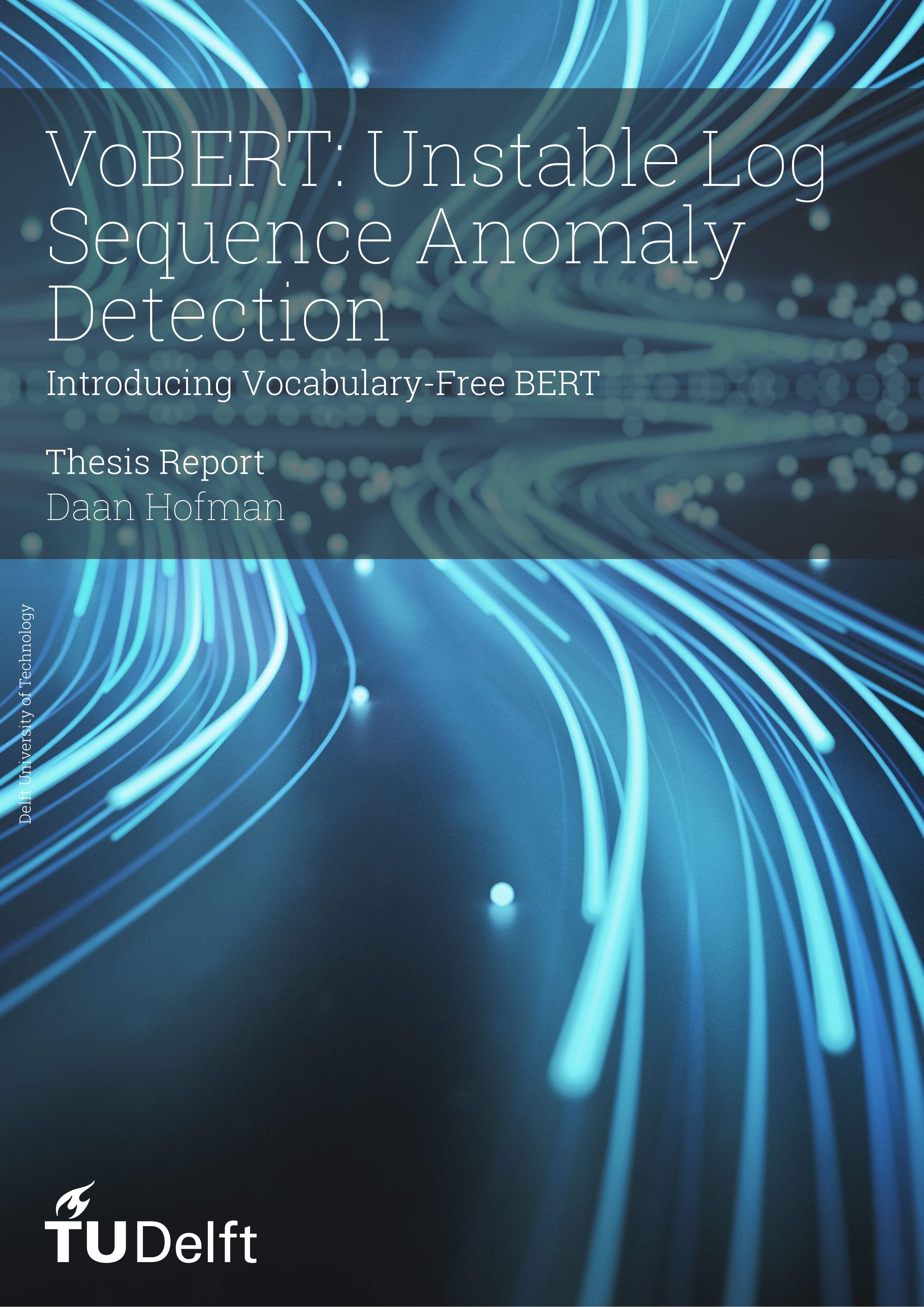


VoBERT: Unstable Log Sequence Anomaly Detection

Introducing Vocabulary-Free BERT

Thesis Report

Daan Hofman

Delft University of Technology

TU Delft

VoBERT: Unstable Log Sequence Anomaly Detection

Introducing Vocabulary-Free BERT

by

Daan Hofman

To obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday July 5, 2023 at 13:30.

Student number:	4688988		
Project duration:	November, 2022 – July, 2023		
Thesis committee:	Dr. A. Lukina	TU Delft	Supervisor
	Dr. Y. Zhauniarovich	TU Delft	Supervisor
	Dr. E. Bárbaro	ING Group	Supervisor
	Dr. M.T.J. Spaan	TU Delft	Advisor
	Ir. S.E. Verwer	TU Delft	Advisor

Preface

In this thesis, I delve into my exploration of artificial intelligence's potential in log analysis. My fascination with artificial intelligence's ability to unravel complex problems guided my journey into the realm of log analysis.

After 312 commits and weeks of continuous experimentation, I am pleased to present my findings. I dedicate this work to fellow researchers, academics, and practitioners who are passionate about utilizing artificial intelligence to enhance the security and resilience of our digital environment.

The journey of writing this thesis was challenging but immensely rewarding. I couldn't have made it without the unwavering support from my mentors, peers, family, and friends. Their faith in my abilities and the relevance of this work served as a steady source of motivation. Special thanks to Dr. A. Lukina, Dr. E. Bárbaro, and Dr. Y. Zhauniarovich for their guidance as my thesis supervisors. Their insights significantly shaped this work. I also express my gratitude to Dr. M.T.J. Spaan and Ir. S.E. Verwer for their contributions as part of my thesis committee. Lastly, my sincere thanks to Mr. drs. C.M. Vaandrager for her constant encouragement throughout this process.

The path to this point has been steep, but the view from here is worth it. I hope that the insights gained from this study will contribute to the field and spark further exploration into the boundless potential of artificial intelligence.

Daan Hofman
Delft, July 2023

Summary

With the ever-increasing digitalisation of society and the explosion of internet-enabled devices with the Internet of Things (IoT), keeping services and devices secure is becoming more important. Logs play a critical role in sustaining system reliability. Manual analysis of logs has become increasingly difficult, accelerating the development of automated methods for log-anomaly detection. Despite significant progress in automating log analysis, current state-of-the-art methods face challenges dealing with unstable log data, which means that the content of log messages evolves over time.

We show that LogBERT [26], a state-of-the-art technique based on Bidirectional Encoder Representations from Transformers (BERT), cannot deal with unstable log data. On the three most prevalent publicly available log datasets, Mathew's Correlation Coefficient (MCC) score (which measures the correlation between a model's output and the correct labels) of LogBERT dropped by 90% after increasing log data instability from 1% to over 80% normal sequences containing logkeys in the test set. Log data instability was increased by only reassigning samples between the train and test set. Furthermore, we show that the high performance of LogBERT reported in the original paper [26] was achieved because the model relied on a simple heuristic that only worked under specific conditions.

To address this issue, we propose a novel sequence anomaly detection technique based on BERT: Vocabulary-Free BERT (VoBERT). VoBERT uses a novel pre-training task we designed specifically for anomaly detection: Vocabulary-Free Masked Language Modeling (VF-MLM). We adapted traditional MLM and removed the fixed vocabulary constraint, which allows VF-MLM to classify out-of-vocabulary logkeys correctly.

We highlight that VoBERT is more stable than LogBERT and outperforms the latter in certain situations where log data is very unstable. For the public datasets, the MCC score of the specific train-test split used in the LogBERT paper dropped by 90% after reassigning the train-test split, increasing log data instability. In addition to sequence-level anomaly predictions, we evaluated all approaches on element level, providing a more granular performance assessment.

To assess the generalisation of the experimental results to real-world scenarios, we conducted a case study evaluating the anomaly detection models on real-world security event data collected at a large bank (50,000+ employees). We found that the simple heuristic did not work for this real-world data, having a negative correlation with the correct results. VoBERT showed performance on par with LogBERT on this real-world security event dataset.

We urge future researchers to evaluate their methods on real-world data, as we showed that the commonly used public datasets do not represent real-world scenarios. Furthermore, it is important to assess how difficult it is to detect anomalies in datasets used for evaluation. When a simple heuristic can perform well, such datasets might not be well suited to evaluate a complex anomaly detection model.

This thesis is a proof of concept for the novel pre-training task VF-MLM and paves the way for future work to refine this technique further, as well as to develop additional robust and adaptable solutions for log and security event anomaly detection.

Contents

1 Introduction

With the ever-increasing digitalisation of society and the explosion of internet-enabled devices with the Internet of Things (IoT), keeping services and devices secure is becoming more important. Software systems, such as extensive data systems and distributed micro-services, generate logs for troubleshooting. Engineers can later use these log files to understand what happened during the execution of software. Often, these logs consist of unstructured text messages used to record events or states of interest [76]. They are critical in sustaining system reliability and uninterrupted functioning [32]. An empirical study on two Microsoft systems and a pair of open-source projects highlighted the widespread application of logging [76]. The study revealed that, on average, for every 58 lines of source code, there exists one line dedicated to logging [76], demonstrating the integral role logging plays in software development and maintenance.

Logs are usually generated by special statements in the source code of a computer program. Logs are the main method to store the most critical information (e.g., states, events) about the execution of software or systems for postmortem analysis [76]. As these systems grow larger and more complex, however, manual analysis of logs has become increasingly difficult, accelerating the development of automated methods for log anomaly detection.

The rapid expansion and escalating complexity of software systems have transformed log anomaly detection into a significant challenge within system maintenance and security. Manually sifting through the vast amounts of log data to detect anomalies is no longer feasible. Moreover, the anomalies are becoming increasingly intricate and subtle, often embedded within a sea of normal behaviour, thereby amplifying the difficulty of manual detection. This increasing complexity demands not only a high level of expertise and understanding but also an enormous amount of time. Therefore, the need for automated, accurate, and efficient log anomaly detection methods is not just desirable but critical in the modern, ever-evolving digital landscape.

There has been tremendous progress in automating log anomaly detection in recent years. Over the years, many machine learning solutions based on feature extraction have been proposed [7, 31, 43, 71]. More recently, advancements in deep learning have led to even better sequential models capable of capturing temporal and contextual information for log anomaly detection [37]. Recent examples include [57, 68, 67, 21, 26, 75].

