92.63%	56.86%	32.22%	41.14%
10	**94.23%**	**80.49%**	**36.68%**	**50.38%**
11	92.45%	60.00%	16.66%	26.08%
12	92.10%	58.47%	15.55%	25.25%

Table 6.12: Variation of length of window for LogDeep with Audit data.

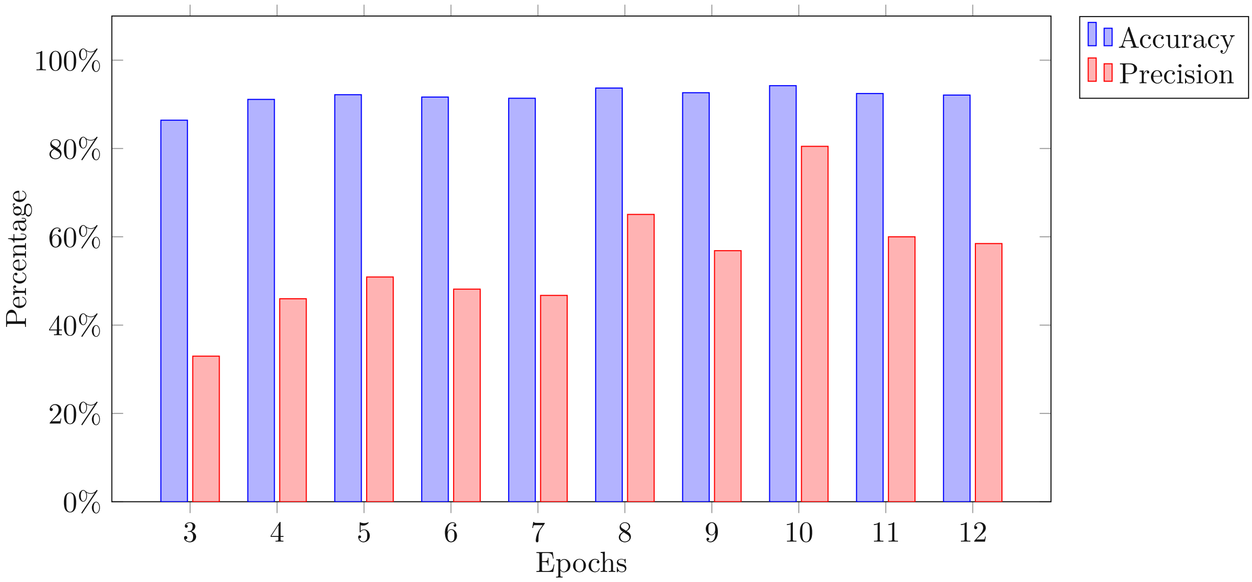


Figure 6.11: Effect on Accuracy and Precision for changing length of window for LogDeep with Audit data.

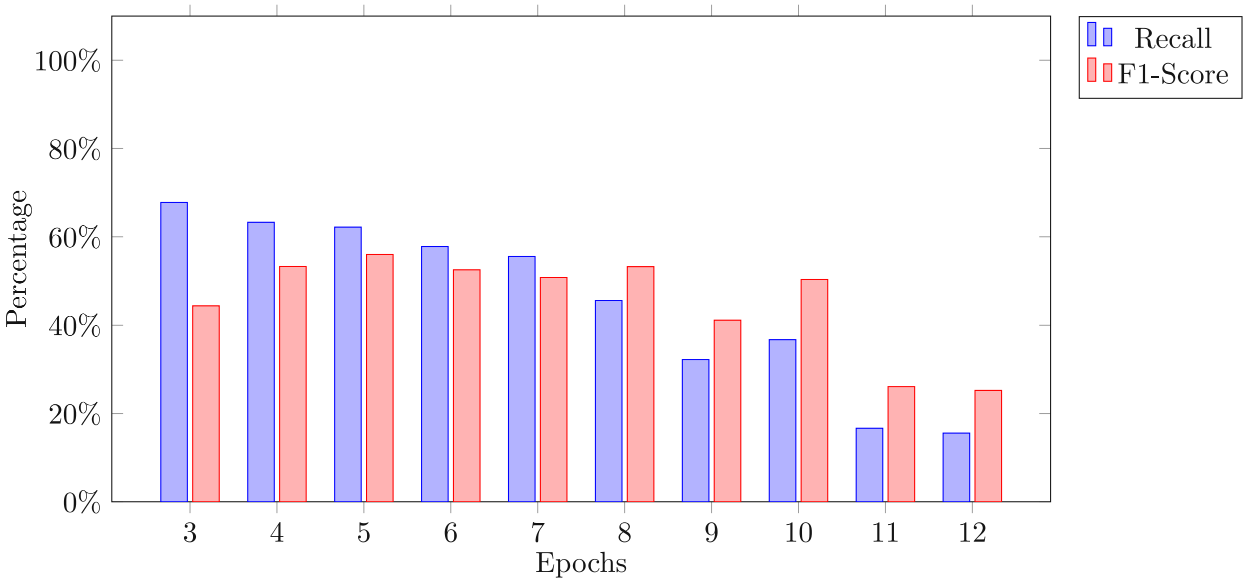


Figure 6.12: Effect on Recall and F1-Score for changing length of window for LogDeep with Audit data.

6.3 Results of Federated learning experiments

This section presents the results retrieved from the FL experiments for the HDFS data set and Audit data sets.

6.3.1 Results of Federated learning experiments with HDFS data set

The effect of FL is not yet clearly visible in Exp. 5.3.1. With the training data set evenly and unevenly distributed over 5 clients, almost the same performance can be achieved as with the centralized implementation. The comparison between centralized and FL implementation is shown in Tab. 6.13, where uneven and even splitting of the training data set are listed separately.

	Accuracy	**Precision**	**Recall**	**F1-Score**
LogDeep (Centralized Sect. 6.2.1)	99.76%	95.63%	96.35%	95.99%
LogDeep (FL) even splitted	99.59%	92.21%	94.22%	93.16%
LogDeep (FL) uneven splitted	99.65%	94.25%	93.73%	93.99%

Table 6.13: Comparison between LogDeep(Centralized) and LogDeep(FL).

6.3.2 Results of Federated learning experiments with even splitted HDFS data set

This section presents the results retrieved from the Exp. 5.3.2. The results of the metrics for even splitting of training data set for 5 clients and 3 rounds can be seen in the Fig. 6.13,6.14,6.15 and 6.16. On the x-axis the respective rounds are shown where the letter L indicates a local evaluation after training phase and letter G indicates a global evaluation after the updates from all clients have been aggregated.

The accuracy which is shown in Fig. 6.13 is already from the first round on a very high level of almost 98% and then seems to stagnate. The reason for this is the imbalanced HDFS data set as explained in Sect. 6.2.2. The precision increases continuously as Fig. 6.14 shows because the false positive log sequence decreases over the rounds. The values for the recall are also already from Round 1 on a high level as Fig. 6.15 shows because of the padding function. The reason for using the padding function is to be able to compare the results in a FL environment with the results of a central implementation from Paper [DLZS17].

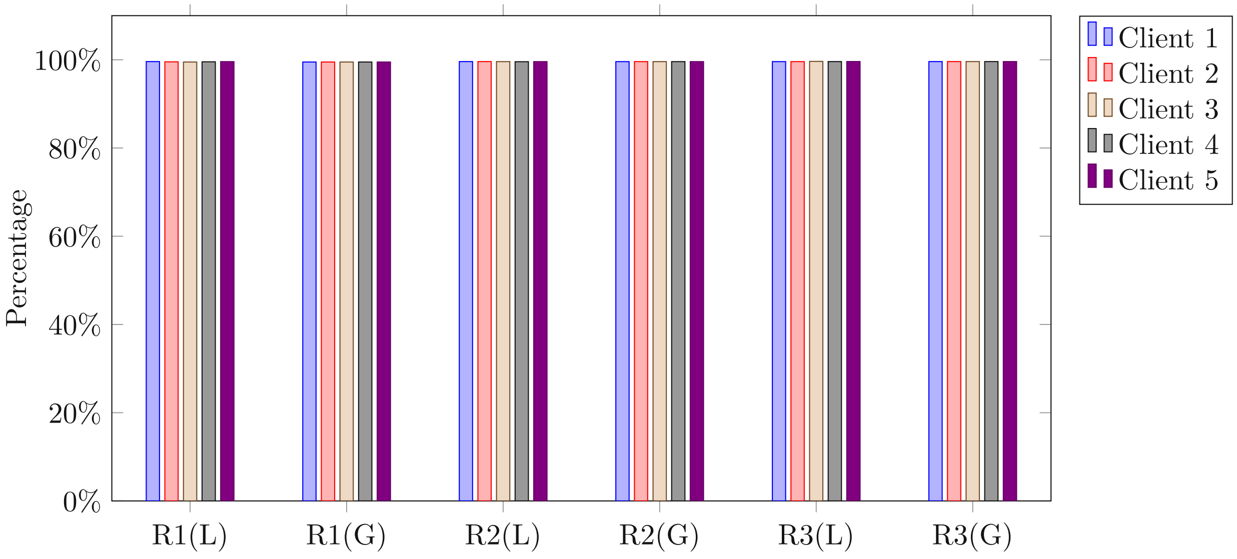


Figure 6.13: Accuracy for 5 Clients with even distributed HDFS data set.