However, there are still many open challenges in this area, such as the issue of log data instability. Log data instability means that the content of log messages changes over time, messages are removed, or new messages are introduced. The primary contributors to log instability are updates to the system or software that cause alterations to logging statements and noise introduced during log anomaly detection pre-processing steps [75]. The effectiveness of most of the current state-of-the-art methods is significantly limited by the instability of log data [75].

In a study [75] looking at log evolution in a real-world online Microsoft service, it was found that when comparing version 1.0 to version 8.0, the number of changed or new logkeys accounts for 30.3% of the total logkeys. Another study examining the stability of logging statements found that around 20% to 45% of logging statements changed throughout their lifetime [35]. In real-world settings, models can often not be retrained in time to accommodate these changes [75]. This creates the need for a log anomaly detection method that can naturally deal with unstable log data.

Typically, minor modifications to an existing log key can yield a new but semantically similar logkey. Current approaches, however, interpret this as an entirely new log key [21, 31, 43, 71, 26]. That means introducing new, different logkeys will always cause these methods to classify a sequence as anomalous. Consequently, these approaches fail due to incompatibility with unseen logkeys or deliver subpar performance because of incorrect classification [75].

In this thesis, we present VoBERT (Vocabulary-Free BERT), a sequence anomaly detection model based on Bidirectional Encoder Representations from Transformers (BERT) [19]. Transformers are a type of artificial neural network architecture that have revolutionised the field of Natural Language Processing (NLP). They were introduced in 2017 in the paper "Attention is All You Need" [66]. The key innovation of transformers is using self-attention mechanisms, which allow the model to consider the relationship between any two input tokens in a sequence rather than just those immediately adjacent. The architecture of transformers has proven to be very effective in a wide range of NLP tasks, including language translation, text classification, and question answering [66].

VoBERT uses a novel pre-training task we designed specifically for anomaly detection: Vocabulary-Free Masked Language Modeling (VF-MLM). We adapted traditional Masked Language Modelling (MLM) and removed the fixed vocabulary constraint, which allows VF-MLM to classify out-of-vocabulary logkeys correctly. MLM is a pre-training task in Natural Language Processing (NLP) where a model predicts missing words in a sentence. Traditional MLM requires a vocabulary to represent possible output words or word pieces as it is used for NLP tasks. Because VF-MLM is designed for sequence anomaly detection rather than NLP, the vocabulary can be removed since we do not need the model output to be mapped back to natural language. We only need to know how well the trained model can reconstruct the original sequence by predicting the masked logkeys to detect anomalies.

VF-MLM addresses the issue of log data instability by eliminating the reliance on a fixed vocabulary. It leverages the fact that when MLM is used for anomaly detection, no vocabulary is required and takes advantage of recent advances in NLP that allow the logkeys to be semantically embedded. Instead of using a fixed vocabulary, VF-MLM uses a separate sentence encoder model to generate semantic embeddings for logkeys, which allows the model to adapt to new or modified logkeys without retraining. This approach mitigates the problem that existing MLM-based models have: incorrectly labelling sequences with unseen elements as anomalous.

We show that LogBERT [26], a current state-of-the-art technique based on BERT, cannot deal with unstable log data. On the three most frequently used publicly available log datasets, the Mathew Correlation Coefficient (MCC) score of LogBERT dropped by 90% after increasing log data instability. Furthermore, we showed that the high performance of LogBERT reported in the original paper [26] was only obtained because the model relied on a simple heuristic that only worked under specific conditions.

We highlight that our novel approach, VoBERT, using the novel pre-training task VF-MLM, is more stable across varying levels of log data instability when evaluated on the three most commonly used public datasets in this field. In addition to sequence-level anomaly predictions, all anomaly detection methods were evaluated on element level, providing a more granular performance assessment.

To assess the generalisation of the experimental results to real-world scenarios, a case study is conducted evaluating the anomaly detection models on real-world security event data collected at a large bank (50,000+ employees). It was found that the simple heuristic did not work for this real-world data; in fact, it was worse than a random guesser. VoBERT showed performance on par with LogBERT on this real-world security event dataset.

To summarise, this work makes the following contributions:

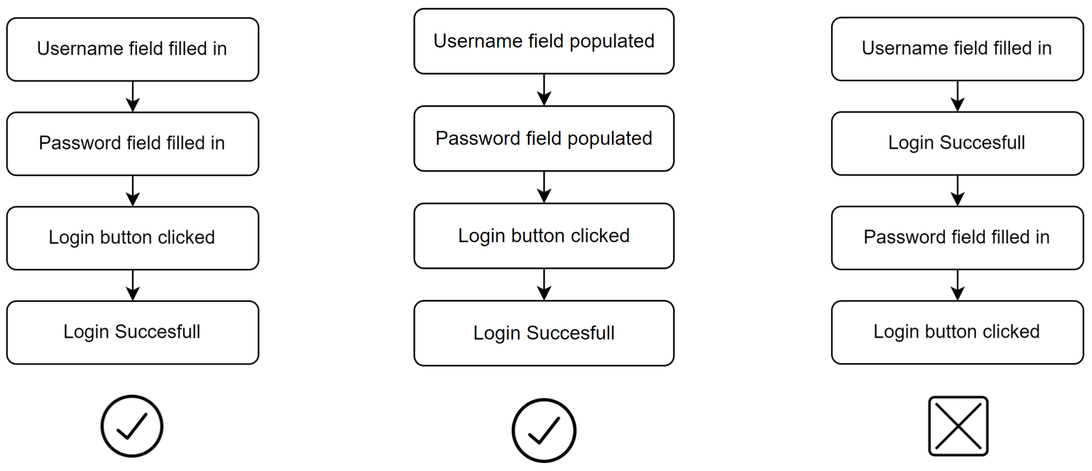
1. A reproduction and critical analysis of the results presented in LogBERT [26].
2. VoBERT, a BERT-based anomaly detection model using a novel pre-training task for sequence anomaly detection: Vocabulary-Free Masked Language Modeling (VF-MLM).
3. A case study at a large bank (50,000+ employees), evaluating whether the proposed technique can be applied in the context of security event sequence anomaly detection.

2 Problem Definition

2.1. Informal Definition

A simple example of log anomalies is depicted in Figure 2.1. In Figure 2.1a, a normal log flow of a user logging into some systems is shown. In Figure 2.1b, a suspicious log flow is shown. In this flow, the user is successfully logged in before entering a password. Finally, Figure 2.1c shows a normal log flow is seen that might appear after the application's logging statements are updated. The changed logkeys still have a similar meaning, but their wording is slightly changed. This final example is an example of unstable log data. However, it should be noted that this is not the only kind of unstable log data. Other possibilities include the removal of log statements or the addition of completely new log statements.

Most existing literature focuses on detecting anomalous sequences (Figure 2.1b) from normal sequences (Figure 2.1a), but disregards the issues presented by the unstable log data (Figure 2.1c).



(a) Normal log sequence

(b) Anomalous log sequence

(c) Normal log sequence with slightly changed log messages. This is an example of log data instability

Figure 2.1: Simplified example of normal and anomalous log sequences

Figure 2.2 presents the typical steps of log analysis. Throughout each step of the workflow, the effect this would have on the simple example with the login flow in Figure 2.1 is shown.

During the first step, the log messages are parsed. During this step, the log message is extracted from the log line by removing other elements such as a timestamp and debugging level. Furthermore, variable parts of the log message are replaced with a '*'. For the example in Figure 2.1 this would mean that "26-06-2020 12:00 INFO Username field filled in with Armaster_11" would be changed into "Username field filled in with *". The final product is called a logkey.

The second step is log message grouping. During this step, log messages need to be grouped into subsequences. This is often done with either fixed, sliding, or session windows.

The example in Figure 2.1 could be produced using fixed windows. This would mean that one big log file containing all three flows after each other in chronological order is split up in the three distinct log sequences shown in Figure 2.1.

Finally, the actual log sequence anomaly detection can be done. Since there are almost always none or very few labelled examples of anomalous log sequences, in most situations, a semi-supervised approach is used, where a model is trained with a train set consisting only of normal sequences. For the example in Figure 2.1, this would mean that a model is trained on a large dataset consisting of different normal sequences such as Figure 2.1a. When tasked with making a prediction about Figure 2.1b, it should ideally be able to classify this sequence correctly as anomalous. A model that can also handle unstable log data should also be able to classify Figure 2.1c correctly as normal.

An extension of this problem is unstable log sequence anomaly detection.

When log data is unstable, the content of the log messages changes slightly but still contains a similar meaning to the old version. An example of such a change can be seen in the first two log messages in Figure 2.1c. In the ideal scenario, a model trained on a set of normal log sequences before this change in log message was introduced should still be able to classify the sequence with slightly changed log messages without requiring retraining. The constraint of not requiring retraining is valuable in practice, as retraining is often not feasible [75].

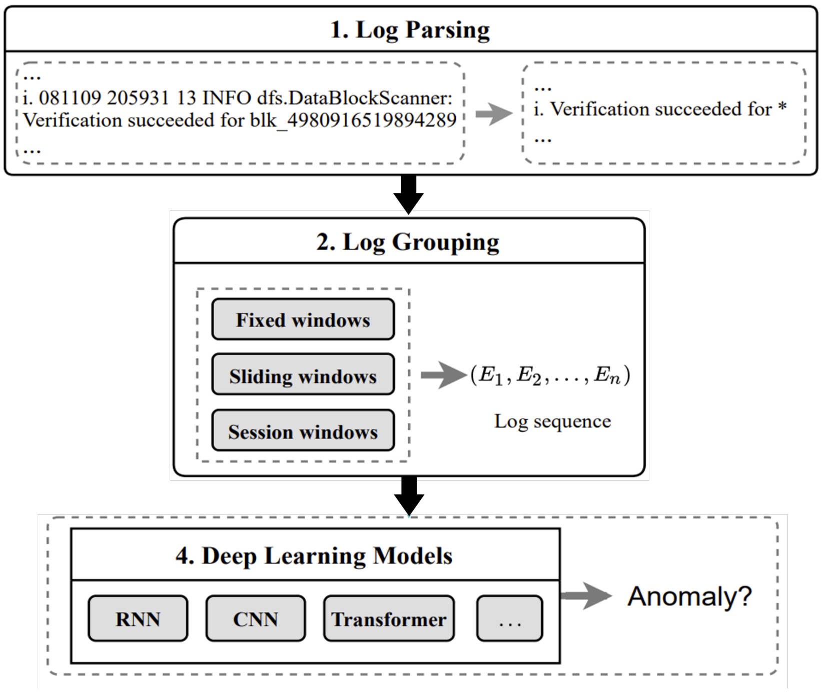

1. Log Parsing
...
i. 081109 205931 13 INFO dfs.DataBlockScanner: Verification succeeded for blk_4980916519894289
...
...
i. Verification succeeded for *
...
2. Log Grouping
Fixed windows
Sliding windows
Session windows
$(E_1, E_2, \ldots, E_n)$
Log sequence
4. Deep Learning Models
RNN
CNN
Transformer
...
Anomaly?

Figure 2.2: The common deep learning log anomaly detection workflow [39]. Steps 1 and 2 are considered pre-processing.

2.2. Formal Definition

2.2.1. Parsing

- Input: $L = (l_1, l_2, \dots, l_n)$ where L is the sequence consisting of all lines in a log file produced by an application, and l_i is the i-th line is this logfile.
- Output: $K = (k_1, k_2, \dots, l_n)$ where K is the sequence consisting of all parsed lines in a log file produced by an application, and k_i is the i-th line is this logfile. The parsed lines are called log keys.

The raw log messages are in free-text form and lack structure. They may contain the specific values of some variables such as file name, numerical variables, or identificators such as IP address or block number. These highly variable parts of the log message can reduce the performance of log anomaly detection [31]. This can be solved by parsing, which is the process of extracting the constant part of the log message by replacing the variable parts with a special symbol, often '*'. This constant part is called the logkey, a term that is frequently used in the rest of this work. When all raw log messages are parsed, the Vocabulary V can be defined as the set consisting of all unique log keys in the sequence of log keys K. In other words: $V = \{K\}$. This is the first step depicted in Figure 2.2.

2.2.2. Grouping

- Input: $K = (k_1, k_2, \dots, k_n)$ where K is the sequence consisting of all parsed lines in a log file produced by an application, and k_i is the i-th line is this logfile.
- Output: $S = (s_1, s_2, \dots, s_m)$ where S is the list of log key sequences obtained by splitting the complete sequence K into subsequences. s_i is the i-th sequence which consists of log keys: $s_j = (k_0, k_1, ..., k_n) \forall k_j | k_j \in V$.

After the parsing of the log messages into log keys, the log keys need to be grouped into sub-sequences that can be analysed. Log grouping is the task of creating subsequences out of the big sequence that is obtained after parsing the file. This is usually done with either a fixed, sliding or session window as shown in step 2 of Figure 2.2.

2.2.3. Sequence Anomaly Detection

- Input: $S = (s_1, s_1, \dots, s_m)$ where S is the list of log key sequences obtained by splitting the complete sequence K into subsequences. s_i is the i-th sequence which consists of log keys: $s_i = (k_0, k_1, \dots) \forall k_j | k_j \in V$.
- Output: $S_{pred} = (p_1, p_2, \dots, p_n)$ where S_{pred} is the sequence containing predictions that a given sequence is either normal or anomalous. p_i is the prediction that the i-th is anomalous. Each p_i is a boolean, where true denotes an anomalous sequence and false denotes a normal sequence.

After the pre-processing tasks (consisting of all previous tasks) are complete, the data is in the form of a set of sequences of ordered logkeys $S = \{s_1, s_2 ..., s_m\}$, where s_i indicates the logkey in the i-th position.

The pre-processing tasks can be applied to a train set $S_{\text{normal}} = \{s_1, s_2, ..., s_l\}$ containing only sequences marked as normal. The goal of log sequence anomaly detection is then to predict whether a new log sequence s_{test} is anomalous based on this training dataset S_{normal}. This problem is equal to the definition of *semi-supervised sequence-based anomaly detection* given in a survey on discrete sequence anomaly detection in general by [10]:

> **Semi-Supervised Sequence based anomaly detection**
> Given a set of n sequences, $S = \{s_1, s_2, ..., s_n\}$ and a sequence t belonging to a test data set T, assign an anomaly score to t with respect to the training sequences in S [10].

After applying a decision threshold to the anomaly score t of all logkeys in the test data set T, S_pred is obtained.

Note that there does not seem to be a consensus on what to call this type of learning. This work calls this semi-supervised learning, following two surveys on this subject: [10] and [37]. Some other works call this type of learning self-supervised. There are also works that call the same scenario unsupervised.

Often, these works assume that there are very few anomalies in the collected data, which is how they justify using the unlabeled dataset as their normal training set.

A 2022 survey [37] on log anomaly detection has identified three main challenges: Unstructured data, log instability, and the low availability of public datasets.

The first challenge is that log files predominantly appear in an unstructured or semi-structured format, varying across different devices, operating systems, or even software versions. The absence of a standardised structure and syntax for log files presents a significant hurdle. Centralising and processing such diverse and irregular data is an intricate challenge.

Second, [75] recognised the issue of log instability, attributing it to two main factors: The evolution of logging statements due to source code modifications and the introduction of noise during log data pre-processing.

Third, since log files are typically unstructured and their contents sensitive, they often can not be made publicly available due to security concerns. This lowers the availability of public datasets for research purposes, slowing progress in the log analysis field.

2.3. Research Question

In this thesis, I answer the following research question:

How can anomalies in unstable sequential log data be detected?

To answer this question, I have decomposed it into the following sub-questions:

1. **Element-Level Evaluation**. What influence does each individual logkey have on the sequence anomaly score?
2. **Unstable Logkeys**. How to adapt detection of anomalies in log sequences with a high ratio of unseen logkeys?
3. **Case Study**. Is it possible to apply the log anomaly detection technique to detect anomalies in real-world security event logs?

3 Related Work

3.1. Log Data Pre-processing

Log data is typically unstructured, which is why pre-processing is necessary before it can be used for log anomaly detection. These steps include log parsing or tokenising and log grouping. A survey [37] on deep learning for anomaly detection in log data has created an overview of the most common techniques to solve these pre-processing problems, which are described in this section. The authors of the survey also created a visual overview of the techniques, shown in Figure 3.1.

3.1.1. Parsing and Tokenising

Parsing and token-based strategies are the two main methods that have been identified for handling unstructured log data [37].

Parsing Parsing, a commonly employed pre-processing strategy, involves using log parsers to pinpoint the static part of each line, otherwise known as log key (KEY in Figure 3.1). Furthermore, the parsing process aids in extracting all the necessary parameters from log events, including possible identifiers and the timestamp. The latter is often used to apply time windows in the subsequent grouping stage of pre-processing. There exist many available log parsing solutions, such as Spell [20], IPLoM [47], Drain [30], and MoLFI [49]. In a study comparing log parsers, it was found that Drain achieves the best performance of the log parsers evaluated [77]. The parsers were evaluated on accuracy, robustness, and efficiency [77].

Accuracy measures how well a log parser can distinguish the constant parts from the variable parts of a log line. Most existing log parsing evaluation studies focus on accuracy, as the accuracy of a log parser has a large impact on the effectiveness of the downstream log analysis tasks [29].

Robustness measures how stable the log parser accuracy is tested on various datasets with different characteristics. A robust log parser is versatile, which makes it well-suited for deployment in various production settings.

Efficiency is a measure of how quickly it can process loglines. The faster a parser can parse a log dataset, the higher the efficiency.

Token-Based On the other hand, token-based strategies (TOK in Figure 3.1) provide an alternative approach. These methods break down log messages into word lists, typically using white spaces as separators. Subsequent data cleaning steps include lowercasing all letters, removing special characters and stop words, and ultimately generating the word vectors. Some techniques opt for a combined approach, generating token vectors from parsed events as opposed to raw log lines [8, 27].

3.1.2. Grouping

The section describes the process of organising log events into groups, a necessary step for detecting unusual patterns among multiple log events with deep-learning techniques. This process is visualised in the 'Event grouping' part of Figure 3.1.

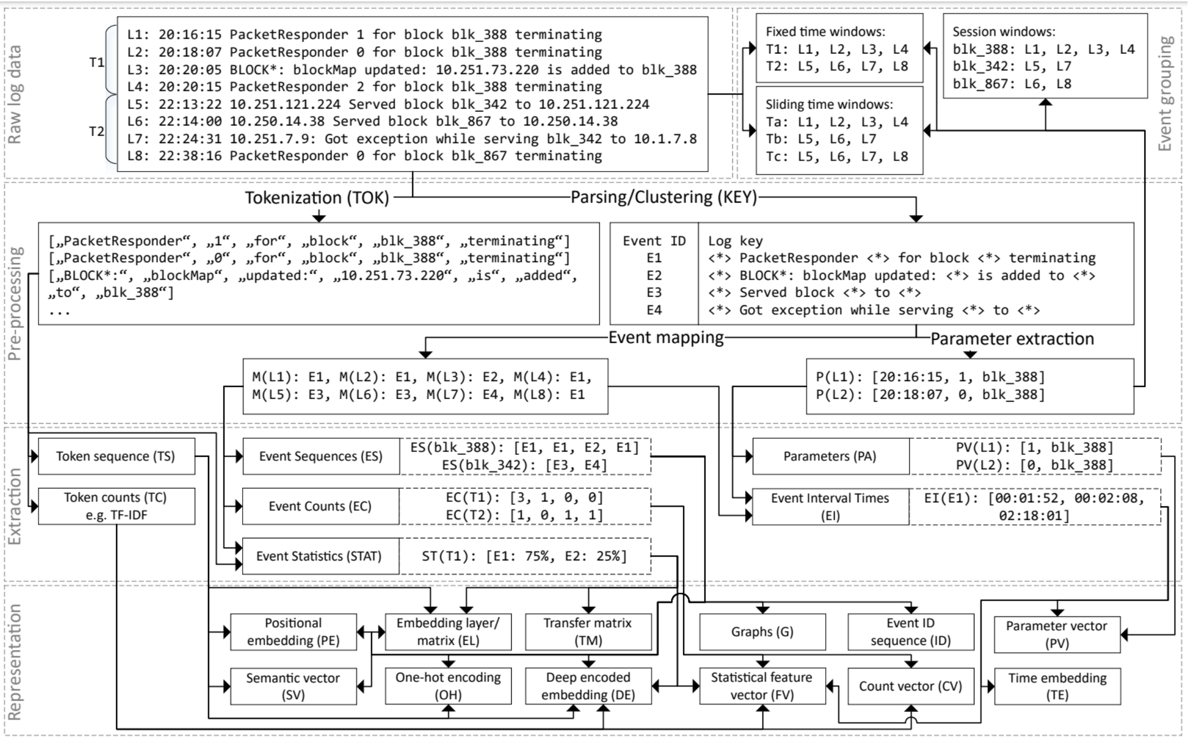
The timestamps attached to most log events make them ideal for groupings [38]. Since these timestamps are typically located at the start of log entries, they are simple to extract. We typically use two main methods to form groups based on time [32].