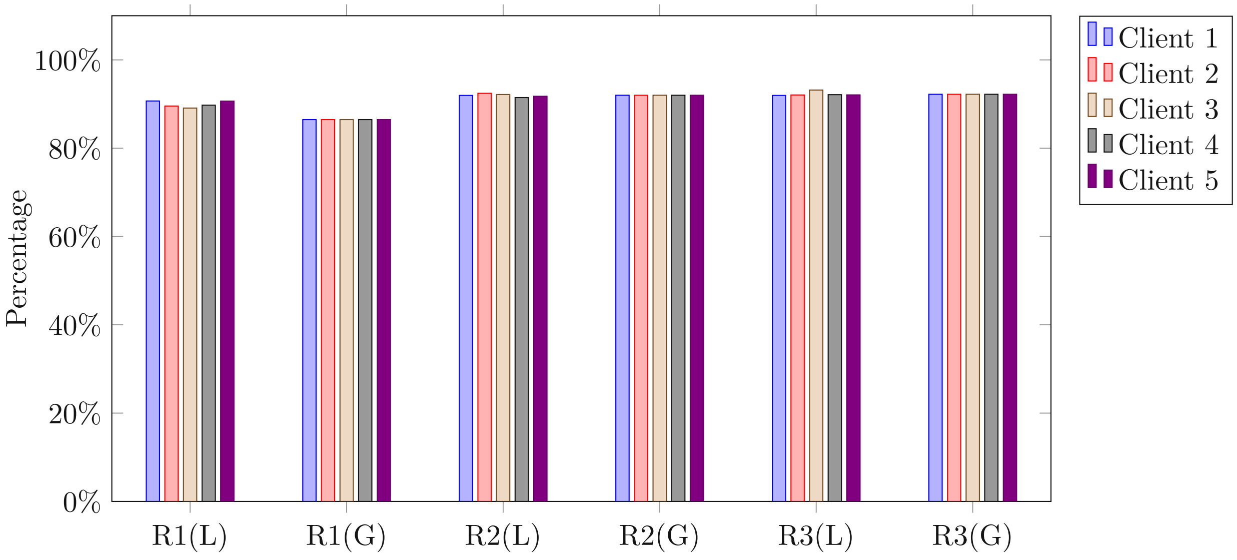


Figure 6.14: Precision for 5 Clients with even distributed HDFS data set.

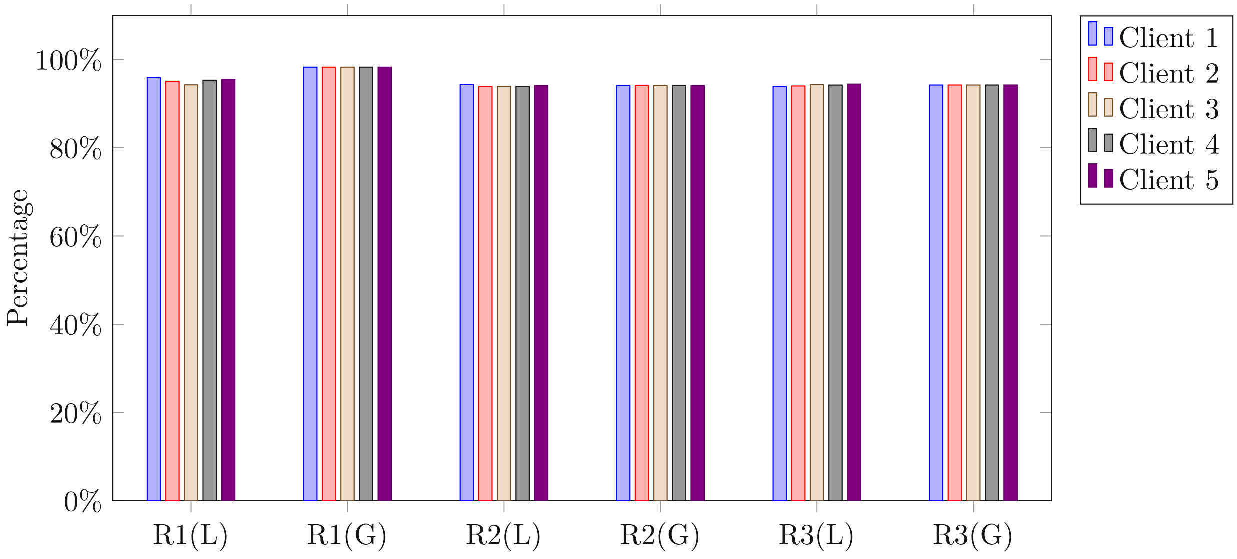


Figure 6.15: Recall for 5 Clients with even distributed HDFS data set.

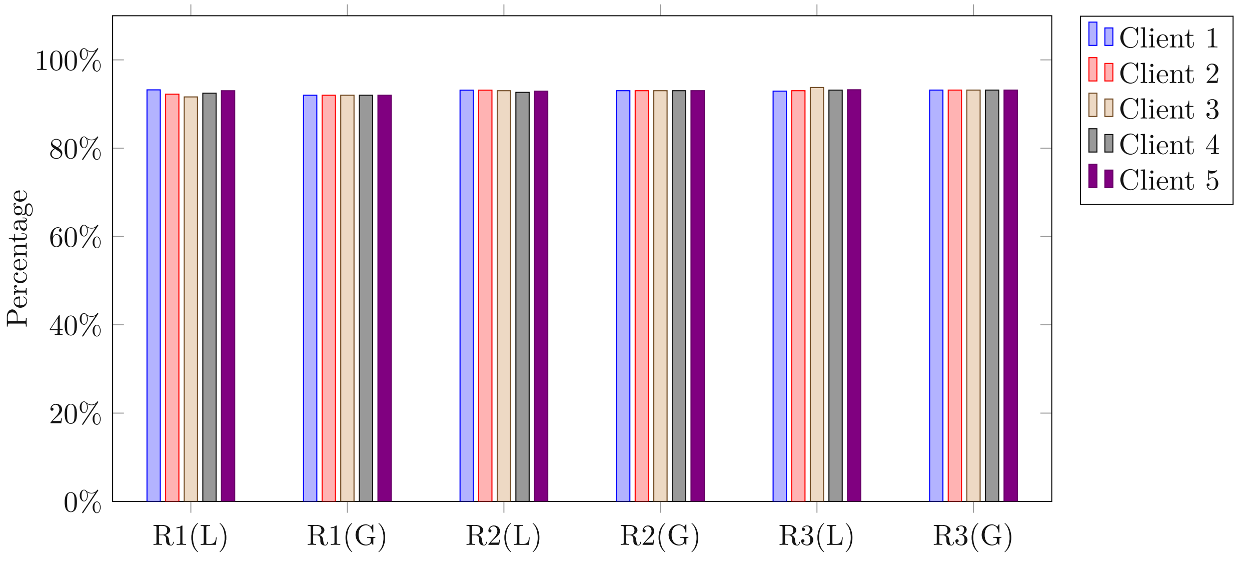


Figure 6.16: F1-Score for 5 Clients with even distributed HDFS data set.

6.3.3 Results of Federated learning experiments with uneven splitted HDFS data set

This section presents the results retrieved from the Exp. 5.3.3. The first signs that FL has a useful effect can be seen in the following results, where training data is unevenly distributed as shown in Tab. 5.10 among 5 clients. The clients with the lowest percentage of training data sets first achieve not so good results and then can improve the performance due to collaborative learning. The results of the metrics for 5 clients and 3 rounds can be seen in the Fig. 6.17,6.18,6.19 and 6.20.

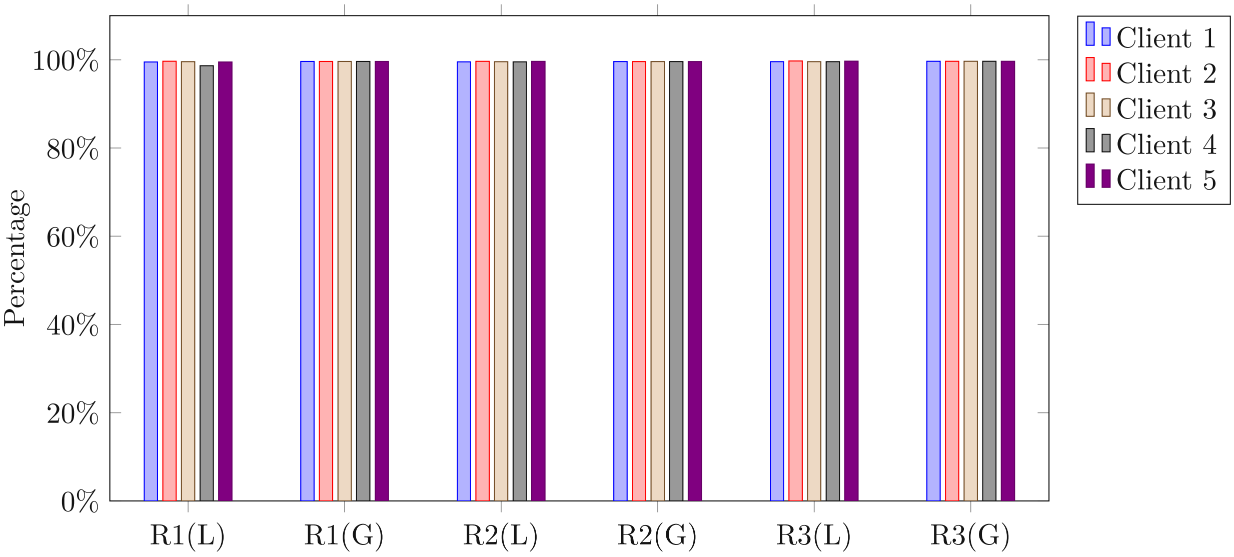


Figure 6.17: Accuracy for 5 Clients with uneven distributed HDFS data set.

The accuracy is again high from the start as Fig. 6.17 shows because of the imbalanced data set as explained in the Sect. 6.3.2. Figure 6.18 shows for Client 4 a Precision of only arround 70%. This could be due to the fact that Client 4 only receives 5% of the train sequences. Client 1 also received only 5% of the train sequences but due to the random splitting of the HDFS train dataset it is possible that LSTM from Client 1 simply received a more significant portion in terms of AE than Client 4. The values for the Recall are also already from Round 1 on a high level as Fig. 6.19 shows because of the padding function.