The first method involves creating sliding time windows. This involves moving a window of fixed length across the log data in uniform steps. At every step, logs falling within the window's current timeframe are grouped together. However, this method can lead to a single log entry appearing in several groups due to overlapping windows.

The second method employs fixed time windows, which is a variant of sliding time windows where the step size equals the window size. Although this approach gives a less detailed view of the data, it guarantees that each log event appears in only one time window, simplifying subsequent analyses like time-series evaluations.

Additionally, we can form groups based on the number of lines instead of the time when using either sliding or fixed windows. This method ensures all groups are the same size and eliminates the need to process timestamps. However, it makes it harder to treat event frequencies as time series because the windows now represent varying durations.

Lastly, session windows offer another way to group events. This method uses an event parameter that serves as an identifier for a specific task or process from which the event originated. This grouping method is useful for tracking event sequences that accurately reflect underlying workflows, even when multiple sessions are running in parallel on the system. However, not all log data types include such session identifiers.


```
L1: 20:16:15 PacketResponder 1 for block blk_388 terminating
L2: 20:18:07 PacketResponder 0 for block blk_388 terminating
L3: 20:20:05 BLOCK*: blockMap updated: 10.251.73.220 is added to blk_388
L4: 20:20:15 PacketResponder 2 for block blk_388 terminating
L5: 22:13:22 10.251.121.224 Served block blk_342 to 10.251.121.224
L6: 22:14:00 10.250.14.38 Served block blk_867 to 10.250.14.38
L7: 22:24:31 10.251.7.9: Got exception while serving blk_342 to 10.1.7.8
L8: 22:38:16 PacketResponder 0 for block blk_867 terminating
```

Fixed time windows:
```
T1: L1, L2, L3, L4
T2: L5, L6, L7, L8
```

Sliding time windows:
```
Ta: L1, L2, L3, L4
Tb: L5, L6, L7
Tc: L5, L6, L7, L8
```

Session windows:
```
blk_388: L1, L2, L3, L4
blk_342: L5, L7
blk_867: L6, L8
```

```
[„PacketResponder“, „1“, „for“, „block“, „blk_388“, „terminating“]
[„PacketResponder“, „0“, „for“, „block“, „blk_388“, „terminating“]
[„BLOCK*:“, „blockMap“, „updated:“, „10.251.73.220“, „is“, „added“,
„to“, „blk_388“]
...
```

Event ID	Log key
E1	<*> PacketResponder <*> for block <*> terminating
E2	<*> BLOCK*: blockMap updated: <*> is added to <*>
E3	<*> Served block <*> to <*>
E4	<*> Got exception while serving <*> to <*>

```
M(L1): E1, M(L2): E1, M(L3): E2, M(L4): E1,
M(L5): E3, M(L6): E3, M(L7): E4, M(L8): E1
```

```
P(L1): [20:16:15, 1, blk_388]
P(L2): [20:18:07, 0, blk_388]
```

Event Sequences (ES)	ES(blk_388): [E1, E1, E2, E1] ES(blk_342): [E3, E4]
Event Counts (EC)	EC(T1): [3, 1, 0, 0] EC(T2): [1, 0, 1, 1]
Event Statistics (STAT)	ST(T1): [E1: 75%, E2: 25%]
Parameters (PA)	PV(L1): [1, blk_388] PV(L2): [0, blk_388]
Event Interval Times (EI)	EI(E1): [00:01:52, 00:02:08, 02:18:01]

Figure 3.1: Overview of log data representation approaches used for Anomaly Detection [37]

3.2. Traditional ML Log Anomaly Detection

Over the years, numerous machine learning-based techniques for log anomaly detection have been developed. Some of them are supervised, many of which represent log sequences as log count vectors, and then use a machine learning technique for anomaly detection. Researchers applied the following traditional Machine Learning methods to log count vectors: Support Vector Machine (SVM) [43], Logistic Regression (LR) [12], and Decision Trees [6].

In addition to supervised learning, unsupervised learning methods for log anomaly detection have also been widely explored. Unsupervised methods are trained only on unlabeled data. For instance, the method proposed by [71] employed Principal Component Analysis (PCA) to construct two distinct subspaces from log count vectors: one representing normal events and the other representing anomalies. A log sequence is deemed anomalous if its log count vector deviates significantly from the normal space. Invariant Mining (IM) [45] and Anomaly Detection Rules (ADR) [74] are other examples of unsupervised approaches where linear relationships among log events are discovered from log count vectors. Log sequences violating these relationships are flagged as anomalous.

These traditional methods share a common framework: they all need a log parser for pre-processing and extracting logkeys from log messages [40]. Subsequently, count vectors are created by counting the frequency of all unique logkeys. The dimension of the log count vector corresponds to the total number of unique logkeys. An example of these count vectors can be seen in Figure 3.1.

While these approaches have significantly contributed to the field, they also share inherent limitations. One of the main issues stems from the instability of log data. When a new logkey is introduced caused by, for example, the evolution of log statements or pre-processing noise, these traditional methods need to change the dimension of the log count vector, which means the model needs to be retrained [75].

Additionally, using log count vectors can lead to losing important contextual information within the log sequences [75]. These methods only use the frequency of logkeys appearing in a sequence, not using the sequential nature of the data.

The final limitation lies in keeping these log anomaly detection methods up to date. To do so, a model needs continuous re-training, which could carry an unacceptable cost when applied to large-scale systems under active development [75].

More recently, deep-learning techniques have shown promise in overcoming some of these limitations. These advanced methods have demonstrated better performance in log anomaly detection. Therefore, the focus of this work is primarily on exploring and improving these deep-learning techniques rather than focusing on traditional machine-learning methods.

3.3. Deep Log Anomaly Detection

In recent years, many deep-learning-based models have been proposed for log sequence anomaly detection. Several recent surveys have been written on using deep learning for log sequence anomaly detection, such as [21, 9, 37, 73, 39]. An overview of recent papers that apply deep learning to log anomaly detection can be seen in Table 3.1. For older deep-learning log anomaly detection papers, an extensive overview is available in [37].

Paper	Year	Input Representation	Model	Training Task	Mode
LogEncoder [57]	2023	Semantic (by BERT)	LSTM	HS, CL	Semi
BERT-Log [13]	2022	Embedding Matrix	Pre-trained BERT	NN	Semi
LogBERT [26]	2021	Embedding Matrix	From scratch BERT	HS, MLM	Semi
OC4Seq [70]	2021	Embedding Matrix	2x GRU	2x HS	Semi
LogRobust [75]	2019	Semantic (FastText+TF-IDF)	Bi-LSTM	CE	S

Table 3.1: Overview of deep-learning techniques used in recent literature. For older literature, see [37]
HS: Hyper-Sphere, CL: Contrastive Learning, MCR: Masked Context Reconstruction, GRU: Gated Recurrent Unit, LSTM: Long short-term memory, MLM: Masked Language Modeling, CE: Cross Entropy, Semi: Semi-supervised, S: Supervised

3.3.1. Supervised Deep Log Anomaly Detection

Although most log anomaly detection methods are semi-supervised, some supervised methods also exist. These often achieve higher predictive performance, but can less easily be used in practice since, in reality, there is rarely sufficient labelled data available, especially labelled data of anomalous sequences. The rarity of such events primarily causes the lack of data on anomalous sequences.

The authors of [63] propose an improved version of the LogEvent2Vec [67] algorithm. The method transforms parsed logs into vectors, which are then used for supervised binary classification. In contrast to the original version, the authors decrease the analysis window. This made anomaly detection more accurate. However, their approach has three main drawbacks. First, it is supervised. Second, their approach is not evaluated on unstable log data. Third, since they combine all log vectors in the sequence to classify by taking the average, they lose the explainability to pinpoint which logs led to this classification after it is made.

The authors of [13] propose a supervised learning-based approach that classifies log messages as normal or anomalous. It uses a pre-trained BERT model to learn the vector representation of a log sequence. It then uses a neural network to classify this representation vector as normal or anomalous. The main drawback of this approach is that it is supervised.

LogRobust [75] introduces a new supervised method for log anomaly detection. This method works by pulling out meaningful information from log events and turning them into something called semantic vectors. To detect anomalies, LogRobust uses attention-based attention-based Bidirectional Long-Short-Term Memory Neural Network (Bi-LSTM). This is a type of LSTM model which can use both past and future input features [33]. LogRobust can understand the context in log sequences and automatically figure out which log events are more important using the attention mechanism. Thanks to these features, LogRobust is good at finding and dealing with unstable log events and sequences, making it an essential development in the field of log anomaly detection. However, the main drawback is that it needs labelled data of both normal and anomalous classes since it is a supervised approach. This makes it a lot less feasible to use in a real-world scenario.

3.3.2. Semi-Supervised Deep Log Anomaly Detection

Because of the lack of labelled data availability and the data class imbalance problem, most recent methods are semi-supervised or unsupervised. Usually, there is only one class available during training, the 'normal' class. A deep-learning model is trained to classify a data point as either part of this 'normal' class, or as anomalous. This type of anomaly detection is called Semi-Supervised anomaly detection, also known as Deep One-Class Classification.