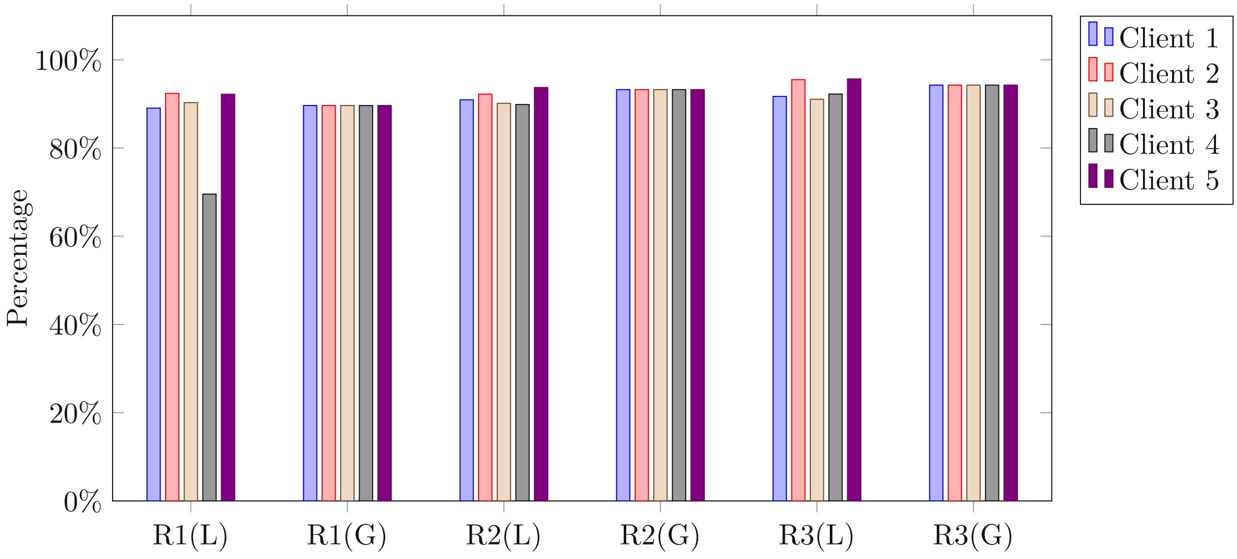


Figure 6.18: Precision for 5 Clients with uneven distributed HDFS data set.

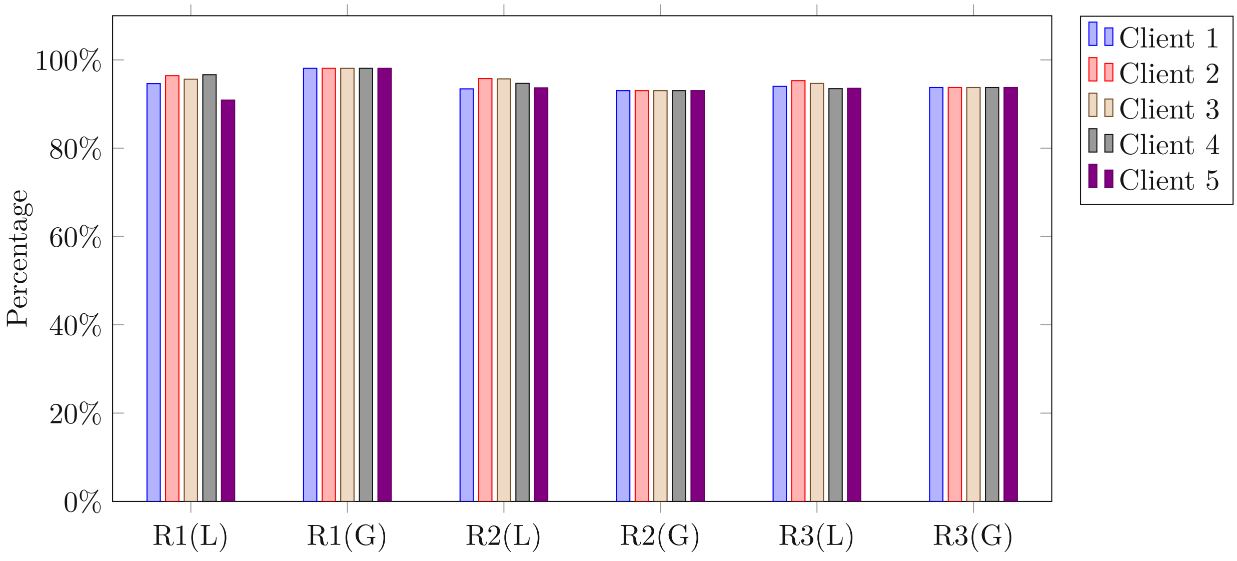


Figure 6.19: Recall for 5 Clients with uneven distributed HDFS data set.

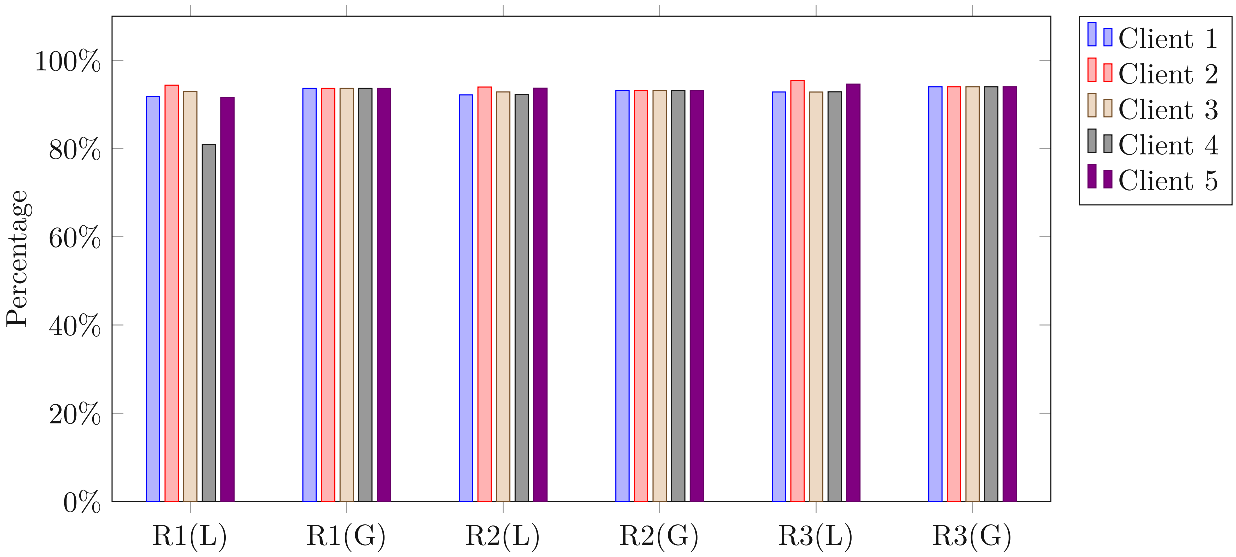


Figure 6.20: F1-Score for 5 Clients with uneven distributed HDFS data set.

6.3.4 Results of Federated learning experiments with HDFS data set: Number of epochs

This section presents the results retrieved from the Exp. 5.3.4. In this experiment, the effect of FL is seen most clearly. By reducing the epoch, the model still converges very quickly. A comparison between 300 and 3 epochs can be seen in Tab. 6.14. Nearly equal metrics can be obtained. The reduced precision comes from increased false positive alarms, but even more anomalies can be detected. The results of the metrics for 5 clients and 3 rounds with 3 epochs of training can be seen in the Fig. 6.21,6.22,6.23 and 6.24.

	Accuracy	**Precision**	**Recall**	**F1-Score**
LogDeep (FL) 300 Epochs	99.59%	92.21%	94.22%	93.16%
LogDeep (FL) 3 Epochs	99.54%	87.15%	99.17%	92.77%

Table 6.14: Comparison between LogDeep(FL) 300 Epochs and LogDeep(FL) 3 Epochs.

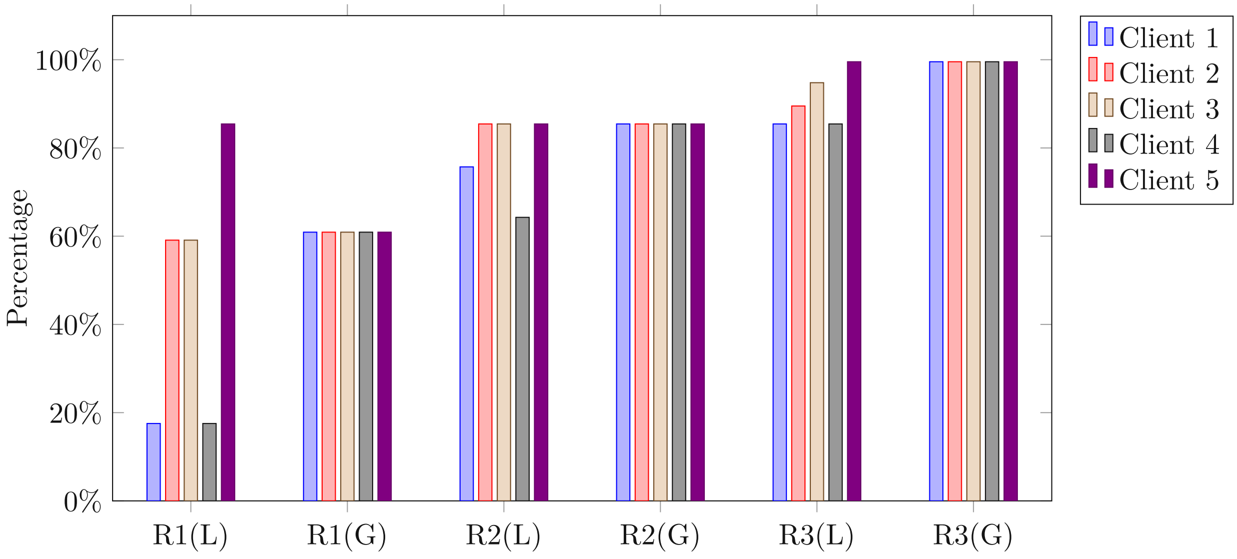


Figure 6.21: Accuracy for 5 Clients with uneven distributed HDFS data set and 3 epochs training.