OC4Seq [70] is a semi-supervised approach that employs Recurrent Neural Networks (RNNs) for detecting anomalies in discrete event sequences. In particular, OC4Seq combines anomaly detection with RNNs to transform discrete event sequences into latent spaces. Within these spaces, it is easier to spot anomalies [70]. The research data and code are publicly available online[1].

LogBERT [26] is a semi-supervised method for log anomaly detection that operates under the same presumption as LAnoBERT [26]. LogBERT, built on the BERT framework, aims to uncover log anomalies by learning and understanding the regular patterns found in normal log sequences. The main training task of LogBERT is MLM, being supported by the additional training task Volume of Hypersphere Minimisation. Masked Language Modeling was originally created as an NLP task, where the goal is to train a model to predict the original tokens on some masked positions in a natural language sentence. LogBERT uses MLM to detect anomalies in log data by training a model to do MLM on normal sequences. The assumption is that if that model is then queried on anomalous sequences, it will not be able to predict the masked logkeys in these sequences as accurately as it could for normal sequences. Thus, using a distance metric between the real and predicted output of a trained model anomalous and normal sequences can be distinguished by LogBERT. VHM aims to train the model in such a way that the hypersphere that contains all vector representations of normal logkeys has a minimal volume. The assumption is that anomalous logkeys will then lie outside of this sphere. Log sequences that deviate from these identified patterns are then classified as anomalous [26]. The implementation of LogBERT is publicly available[2]

During the writing of this thesis, LogEncoder was published. In [57], a semi-supervised method that can deal with unstable log data is proposed. While this work also addresses the issue of unstable log data, they do this with a different method than this thesis. Their framework consists of three modules: Log to embedding (Log2Emb), Embedding to representation (Emb2Rep), and Anomaly detection. They utilise a pre-trained model to obtain semantic vectors from each log event and use one-class and contrastive learning objectives to train a representation model. Their one-class learning objective is inspired by the VHM objective, as introduced in DeepSVDD [62]. They then use a third model to classify the representations generated by the second model as anomalous or normal. When tested on three benchmark datasets against six state-of-the-art methods, LogEncoder outperformed five of them and performed comparably to a supervised method, LogRobust. Besides using a different, more complicated structure, LogEncoder does not use Masked Language Modeling, which is the focus of this thesis.

[1]https://github.com/wzwtrevor/Multi-Scale-One-Class-Recurrent-Neural-Networks
[2]https://github.com/HelenGuohx/logbert

4 Preliminaries

4.1. Transformers and BERT

In NLP tasks, the input is typically a sequence of word or subword tokens, and the goal is to predict some property of the sequence, such as its sentiment, the named entities it contains, or the next word in the sequence. The traditional approach to processing sequences uses recurrent neural networks (RNNs), which pass information from one token to the next along a sequence. However, RNNs can be slow to train and struggle to handle sequences of long length since each step needs information from the previous.

Transformers solve these problems by using self-attention to dynamically weigh the importance of different tokens in the input sequence. In a transformer, each token is first transformed into a vector representation, which is then used to calculate the attention scores between all pairs of tokens. These attention scores are used to weigh the contribution of each token to the representation of the whole sequence, which is then passed through a feedforward neural network to make the final prediction.

The architecture of transformers has proven to be very effective in a wide range of NLP tasks, including language translation, text classification, and question answering [66]. It has quickly become the default choice for many NLP problems, and many state-of-the-art models in NLP are based on transformers, of which perhaps the best well-known model is Bidirectional Encoder Representations from Transformers (BERT) [19].

Bidirectional Encoder Representations from Transformers (BERT) is a language model introduced by Devlin et al. in 2018 [19]. BERT revolutionised the field of Natural Language Processing (NLP) by employing a bidirectional training approach. Unlike traditional language models that read the text sequentially (either from left to right or right to left), BERT is designed to analyse the context of words in both directions. This is made possible by its underlying transformer architecture and its attention mechanism, enabling it to weigh the influence of different words when encoding the information of a given word.

BERT's training strategy consists of two steps: pre-training and fine-tuning. The pre-training phase involves learning word representations from a large text corpus, where BERT is trained on tasks such as Masked Language Modelling (MLM) and Next Sentence Prediction (NSP). This allows BERT to learn a universal language model that understands syntax and semantics. In the fine-tuning phase, the pre-trained BERT model is adapted to a specific task (such as question answering or sentiment analysis) with a smaller amount of task-specific data. Despite the requirement of significant computational resources for training and challenges in adapting to low-resource languages, BERT's capacity to comprehend the nuances of language has led to state-of-the-art results in numerous NLP tasks, contributing significantly to the ongoing evolution of language understanding models.

In the original BERT model paper [19], text input is first tokenised using WordPiece tokenisation [65]. Each token is then converted into an embedding vector. These initial token embeddings are then updated during the training of BERT.

4.2. LogBERT

The proposed Vocabulary-Free Masked Language Modelling method is based on the traditional MLM method used in LogBERT [26]. The goal of LogBERT is to classify sequences of logkeys produced by the pre-processing steps as either normal or anomalous. The output of this task is a label y_i (normal or anomalous) for every log sequence s_i as shown in Equation (4.1).

$$output = \{(s_1, y_1), ..., (s_n, y_n)\} \tag{4.1}$$

Where y_i is the classification result made by the model for log sequence s_i.

LogBERT [26] is a self-supervised framework for log anomaly detection based on BERT. It is self-supervised because it is trained only on labelled normal data, no anomalous data is required. LogBERT learns the patterns of normal log sequences by using Masked Language Modelling (MLM) supported by a secondary training task: Volume of Hypersphere Minimisation (VHM). LogBERTs' original implementation is publicly available[1].

The main principle behind LogBERT is to use Masked Language Modeling as an anomaly detection method. By training on normal sequences only, the model attains a certain proficiency in performing Masked Language Modelling on normal sequences. After training, the accuracy with which LogBERT can do Masked Language Modelling on average is higher for normal sequences than for anomalous sequences, as it has never seen the latter during training. Using this, a predictive performance threshold can be found below which sequences are marked as anomalous.

The core assumption made for LogBERT to work is that anomalous sequences differ from normal sequences to such an extent that MLM is more difficult on anomalous sequences than on normal sequences for a model trained only on normal sequences.

4.2.1. Pre-Processing

Before the LogBERT model can be used, pre-processing of the logs is required. In this section, the pre-processing steps used in the original LogBERT paper are described.

As the first pre-processing step, LogBERT uses the log parser Drain [30]. Then, for the TBird and BGL datasets (Section 6.1.1), LogBERT uses a straightforward sliding window grouping method. It uses a sliding window of size w_{size} with slides with increments of w_{step} time across the parsed log file. If a sequence is shorter than minimal sequence length l_{min}, the sequence is discarded. If a sequence is longer than the maximum sequence length l_{max}, the sequence is truncated to a length of l_{max}.

4.2.2. Embedding Layer

The logkeys need a numerical representation to be fed to the BERT model. LogBERT achieves this by using a strategy we call matrix embedding. In matrix embedding, the logkeys all correspond to a randomly generated vector, which is called the embedding of the log key. The vectors are stored in a randomly generated matrix $E \in R^{|K|*sd}$, where d is the dimension of log key embedding vectors. The authors call this the logkey embedding matrix. Positional embeddings are generated by using a sinusoid function to encode positional information similar to their use in the original BERT implementation [19]. Thus, the input representation of the logkey k_t is defined as $x_t^j = e_{k_t^j}$. This method is used, apart from LogBERT, in [13, 70].

[1] `https://github.com/HelenGuohx/LogBERT`

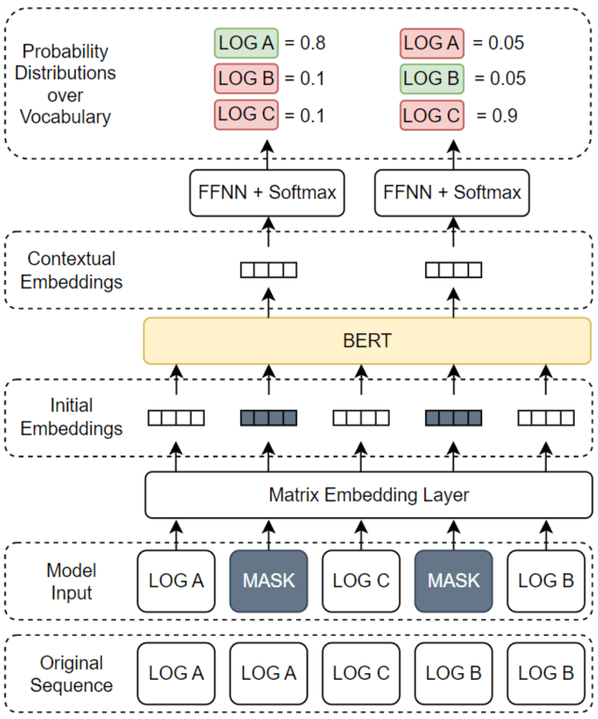


Figure 4.1: Masked Language Modelling. A trained BERT model predicts the masked tokens. The model correctly predicted Log A for the first masked token and incorrectly predicted Log C for the second masked token. If $g = 1$ and $r = 1$, then the sequence would be marked as anomalous.

4.2.3. Training

The goal of training a sequence model for anomaly detection is to be able to differentiate between normal and anomalous sequences, by leveraging the temporal context information of the elements in the sequence, in addition to looking at the elements separately. LogBERT introduces two distinct training tasks to enhance anomaly detection in log sequences [26].

Firstly, the Masked Language Modelling task concentrates on predicting the log keys within normal log sequences that have been randomly masked. Note that the authors of LogBERT call this task Masked Log Key Prediction (MLKP), but since it is no different from the original MLM training task originally introduced in [19] the term MLM is used in this work.

Secondly, the Volume of Hypersphere Minimisation task is designed to make normal log sequences closer together in the embedding space. These combined efforts contribute to more accurate and efficient detection of anomalies [26]. Other training tasks used by some other log anomaly detection works include Contrastive Learning (used in [68, 57]) and Masked Context Reconstruction (used in [68]).

Training Task 1: Masked Language Modelling The first training task is Masked Language Modelling, introduced by the original BERT paper [19]. The goal of training the model is to capture the bidirectional context information present in log sequences. This is accomplished by randomly replacing a certain ratio of logkeys with a special [MASK] token with a certain probability and then letting the model predict the masked logs. This method is called ratio masking. LogBERT uses a masking ratio of 15%.

If the log that was originally on the masked position is not among the top g candidate keys predicted by the model, it is considered anomalous. Figure 4.1 displays a sequence of length 4 inserted into a BERT model with a masking ratio of 50%. For the model's output, only the logkey with the highest predicted probability is shown, which depicts the situation when $g = 1$. In LogBERT [26], a log sequence is labelled as anomalous if it contains $\geq r$ anomalous logs keys. g is a hyper-parameter, and r is used as the decision threshold. In Figure 4.1, if $r = 1$, the sequence would be predicted as anomalous (since $1 \geq 1$).

The intermediate output of the BERT model is contextual embedding vectors of the [MASK] tokens. The next step is to feed these vectors into a feed-forward neural network that has an output node for each logkey in the vocabulary V of the model. The output is then fed into a softmax function, which outputs a probability distribution over all possible logkeys in the model's vocabulary V. The computation of the model's output for a particular masked token in a particular sequence is shown in Equation (4.2). This output is a probability distribution over the entire vocabulary V:

$$\hat{\mathbf{y}}^j_{[\mathrm{MASK}_i]} = \mathrm{Softmax}\left(\mathbf{W}_C \mathbf{h}^j_{[\mathrm{MASK}_i]} + \mathbf{b}_C\right) \tag{4.2}$$

where $\hat{\mathbf{y}}^j_{[\mathrm{MASK}_i]}$ is the output for the i-th masked token in the j-th sequence, $\mathbf{h}^j_{[\mathrm{MASK}_i]}$ is the output of the BERT model, a contextual embedding, of the i-th masked token in the j-th sequence. $\mathbf{W}_C$ and $\mathbf{b}_C$ are trainable parameters in the single-layer feedforward neural network.

After obtaining the model's output, the MLM loss must be computed. The MLM loss is defined as the Cross-Entropy Loss between the output probability distribution and the perfect probability distribution. The perfect probability distribution for a certain masked position is defined as the probability distribution where the original token has a probability of 1 and all other tokens in the vocabulary have a probability of 0. After calculating the loss for all masked tokens, the loss is averaged to obtain a sequence-level anomaly score. The MLM loss calculation can be seen in Equation (4.3).

$$\mathcal{L}_{MLM} = -\frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{M_j} \mathbf{y}^j_{[\mathrm{MASK}_i]} \log(\hat{\mathbf{y}}^j_{[\mathrm{MASK}_i]}) \tag{4.3}$$

where $y^j_{[MASK_i]}$ is the label for the i-th masked token in the j-th sequence, M_j is the total number of masked tokens in the j-th log sequence, and $\hat{\mathbf{y}}^j_{[\mathrm{MASK}_i]}$ is the softmax probability for the i-th masked token in the j-th sequence.

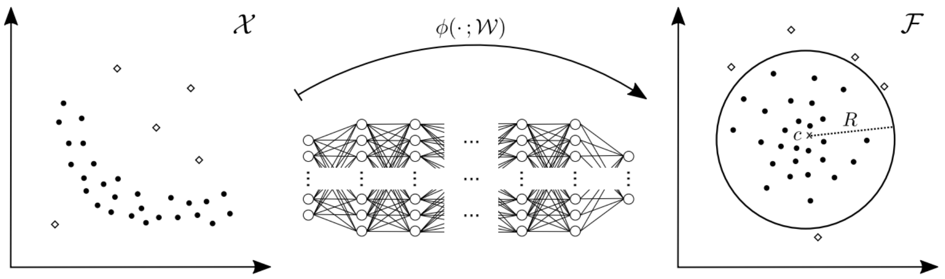


Figure 4.2: Deep Support Vector Data Description (Deep SVDD) learns a neural network transformation $\phi(\cdot; W)$ with weights W from input space $X \subseteq \mathbb{R}^d$ to output space $F \subseteq \mathbb{R}^p$ that aims to map the data network representations into a hypersphere with centre c and radius R in such a way that the volume is minimised. Normal samples fall within, and anomalous samples fall outside the hypersphere [62].

Training Task 2: Volume of Hypersphere Minimisation This task is inspired by a paper on Deep One Class Classification [62], where the objective is to minimise the volume of a data-enclosing hypersphere. This is shown in Figure 4.2. BERT models have a special [CLS] (Classification) token, and the corresponding output node for this token contains the sequence embedding for a sequence fed to BERT. The BERT model is trained to minimise the distance between the sequence embedding of all sequences during training. "The motivation is that normal log sequences should be concentrated and close to each other in the embedding space, while the anomalous log sequences are far to the centre of the sphere" [26]. The loss function of this training task is $\mathcal{L}_{VHM}$. The exact formula can be found in [19].

$$\mathcal{L}_{VHM} = \frac{1}{N}\sum_{j=1}^{N}\left\|\mathbf{h}^{j}_{\text{DIST}} - \mathbf{c}\right\|^2 \tag{4.4}$$

The total loss function is defined as:

$$\mathcal{L} = \mathcal{L}_{MLM} + \alpha\mathcal{L}_{VHM} \tag{4.5}$$

where α is a hyper-parameter that determines the balance between the two training tasks.

4.2.4. Classification

Once trained, LogBERT can be utilised for detecting anomalies in log sequences. The core assumption is that if a testing log sequence is normal, the model can accurately predict the masked logs. Therefore, the anomalous score of a log sequence can be determined based on the model's predictions for the [MASK] tokens. If the log that was originally on the [MASK] spot is not among the top g candidate keys predicted by the model, that logkey is considered anomalous. A log sequence is labelled as anomalous if it contains $\geq r$ anomalous logs. In LogBERT [26] g is a hyperparameter, and r is used as a decision threshold. LogBERT optimises this decision threshold on the test set, which does not reflect performance in real situations. Therefore, this thesis will optimise the decision threshold on a development set instead.

5 Method

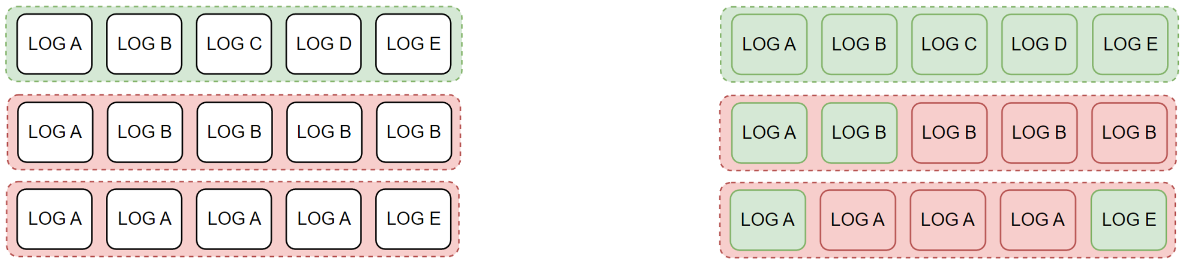
This chapter describes how the research questions were answered. For RQ1 about Element-Level Evaluation, a per-element masking method is introduced. To answer RQ2 about how to be more robust with respect to Unstable Logkeys, a novel technique is proposed: Vocabulary-Free Masked Language Modeling.

5.1. RQ1: Element-Level Evaluation

In this section, we answer the following question:

RQ1: What influence does each individual logkey have on the sequence anomaly score?

To evaluate the influence of the logkeys in the sequence on the sequence anomaly score, we computed anomaly scores per element. To achieve this, per-element masking instead of ratio masking was implemented, as well as aggregation of per-element scores into sequence scores. The difference between sequence and element-level evaluation is displayed in Figure 5.1. It was found that while Element-Level evaluation can provide some additional accuracy and explainability, it comes at a great computational cost.



(a) Sequence-level Evaluation. Predictions are made only on sequence-level.

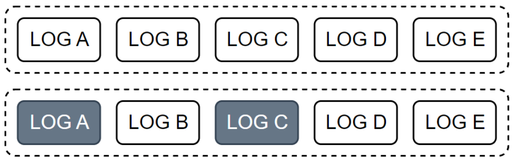
(b) Element-level Evaluation. Predictions are also made on element-level.

Figure 5.1: Sequence vs element-level evaluation visualised for the same three sequences.

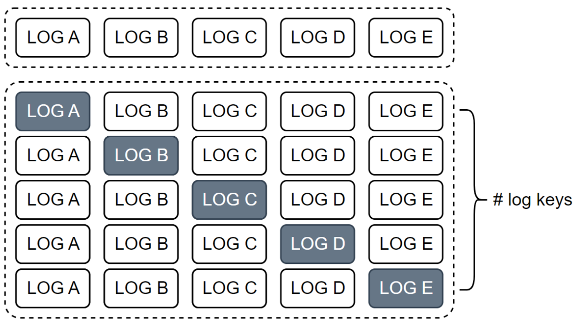
We define the element-level score for each element e_i in the sequence s of length n as the anomaly score for a sequence $s = (e_1, e_2, e_i, e_n)$ where only element e_i is masked. Masking each element in a sequence exactly once is not supported in the implementation of LogBERT, so we need to modify the implementation to support element-level evaluation. To this end, multiple changes were introduced, described in the rest of this section.

5.1.1. Masking method

The masking of tokens is traditionally only reported on sequence-level. However, to assess the influence of all elements, per-element masking is needed. The LogBERT implementation uses ratio masking, but for computing an element-wise anomaly score, per-element masking is required. The difference between ratio and per-element masking is visualised in Figure 5.2.



(a) Ratio Masking



(b) Per Element Masking

Figure 5.2: Ratio Masking versus Per Element Masking.