Figures 6.21, 6.22, and 6.24 show the effect of uneven splitting very clear. It can be seen quite exactly the distribution of train sequences in percentages as in Tab. 5.10 to the individual Clients in the first round of the local calculation of the model. Compared to the results from Sect. 6.3.2 and 6.3.3, the reduction of the epochs leads to a slower increase for the Accuracy and Precision. The recall as shown in Fig. 6.23 remains at a high level from the beginning because of the padding function.

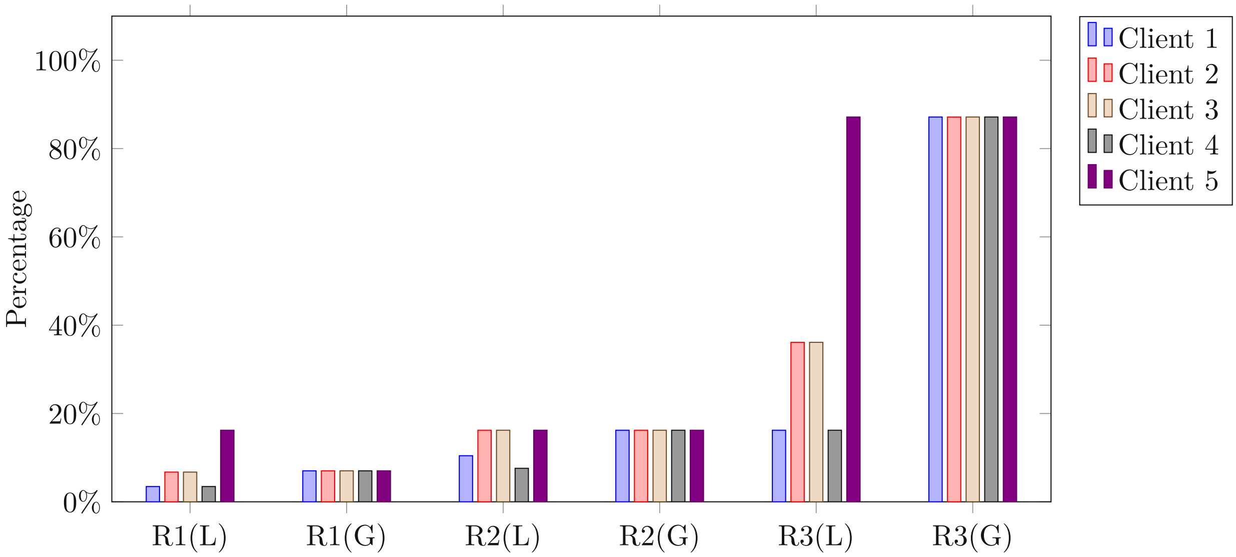


Figure 6.22: Precision for 5 Clients with uneven distributed HDFS data set and 3 epochs training.

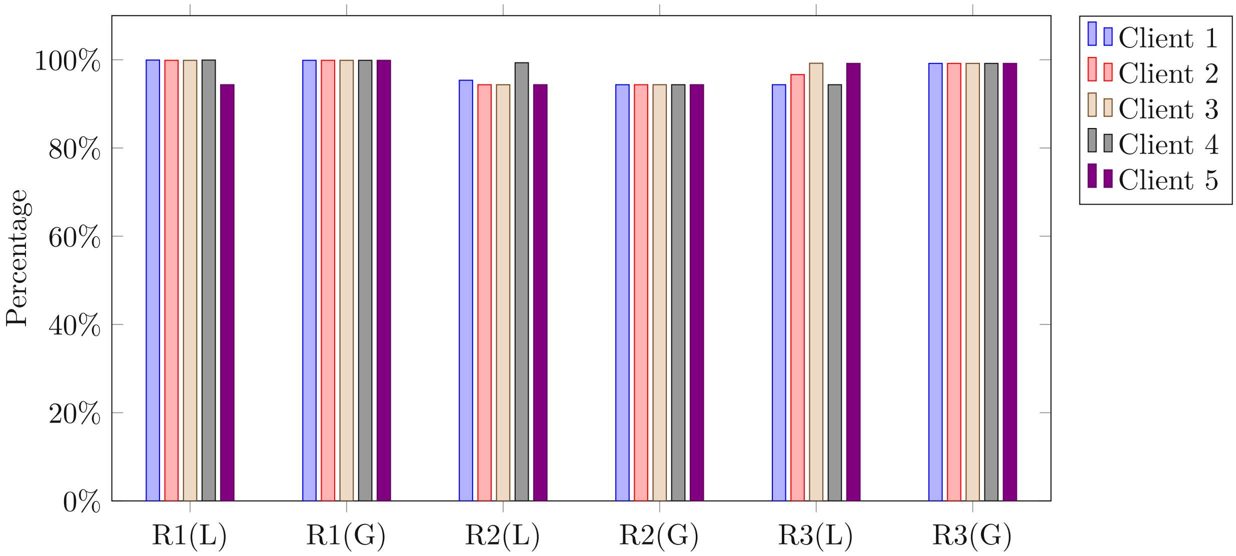


Figure 6.23: Recall for 5 Clients with uneven distributed HDFS data set and 3 epochs training.

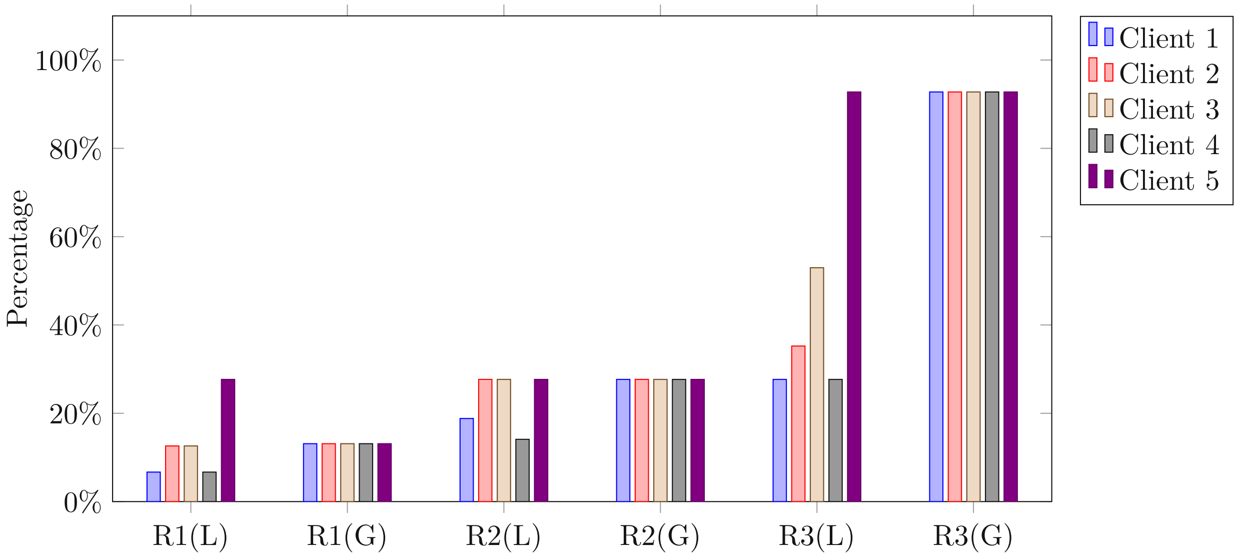


Figure 6.24: F1-Score for 5 Clients with uneven distributed HDFS data set and 3 epochs training.

6.3.5 Results of Federated learning experiments with Audit data set

This section presents the results retrieved from the Exp. 5.3.5. With the training data set evenly and unevenly distributed over 5 clients, similar metrics can be achieved. The comparison between centralized and FL implementation is shown in Tab. 6.15, where uneven and even splitting are listed separately. For AD, a high Recall is desirable, because this metric indicates the probability that an anomaly is recognized as such. Therefore, a centralized approach would be favored. The reason for the poor values of the Recall is explained in Sect 6.2.3 and also applies in a FL environment. A generalized statement as to whether FL is suitable for Audit data cannot be made definitively due to the limitations of Audit data discussed above.

	Accuracy	**Precision**	**Recall**	**F1-Score**
LogDeep (Centralized Sect. 6.2.3)	94.23%	80.49%	36.68%	50.38%
LogDeep (FL) even splitted	93.16%	78.26%	20.00%	31.86%
LogDeep (FL) uneven splitted	93.43%	83.33%	22.22%	35.09%

Table 6.15: Comparison between LogDeep(Centralized) and LogDeep(FL) for Audit data.

6.3.6 Results of Federated learning experiments with even splitted Audit data set

This section presents the results retrieved from the Exp. 5.3.6. The results of the metrics for even splitting of training data set for 5 clients and 3 rounds can be seen in the Fig. 6.25,6.26,6.27 and 6.28. On the x-axis the respective rounds are shown where the letter L indicates a local evaluation after training phase and letter G indicates a global evaluation after the updates from all clients have been aggregated.

The Accuracy is at a high level from the beginning as Fig. 6.25 shows because of the imbalanced Audit data set with only 90 malicious log sequences. The Precision increases continuously as Fig. 6.26 shows because the false positive log sequence decreases over the rounds. Recall remains at a poor level and the explanation for this is described in Sect. 6.3.5.

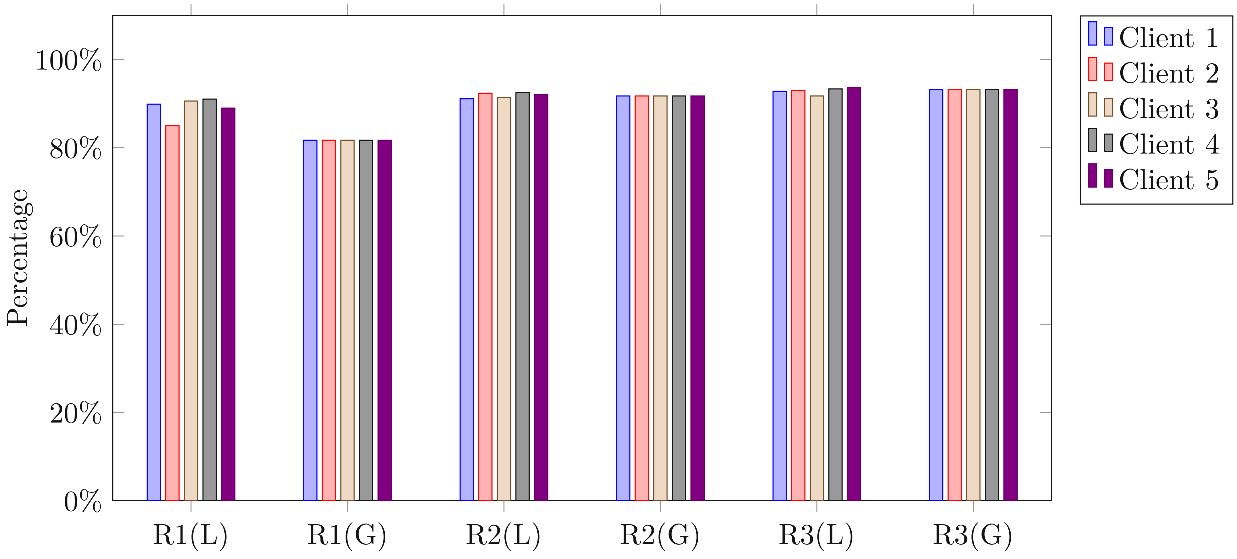


Figure 6.25: Accuracy for 5 Clients with even distributed Audit data set.

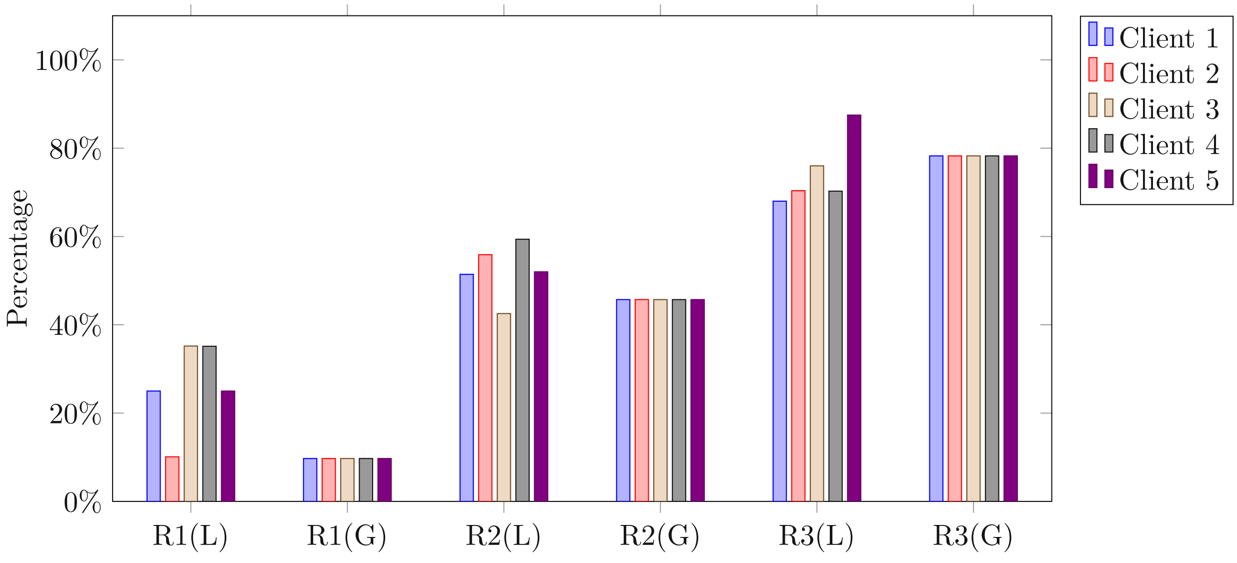


Figure 6.26: Precision for 5 Clients with even distributed Audit data set.

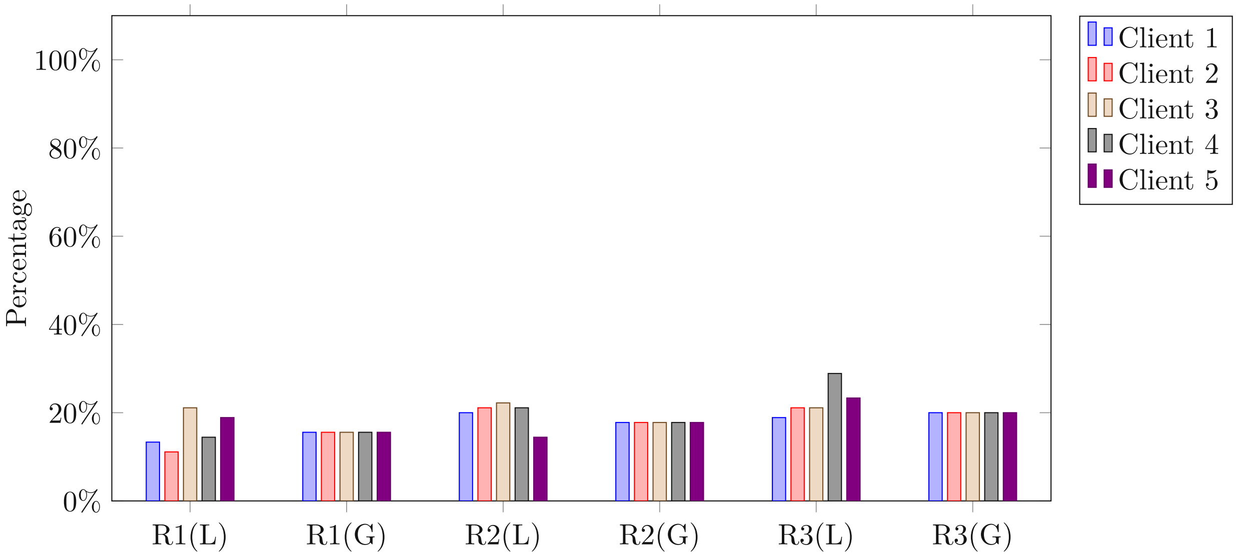


Figure 6.27: Recall for 5 Clients with even distributed Audit data set.

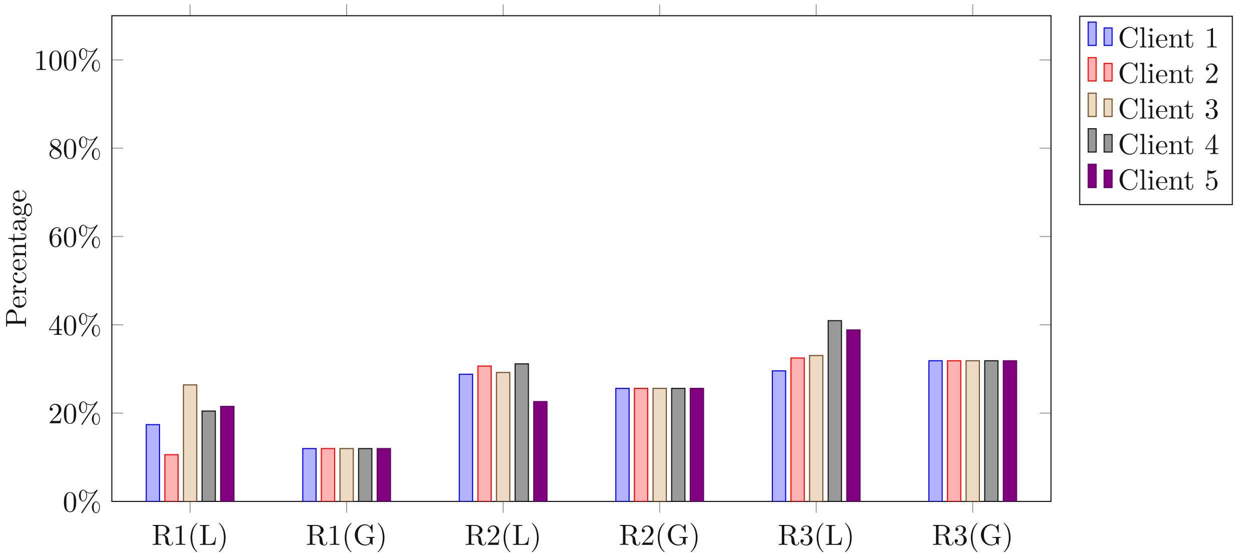


Figure 6.28: F1-Score for 5 Clients with even distributed Audit data set.

6.3.7 Results of Federated learning experiments with uneven splitted Audit data set

This section presents the results retrieved from the Exp. 5.3.7. The results of the metrics for uneven splitting of training data set for 5 clients and 3 rounds can be seen in the Fig. 6.29,6.30,6.31 and 6.32. The precision metric shows that the client with the largest training data set dominates at least in the first 2 rounds. Furthermore, the best Precision is achieved with uneven splitting. Unfortunately, the AD indicated by the Recall is again poor. Basically, it can be stated again that FL leads to the situation that even clients with small data sets can achieve comparable metrics as clients that have training data sets 10 times as large.