For per-element masking, instead of masking a predetermined ratio of tokens, all tokens are masked one by one (see Figure 5.2b). There are two main benefits: First, the sequence-level anomaly score might be more accurate since all tokens are masked once and therefore considered in computing the sequence-level anomaly score, which eliminates the stochastic element of randomly choosing certain tokens to mask. Second, this allows for the computation of per-element anomaly scores, which enable an element-level evaluation. The drawback of this approach is that instead of the one prediction for a sequence on length n that ratio-masking requires, computing the sequence-level anomaly score now requires making n predictions. However, making a prediction has a relatively low computational cost, and the maximum sequence length $n_{\max}$ is known beforehand. The n predictions are aggregated into a sequence-level prediction while retaining the element-level score.

5.1.2. Element Score Aggregation

After making all m predictions for each sequence to compute the element-level anomaly scores, the sequence anomaly scores still need to be computed. The aggregation of element-level anomaly scores into sequence-level scores differs between traditional MLM and VF-MLM pre-training tasks because they use different loss functions. For VF-MLM, the aggregation is done by averaging all element-level anomaly scores, as shown in Equation (5.1).

$$\text{anomaly\_score}(s_i) = \frac{1}{m}\sum_{j=1}^{m} \text{loss}(s_i^j) \tag{5.1}$$

where s_i is the i-th sequence in the test set, s_i^j is the generated variation of the i-th sequence where only the j-th element is masked, and m is the number of masked tokens in the sequence. For per-element masking, $m = n$ where n is the sequence length.

For traditional MLM, the aggregation is done by summing the number of masked tokens and amount of undetected tokens of all element-level anomaly scores. The sequence-level anomaly score is then defined as the ratio of undetected tokens, calculated by dividing the number of undetected tokens by the number of masked tokens. For per-element masking, the number of masked tokens always equals the sequence length. This calculation is shown in Equation (5.2).

$$\text{anomaly\_score}(s_i) = \frac{1}{m}\sum_{j=1}^{m} \text{num\_undetected}(s_i^j) \tag{5.2}$$

where s_i is the i-th sequence in the test set, s_i^j is the generated variation of the i-th sequence where only the j-th element is masked, and m is the number of masked tokens in the sequence. For per-element masking, $m = n$ where n is the sequence length. The algorithm used to generate all sequence and masked position pairs for per-element masking is shown in Algorithm 1.

Algorithm 1 Generation of per element masking `dataloader` key index

```
$s$                                        ▷ List of all sequences

$k \leftarrow \{\}$                        ▷ Stores sequence and mask index pairs for a given key index
$c \leftarrow 0$                           ▷ Initialise the index count to 0
for $i \in \{0, \ldots, |s| - 1\}$ do
    $n \leftarrow |s_i|$                   ▷ Current sequence length
    $x \leftarrow 0$                       ▷ Initialise mask index count
    for $j \in \{c, \ldots, c + n - 1\}$ do
        $k_j \leftarrow \{i, x\}$          ▷ Assign key index values
        $x \leftarrow x + 1$               ▷ Increment mask index count
        $c \leftarrow c + 1$               ▷ Increment count
    end for
end for
```

5.2. RQ2: Unstable logkeys

In this section, we answer the following question:

> *RQ2: How to adapt detection of anomalies in log sequences with a high ratio of unseen logkeys?*

To answer this question, we propose a novel BERT-based sequential anomaly detection model: Vocabulary-Free BERT (VoBERT). VoBERT uses a novel pre-training task we designed specifically for anomaly detection: Vocabulary-Free Masked Language Modeling (VF-MLM). We adapt the implementation of traditional MLM published by the authors of LogBERT [26] and remove the fixed-vocabulary constraint. This is possible since VF-MLM is designed for sequence anomaly detection rather than NLP, so we do not need the model output to be mapped back to natural language words. It is enough to know how well the trained model can reconstruct the original sequence.

To enable vocabulary-free masked language modelling, two main challenges need to be solved:

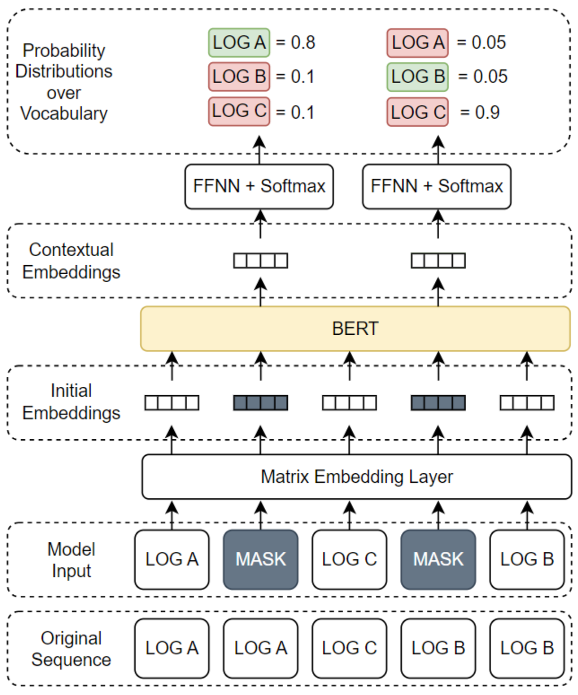
1. **Model Architecture**. The model architecture needs to be changed in such a way that it is not dependent on the vocabulary.
2. **Token Embedding**. The element embedding method needs to be able to generate an initial embedding for any possible element, regardless of if it has seen this element before.

Both of these requirements are violated in traditional MLM, so a novel method needs to be found for these requirements.

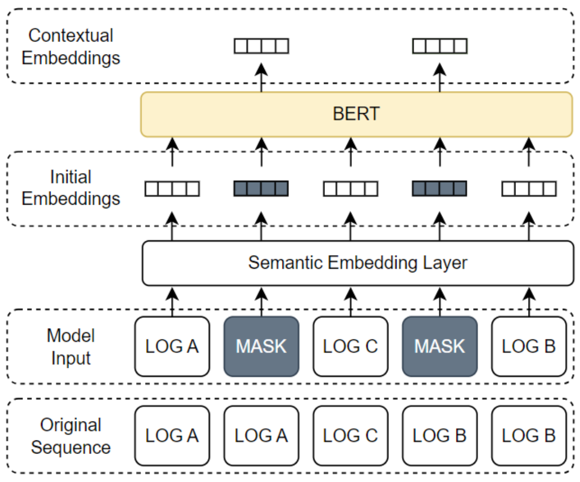
5.2.1. Vocabulary-Free Model Architecture

The VF-MLM model architecture is an extension of the MLM model architecture implemented in LogBERT [26] (see Section 4.2). The implementation of VF-MLM created for this work is based on the publicly available code of LogBERT and implemented using PyTorch [54]. The difference between the traditional and vocabulary-free MLM model architecture is visualised in Figure 5.3.

Traditional MLM Architecture The traditional MLM model architecture is shown in Figure 5.3a. When making an MLM prediction, some tokens are masked meaning they are replaced with a special [MASK] value. As a next step, the initial embedding module generates an initial embedding for every token in the sequence using the matrix embedding method explained in Section 4.2.2. The BERT model then computes contextual embeddings for each token in the sequence. These contextual embeddings are then passed to a single-layer fully connected feed-forward neural network (FFNN). For these output positions, this layer has an output node for every possible token in the model's vocabulary. As a final step, the softmax function is applied to all output nodes of this FFNN layer, resulting in a probability distribution over all tokens in the vocabulary. The outputs at the positions where the [MASK] tokens were placed are selected. The probability distributions of these positions specify the probability that the masked value is a certain token out of the vocabulary.



(a) Traditional MLM Model Architecture.



(b) Vocabulary-Free Model Architecture. Note that the embedding method changed, and the FFNN + Softmax layer is removed.

Figure 5.3: Vocabulary-Free MLM Model Architecture to traditional MLM model architecture. Adapted from [4].

Since the number of output nodes must be equal to the number of tokens in the vocabulary, the model architecture is dependent on the vocabulary, violating the first requirement of VF-MLM: “The model architecture needs to be changed in such a way that is not dependent on the vocabulary”.

Vocabulary-Free MLM architecture In Figure 5.3b, the proposed VF-MLM architecture is shown. As visible in the picture, the FFNN layer and softmax function are removed. This means the model now outputs the raw contextual vector for every token instead of a probability distribution over all tokens in the vocabulary. This removes the dependency of the model architecture on the vocabulary.

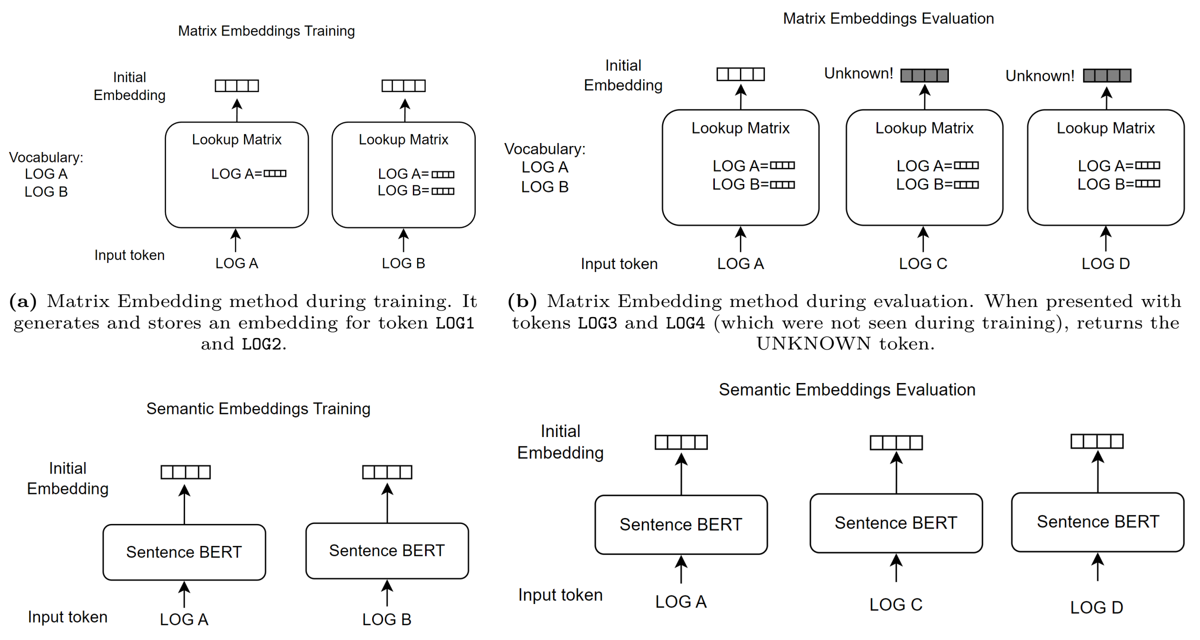
Additionally, the initial embedding is changed from matrix embeddings to semantic embeddings. This is explained in more detail in Section 5.2.2.