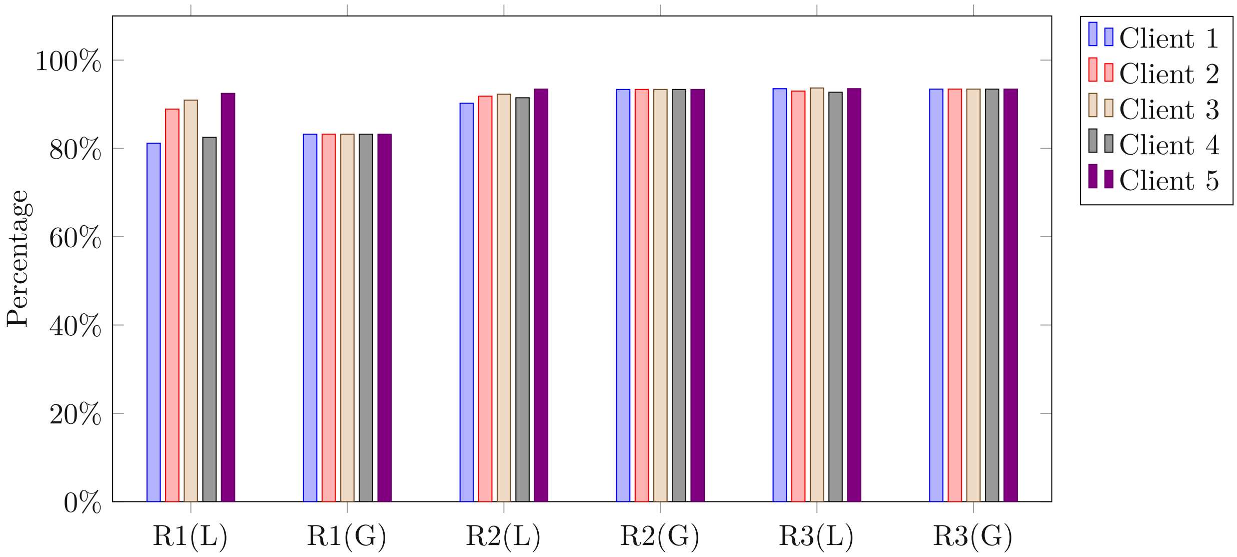


Figure 6.29: Accuracy for 5 Clients with uneven distributed Audit data set.

The values for Accuracy in round 1 as shown in Fig. 6.29 reflect the uneven distribution from Tab. 5.12 very well. Precision increases from round to round and the explanation for this is described in Sect. 6.3.6 and is independent of splitting. The reason for the poor values of the Recall is explained in Sect 6.2.3 and also applies in a FL environment. Interesting is that the Recall in the first round is highest for Client 4 even though it has only 5% of the data set. The reason for this is that Client 4 flags many anomalies in the first round but these are actually false positives which reduces the Recall over the rounds.

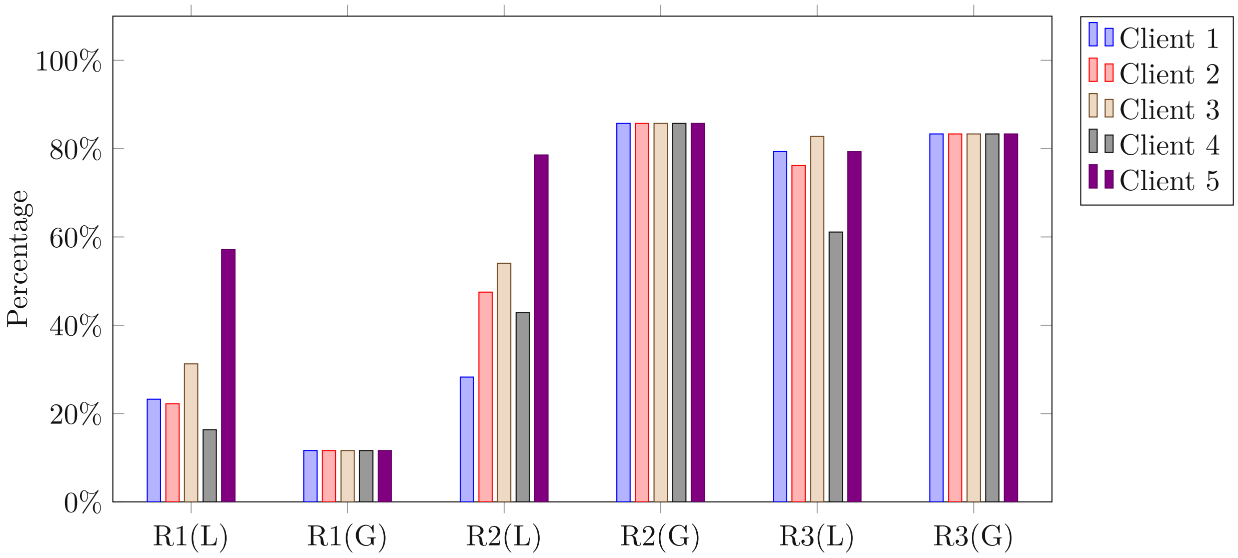


Figure 6.30: Precision for 5 Clients with uneven distributed Audit data set.

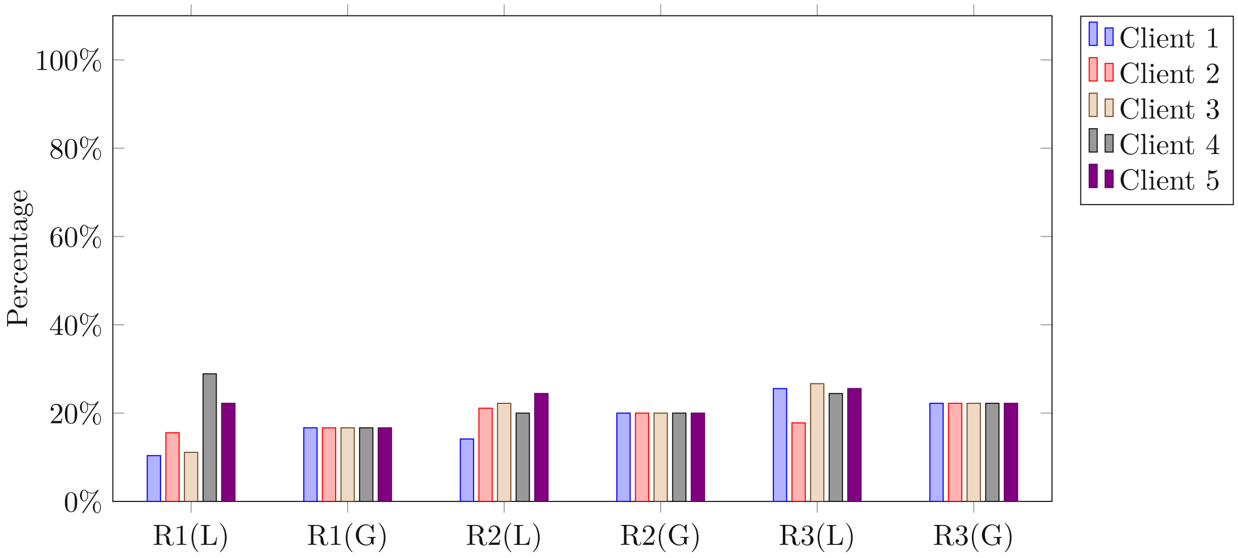


Figure 6.31: Recall for 5 Clients with uneven distributed Audit data set.

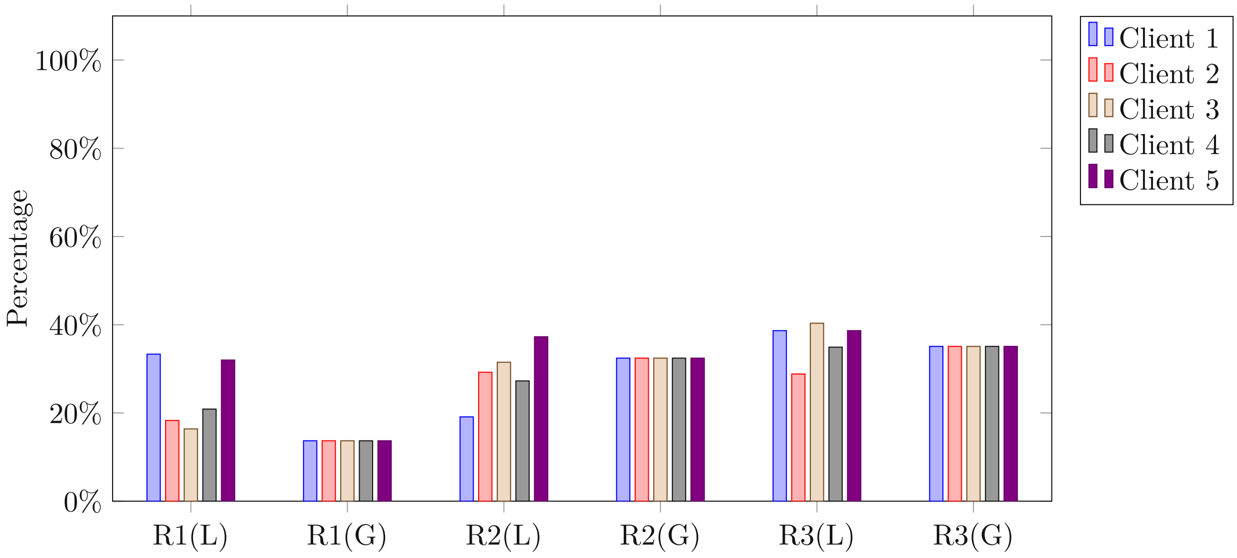


Figure 6.32: F1-Score for 5 Clients with uneven distributed Audit data set.

CHAPTER 7

Discussion

With this master thesis we tried to show the advantages of implementing DL models for AD with FL and to answer the following research questions:

- R1: Which DL algorithm, AE or LSTM, is more suitable with regard to Accuracy, Precision, Recall and F1-Score for the Audit data sets in the domain of AD?
- R2: Can a FL AD implementation reach the same Recall as a centralized DL AD for a well-studied HDFS and an Audit data set?
- R3: How fast do DL models for AD converge with regard to the Recall when FL is used in comparison to a centralized DL AD approach?
- R4: What are the open challenges for integrating FL for AD?

In the beginning we have investigated a simple approach using the AE to detect point anomalies in the Audit [LSW$^+$21] data set. This requires extensive knowledge of how Audit log lines are constructed to select the most meaningful features. If no feature reduction is performed, the number of dimensions and the number of Audit log lines exceeds the regular processing capabilities of the AE. But even by reducing the features of Audit log lines, good metrics for AD are not guaranteed. The main problem is that when creating the Audit data set, sometimes identical log lines were labeled differently. They were labeled as malicious if they occurred within a time when an attack was active. So the difference in how such a log line was labeled is based on a different context which makes an analysis of individual log lines very difficult. Therefore the Audit data set is not usable for the application of an AE. As results in Sect. 6.1.1 and 6.1.2 show, an AE can only work as good as its input data. Specifically, if the AE learns Audit log lines in the training data that occur identically in the abnormal test data set, it is unlikely that these log lines will be detected as anomalies. We can also extend this statement, if we want to detect contextual anomalies to LSTM models in general and our LogDeep implementation as the results in Sect. 6.2.3 show. Here we have chosen the syscall feature as log key. Experiments have shown that due to the small number of distinct values for the syscall and again the occurrence of identical log sequences in the train, test normal and test abnormal data set a precise AD is not possible as shown in Sect 6.2.3. Another important point for the Audit data set is that it is an imbalanced data set. As Tab. 5.3 for the AE and Tab. 5.8 for LSTM show. The Audit data set consists of a lot of benign log lines but very few malicious log lines.

To overcome the limitation of the Audit data set, further research is necessary. For the investigated Audit data set, the multiple occurrences of log lines in the training- and test data set cannot be avoided, because attackers partly behave the same way as normal users. That means up to a certain point the behavior looks identical to the logging system, because the same commands are executed. It would be necessary to improve the logging fundamentally and to collect more features. In this master thesis we decided not to simply discard the multiple occurrences of Audit log lines just to achieve better metrics. Simply deleting these multiple occurrences would falsify the existing Audit data set.

Apart from this limitation, due to the structure of the Audit data set, the variation of the number of epochs for the LSTM model implemented with LogDeep, shows visible effects for the HDFS [XHF$^+$09] data set and Audit data set, which are presented in Sect. 5.2.1 and 5.2.3. Adjustment of the number of epochs has to be done carefully that no over-fitting is introduced. Also the change of number of candidates and length of window have a visible impact on the metrics. The choice of the appropriate number of candidates is a trade off. In our experiments a small number of candidates led to better Recall because it is a middle way between high detection of anomalies and less false positive alarms. A similar trade off occurs with the choice of window size. If we increase the window size, the Precision increases but the Recall decreases. While validating already published scientific results for HDFS data sets processed by a LSTM with LogDeep, we noticed that the authors used a specific padding function. Padding results in an artificial lengthening of log sequences. That means if log sequence are not at least as long as the window size they will not be considered at all and discarded. Especially for the HDFS data set this has strong effects. In the training and test normal HDFS data set there are sufficiently long log sequences. This is important for the applied window size of the LSTM model. In contrast to this, the log sequences in the test abnormal data set are artificially lengthened with a unique log key. For AD it is very simple to detect these log sequences, because they are not visible during training and testing with normal HDFS data sets. Only abnormal log sequences are shorter and padded. Therefore these log sequences could be easily detected. The described effect for the HDFS data set can be seen in from Tab. 6.6. We have also investigated the effects of padding for the Audit data set. In contrast to the HDFS data set, the training- , test normal-, and test abnormal Audit data set have shortened log sequences with respect to the window size. When artificially lengthened with a unique log key, padding has the same effect on all 3 data sets. But even here you can see an improvement when padding is used. The results are shown in Tab. 6.7 and Tab. 6.9.

The use of the padding function has also led to distorted results as shown in Sect 6.2.4.

- A1: For the investigated Audit data set, LSTM is better suited as a DL algortihm than AE, because the LSTM approach implemented with LogDeep detects contextual anomalies. The AE experiments unfortunately do not provide interpretable results for point anomalies, because there were limitations in recording the Audit data set.

After discarding AE as DL algorithm for the data sets under investigation, we have settled on LSTM implemented with LogDeep for the remainder of this master thesis. This implementation is the foundation and the central approach for our log-based AD approach. In the next steps, we implemented this central approach in a FL environment. Useful for this was the open source framework flower, which allows a transfer of central approaches into FL approaches. For the experiments we implemented only 5 clients due to limitations in computational resources. Furthermore, we have only examined FedAvg as an aggregation algorithm. Newer aggregation algorithms are currently being researched. These try to get a grip on the problems, which would be, for example, the correct client selection and excluding unsatisfying local models [LMD+22][LPBA22]. For the HDFS data set slightly worse metrics for the FL approach could be achieved compared to a centralized approach. This can be explained by the aggregation algorithm FedAvg where only an averaging of the aggregated updates takes place. The results can be seen in Tab. 6.13. The first noticeable effects can be seen when the HDFS data set is distributed unevenly in size across the 5 clients. This was also one of the advantages we wanted to prove with the implementation of FL. In a scenario where one client holds a lot of log data locally, FL allows other clients to learn a lot from it and adapt their models locally. The same statements can be made for the Audit data set. The results are shown in Tab. 6.15. Again, we achieve slightly worse metrics compared to a central approach. This means that the effect is independent of the data sets we examined.

- A2: Our FL AD implementation cannot achieve the same values for the Recall as a centralized approach. Further research is needed to verify if other aggregation algorithms can achieve better metrics.

For the time measurement, when our models converge the training time with padding of the models was used. The central implementation of LogDeep and the HDFS data set requires 1 hour and 52 minutes for 300 epochs for the training of the LSTM model. For the Audit data set, LogDeep requires for 18 million log sequences 9 hour and 55 minutes for 20 epochs and for 2 million log sequences 53 minutes. Our FL implementation with 5 clients for the HDFS data set, requires for the even splitted data set 4 hours for 300 epochs and 3 rounds, and for the uneven splitted data set 4 hours and 30 minutes. For 2 million even splitted Audit log sequences, the FL implementation requires 4 hours and 11 minutes for 20 epochs and 3 rounds, and 4 hours and 46 minutes for the uneven splitted Audit log sequences.

- A3: Due to the fact that our FL implementation does not achieve the same results as the central implementation, the time for convergence of the models is of secondary importance. Nevertheless, it can be stated that time saving of our FL approach with a number of 5 clients is not yet recognizable. With the size of the investigated data sets, the central implementation is faster than FL implementations with 3 rounds.

With this master thesis we tried to prove some advantages of FL. It is evident that with a heterogeneous distribution of log data, FL can support clients to learn and benefit from each other without an actual exchange of data sets taking place as shown in Sect. 6.3.3 and 6.3.7. Nevertheless, open challenges remain for further research in this area. It is questionable if the experiments carried out here on a small scale with five clients can be generalized to large-scale problems. As the number of clients continues to grow, it is important to implement better aggregation algorithms than FedAvg. By including clients with superior models, overall performance can be increased. Also mitigating the inclusion of erroneous or intentionally malicious clients will be necessary in the future. In course of this master thesis, however, the still high demand for computational resources was pointed out. This area is also a challenge for future applications, especially in constrained environments such as low-bandwidth networks, or low-powered devices. There is also the disadvantage with FL that the raw data does not have to be sent to a central entity but the updates of the local models still have to be sent. Furthermore, there is an additional effort for the calculation of the final score for the AD.

- A4: The aggregation algorithms and the optimization with respect to computational resources are examples of open challenges we could identify in this master thesis. These issues should be resolved to advance the integration of FL.

CHAPTER 8

Conclusion

The goal of this master thesis was to identify DL algorithms for the investigated HDFS [XHF+09] and Audit [LSW+21] data sets and to integrate them into FL. We succeeded in developing a running implementation for this purpose. Nevertheless, some limitations were identified in the process.

It is imperative to have a basic understanding of how the log data to be analyzed are collected and structured. One of the biggest challenges for AD is to find the right DL method for the use case at hand and its suitability for a given data set. First, we implemented an AE for AD of individual log lines for the Audit [LSW+21] data set provided by AIT. In doing so, we encountered problems that occur during the processing of the audit data set. When recording the audit data set, individual log lines were labelled on the basis of their time occurrences. This means that certain log lines that were recorded during an attack were labelled as malicious. If identical log lines occurred during normal times when no attacks were active, they were labelled as benign. This means that an AE that has only been trained with benign log lines cannot distinguish between identical log lines that are labelled once as malicious and once as benign. We then turned to an LSTM approach that uses log sequences as input. Log sequences are individual log lines that can be grouped based on features such as the pid. We reimplemented a state of the art open source application called LogDeep and were able to replicate the results from paper [DLZS17] which uses a HDFS [XHF+09] data set. However, we found a limitation that occurs due to the padding function that was used in the paper [DLZS17]. Through this function it is possible to change the too short log sequences with respect to the window size important for the LSTM in the abnormal test data set in such a way that a simple AD is possible. The very good detection performance reported in paper [DLZS17] cannot be achieved without the padding function. Afterwards, we adapted LogDeep for the Audit data set. Unfortunately, we could not achieve nearly satisfactory values for the metrics of AD. The padding function described above did not improve the results. One of the reasons for the poor results is that the Audit data set is imbalanced. In this data set there is an overabundance of benign log sequences compared to a small number of malicious log sequences. Furthermore, a large number of log sequences were again found to be identical in the benign and malcious data set. This is again due to the labelling problem described above. With the limitations in mind, we ported our central implementation of LogDeep to a FL environment. There is evidence that FL has a positive impact on the AD of log data, but due to the limitations described above, further scientific work is needed to substantiate this statement.

With this master thesis we wanted to investigate if FL can improve the AD performance. However, we discovered several problems with the two data sets in use. A preprocessing step has altered the HDFS data set which leads to a very simple AD and enables the very good results for the metrics described in the literature [DLZS17]. The Audit data set provided by AIT was labelled based on attack times. After removing the timestamps and extracting the features it contains identical log lines that are labelled as benign and malicious. With this data as basis it is difficult to train a line-based classifier such as an AE. We found out that also identical log sequences occur that are labelled as benign and malicious, making the data unsuitable also for sequence-based classifier such as a LSTM.

Nevertheless, FL could play a major role in future log-based AD systems. The exchange of log data across users, companies or countries etc. could become more important. This would enable AD systems to get a large overview of occurring attacks or unintended behavior. This in turn would contribute to better countermeasures and greater resilience of future systems. With this master thesis we made a contribution to advance the integration of FL.

APPENDIX A

Acronyms

AD	Anomaly Detection
AE	Autoencoder
AIT	Austrian Institute of Technology
BGL	Blue Gene/L
CNN	Convolutional Neural Networks
DT	Decision Tree
DL	Deep Learning
FedAvg	Federated Averaging
FL	Federated Learning
GRU	Gated Recurrent Unit
GPUs	Graphics Processing Units
HDFS	Hadoop Distributed File System
IDS	Intrusion Detection Systems
IM	Invariant Mining
LSTM	Long Short-Term Memory
LCS	Longest Common Subsequence
ML	Machine Learning

PCA	Principal Component Analysis
RAM	Random Access Memory
ReLu	Rectified Linear Unit
RNN	Recurrent Neural Network
RPC	Remote Procedure Call
RMSE	Root Mean Squared Error
SVM	Support Vector Machine

List of Figures

List of Tables

Bibliography

[BB15] Jakub Breier and Jana Branišová. Anomaly Detection from Log Files Using Data Mining Techniques. In Kuinam J. Kim, editor, *Information Science and Applications*, pages 449–457. Springer Berlin Heidelberg, 2015.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[BTM+20] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, and Nicholas D. Lane. Flower: A friendly federated learning research framework, 2020.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, July 2009.

[CC19] Raghavendra Chalapathy and Sanjay Chawla. Deep Learning for Anomaly Detection: A Survey, January 2019.

[CLG+21] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. Experience report: Deep learning-based system log analysis for anomaly detection, 2021.

[DA20] Donglee-Afar. Logdeep. `https://github.com/donglee-afar/logdeep`, 2020.

[DL16] Min Du and Feifei Li. Spell: Streaming Parsing of System Event Logs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 859–864, December 2016.

[DLZS17] Min Du, Feifei Li, Guineng Zheng, and Vivek 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*, pages 1285–1298. ACM, October 2017.

[GWZ+21] Yalan Guo, Yulei Wu, Yanchao Zhu, Bingqiang Yang, and Chunjing Han. Anomaly Detection using Distributed Log Data: A Lightweight Federated

Learning Approach. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2021.

[GYW21] Haixuan Guo, Shuhan Yuan, and Xintao Wu. LogBERT: Log Anomaly Detection via BERT. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2021. ISSN: 2161-4407.

[HWZ+21] Shangbin Han, Qianhong Wu, Han Zhang, Bo Qin, Jiankun Hu, Xingang Shi, Linfeng Liu, and Xia Yin. Log-Based Anomaly Detection With Robust Feature Extraction and Online Learning. *IEEE Transactions on Information Forensics and Security*, 16:2300–2311, 2021. Conference Name: IEEE Transactions on Information Forensics and Security.

[HXY15] Zhiheng Huang, Wei Xu, and Kai Yu. Bidirectional LSTM-CRF Models for Sequence Tagging, August 2015. arXiv:1508.01991 [cs].

[HZHL16] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Experience Report: System Log Analysis for Anomaly Detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, October 2016. ISSN: 2332-6549.

[ITM21] Rei Ito, Mineto Tsukada, and Hiroki Matsutani. An On-Device Federated Learning Approach for Cooperative Model Update between Edge Devices. *IEEE Access*, 9:92986–92998, 2021.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.

[KMDH19] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Samuel Harford. Multivariate lstm-fcns for time series classification. *Neural Networks*, 116:237–245, 2019.

[LCL+19] Suyi Li, Yong Cheng, Yang Liu, Wei Wang, and Tianjian Chen. Abnormal Client Behavior Detection in Federated Learning, December 2019.

[LFL20] Li Li, Yuxi Fan, and Kuo-Yi Lin. A Survey on federated learning. In *2020 IEEE 16th International Conference on Control & Automation (ICCA)*, pages 791–796, October 2020. ISSN: 1948-3457.

[LL19] Hongyu Liu and Bo Lang. Machine learning and deep learning methods for intrusion detection systems: A survey. *Applied Sciences*, 9(20), 2019.

[LMD+22] Beibei Li, Shang Ma, Ruilong Deng, Kim-Kwang Raymond Choo, and Jin Yang. Federated Anomaly Detection on System Logs for the Internet of Things: A Customizable and Communication-Efficient Approach. *IEEE Transactions on Network and Service Management*, pages 1–1, 2022. Conference Name: IEEE Transactions on Network and Service Management.

[LOSW22] Max Landauer, Sebastian Onder, Florian Skopik, and Markus Wurzenberger. Deep Learning for Anomaly Detection in Log Data: A Survey, July 2022. arXiv:2207.03820 [cs].

[LPBA22] Léo Lavaur, Marc-Oliver Pahl, Yann Busnel, and Fabien Autrel. The Evolution of Federated Learning-based Intrusion Detection and Mitigation: a Survey. *IEEE Transactions on Network and Service Management*, pages 1–1, 2022.

[LSW+21] Max Landauer, Florian Skopik, Markus Wurzenberger, Wolfgang Hotwagner, and Andreas Rauber. Have it Your Way: Generating Customized Log Datasets With a Model-Driven Simulation Testbed. *IEEE Transactions on Reliability*, 70(1):402–415, March 2021. Conference Name: IEEE Transactions on Reliability.

[LZXS07] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure Prediction in IBM BlueGene/L Event Logs. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 583–588, October 2007. ISSN: 2374-8486.

[MLZ+19] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. LogAnomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 4739–4745. International Joint Conferences on Artificial Intelligence Organization, August 2019.

[MMR+17] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data, February 2017.

[NMM+19] Thien Duc Nguyen, Samuel Marchal, Markus Miettinen, Hossein Fereidooni, N. Asokan, and Ahmad-Reza Sadeghi. DÏot: A federated self-learning anomaly detection system for iot. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 756–767, 2019.

[OS07] Adam Oliner and Jon Stearley. What Supercomputers Say: A Study of Five System Logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 575–584. IEEE, June 2007.

[PRT+18] Davy Preuveneers, Vera Rimmer, Ilias Tsingenopoulos, Jan Spooren, Wouter Joosen, and Elisabeth Ilie-Zudor. Chained anomaly detection models for federated learning: An intrusion detection case study. *Applied Sciences*, 8(12):2663, 2018.

[PSCH22] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton van den Hengel. Deep Learning for Anomaly Detection: A Review. *ACM Computing Surveys*, 54(2):1–38, March 2022.

[RH] Inc. Red Hat. Understanding audit log files.

[RTTM20] Sawsan Abdul Rahman, Hanine Tout, Chamseddine Talhi, and Azzam Mourad. Internet of Things Intrusion Detection: Centralized, On-Device, or Federated Learning? *IEEE Network*, 34(6):310–317, November 2020. Conference Name: IEEE Network.

[SH20] Johannes Schmidt-Hieber. Nonparametric regression using deep neural networks with ReLU activation function. *The Annals of Statistics*, 48(4), August 2020.

[SH21] Raed Abdel Sater and A. Ben Hamza. A Federated Learning Approach to Anomaly Detection in Smart Buildings. *ACM Transactions on Internet of Things*, 2(4):1–23, November 2021.

[SWJR07] Xiuyao Song, Mingxi Wu, Christopher Jermaine, and Sanjay Ranka. Conditional Anomaly Detection. *IEEE Transactions on Knowledge and Data Engineering*, 19(5):631–645, May 2007.

[SWL21] Florian Skopik, Markus Wurzenberger, and Max Landauer. *Smart Log Data Analytics*. Springer International Publishing, 2021.

[VAS$^+$19] R. Vinayakumar, Mamoun Alazab, K. P. Soman, Prabaharan Poornachandran, Ameer Al-Nemrat, and Sitalakshmi Venkatraman. Deep learning approach for intelligent intrusion detection system. *IEEE Access*, 7:41525–41550, 2019.

[vEAS$^+$22] Thijs van Ede, Hojjat Aghakhani, Noah Spahn, Riccardo Bortolameotti, Marco Cova, Andrea Continella, Maarten van Steen, Andreas Peter, Christopher Kruegel, and Giovanni Vigna. DEEPCASE: Semi-Supervised Contextual Analysis of Security Events. 2022.

[VSP17] R. Vinayakumar, K. P. Soman, and Prabaharan Poornachandran. Long short-term memory based operation log anomaly detection. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 236–242, September 2017.

[WSSF18] Markus Wurzenberger, Florian Skopik, Giuseppe Settanni, and Roman Fiedler. Aecid: A self-learning anomaly detection approach based on light-weight log parser models. *Proceedings of the 4th International Conference on Information Systems Security and Privacy*, pages 386–397, 2018.

[XHF+09] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 117. ACM Press, 2009.

[YB22] Bang Xiang Yong and Alexandra Brintrup. Do Autoencoders Need a Bottleneck for Anomaly Detection? *IEEE Access*, 10:78455–78471, 2022. Conference Name: IEEE Access.

[YKD20] Rakesh Bahadur Yadav, P Santosh Kumar, and Sunita Vikrant Dhavale. A Survey on Log Anomaly Detection using Deep Learning. In *2020 8th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pages 1215–1220, June 2020.

[YLCT19] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated Machine Learning: Concept and Applications, February 2019.

[ZXL+19] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, Junjie Chen, Xiaoting He, Randolph Yao, Jian-Guang Lou, Murali Chintalapati, Furao Shen, and Dongmei Zhang. Robust log-based anomaly detection on unstable log data. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 807–817. ACM, August 2019.