5.2.2. Vocabulary-Free Embedding Layer

Traditional MLM uses an embedding matrix to get the initial token embeddings. This method is described in more detail in Section 4.2.2. This method violates the second constraint of VF-MLM: “The element embedding method needs to be able to generate an initial embedding for any possible element, regardless of if it has seen this element before”. A matrix embedding module can only generate useful embeddings for tokens it has seen during training. If presented with a token it has not seen during training, it generates a special unknown vector, often denoted as [UNK]. This unknown token is the same as any token it has not seen during training, which means unseen tokens can not be distinguished from one another after vectorisation.

To not violate the second constraint, a method is needed to create embeddings from the logkeys in such a way that they can be generated for any logkey regardless of whether it was included in the train set. This can be done by generating an embedding that captures the semantic meaning of the logkey. Apart from that the produced vectors need to be of a fixed size, [75] identified two more requirements semantic vectors should have: Compatibility and Discrimination.

Compatibility means that semantically similar logkeys should have similar vectors. For example, “Username field filled in” and “Username field populated” are two logkeys with similar meanings, so their embedding should also be similar.



(a) Matrix Embedding method during training. It generates and stores an embedding for token `LOG1` and `LOG2`.

(b) Matrix Embedding method during evaluation. When presented with tokens `LOG3` and `LOG4` (which were not seen during training), returns the UNKNOWN token.

(c) Semantic Embedding method during training. It generates an embedding for token `LOG1` and `LOG2`, but does not store anything.

(d) Semantic Embedding method during evaluation. When presented with tokens `LOG3` and `LOG4` (which were not seen during training), still returns a semantic embedding.

Figure 5.4: Comparison of the behaviour of matrix and semantic embedding methods.

Discrimination means that semantically different logkeys should have different vectors. For example, "Login Succesful" and "Username filled in" are two logkeys with different meaning, so their embedding should also be different.

Many prior studies [40], [75], [39], [14] have focused on log anomaly detection by utilising the semantic information embedded in the log messages, and they show that this can have a significant impact on performance. To extract semantics from text data, there are many NLP models available, such as Glove [55], Word2Vec [52], and FastText [34]. More sophisticated approaches, such as the BERT transformer model [19], also exist. The BERT transformer model has an advantage over models like Word2Vec because, with Word2Vec, each word has a fixed representation that does not consider the context of the word. On the other hand, BERT produces embeddings that consider the context, i.e., the words around them. A specific type of pre-trained BERT model is SentenceBERT [59], which is pre-trained to embed English natural language sentences. The implementation of VF-MLM created in this thesis uses SentenceBERT as the semantic embedding layer.

The difference between matrix and semantic embeddings is visualised in Figure 5.4. In Figure 5.4a, the matrix embedding method is trained with the vocabulary of $\{LOGA, LOGB\}$. It generates random embeddings for these tokens and updates the embeddings during each backward pass of the model's training. After training, it stores these embeddings for use during prediction. When faced with tokens `LOG C` and `LOG D`, which were not present in the training set, it produces an UNKNOWN token as shown in Figure 5.4b.

In Figure 5.4c, the semantic embedding method is trained on the same training set, which has a vocabulary of $\{$`LOG A`, `LOG B`$\}$. It generates an embedding based on the token, which in this study is a vector that captures the natural language meaning of the logkey. When presented with tokens `LOG C` and `LOG D` during prediction as depicted in Figure 5.4d, it can still generate meaningful embeddings for these tokens.

5.2.3. Vocabulary-Free Training

The main difference in the training process of VF-MLM compared to MLM is the loss function. In general, the loss function of MLM is defined as the similarity between the predicted token and the label. This similarity can be measured in different ways. As explained in Section 4.2, the loss function of traditional MLM is the cross-entropy loss between the output of the model for a masked position, which is a probability distribution over all tokens in the vocabulary, and a perfect probability distribution in which the token embedding that was originally on that position has a value of 1 and all other tokens have a value of 0.

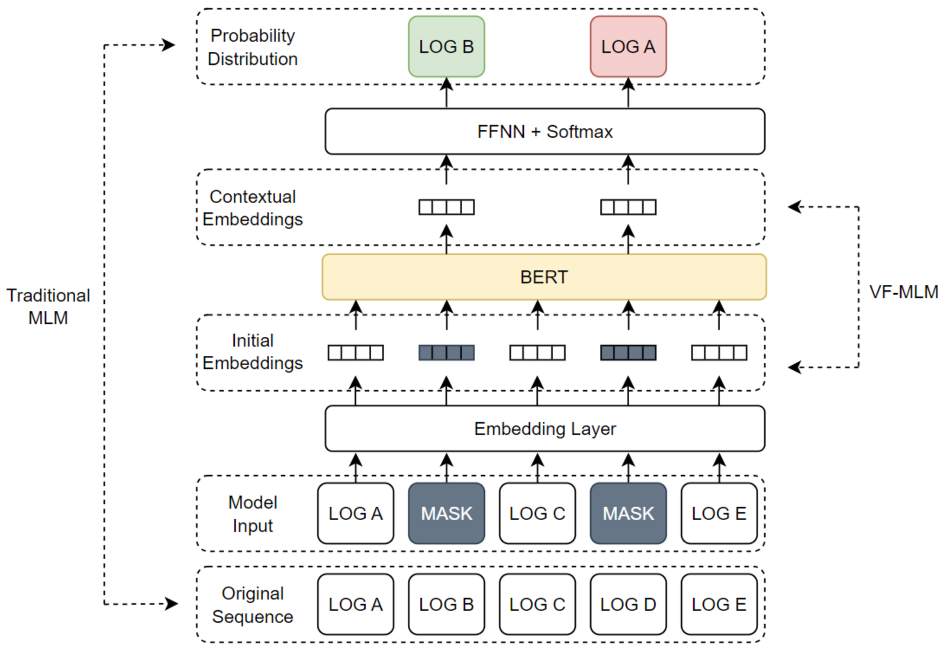


Figure 5.5: Vocabulary-Free MLM compares initial and contextual embeddings of masked logkeys directly, whereas traditional MLM compares the original logkey and the probability distribution over all logkeys in the vocabulary.

However, this method can not be used for VF-MLM since it does not output a probability distribution over all tokens in the vocabulary. Doing so would make the model architecture dependent on the vocabulary. Instead, the predicted contextual embeddings created by BERT and the initial embeddings created by the embedding layer are compared directly. The difference in measuring similarity between the masked and predicted token is visualised in Figure 5.5. Since the probability distribution is not required, the final layer can be removed. The altered model output equation for VF-MLM is shown in Equation (5.3). Note that the model's output is simply the contextual embedding produced by BERT.

$$\hat{\mathbf{y}}^j_{[\text{MASK}_i]} = \mathbf{h}^j_{[\text{MASK}_i]} \tag{5.3}$$

where $\hat{\mathbf{y}}^j_{[\text{MASK}_i]}$ is the output for the i-th masked token in the j-th sequence, $\mathbf{h}^j_{[\text{MASK}_i]}$ is the output of the BERT model, a contextual embedding, of the i-th masked token in the j-th sequence. Note that the softmax layer as well as the single feedforward layer are removed.

To compare the initial embedding to the predicted embedding, different loss criteria can be used, such as L1 loss (Mean Absolute Error), L2 loss (Mean Squared Error), or cosine similarity loss. The new loss function for VF-MLM using L1 loss can be seen in Equation (5.4).

$$\mathcal{L}_{VF-MLM} = -\frac{1}{N}\sum_{j=1}^{N}\frac{1}{M_j}\sum_{i=1}^{M_j}\left|\mathbf{y}^j_{[\text{MASK}_i]} - \hat{\mathbf{y}}^j_{[\text{MASK}_i]}\right| \tag{5.4}$$

where $y^j_{MASK_i}$ indicates the real logkey for the i-th masked token in the j-th sequence, and M_j is the total number of masked tokens in the j-th log sequence.

The total loss function is then defined as Equation (5.5):

$$\mathcal{L} = \mathcal{L}_{VF-MLM} + \alpha\mathcal{L}_{VHM} \tag{5.5}$$

where α is a hyper-parameter that determines the balance between the two training tasks.

Similarly to the cross-entropy Loss in LogBERT, the loss only needs to take into account the model output for masked tokens. In LogBERT, a standard PyTorch [54] functionality can be used, specifying an ignore index for the cross-entropy loss. However, that functionality is not present for L1, L2, and cosine similarity loss, because it is not normally used in such a way. Therefore, this was implemented manually as shown in Algorithm 2.

Algorithm 2 The MLM loss computation, ignoring padding tokens. In the comments, the tensor shape is denoted between parentheses. a: batch size, b: sequence length, c: token dimension

```
label                                              ▷ The labels for this batch. (a,b,c)
output                                   ▷ The generated output for this batch. (a,b,c)

MLM_loss ← loss_criterion(output, label])      ▷ Loss measured by loss criterion (a,b,c)

non_pad_mask ← label.sum(dim=2) ≠ 0   ▷ Boolean Mask to select non-padding tokens (a,b)
$l_{non\_pad}$ ← non_pad_mask.sum(dim=0)                   ▷ Count non-padding tokens

MLM_loss ← MLM_loss[non_pad_mask]          ▷ Non-padding tokens. ($l_{non\_pad}$, 768)
if $l_{non\_pad} > 0$ then
    MLM_loss ← mean(MLM_loss)                ▷ Compute mean of non-padding tokens
else
    MLM_loss ← 0                 ▷ Set loss to zero if there are no non-padding tokens
end if
return MLM_loss
```

5.2.4. Vocabulary-Free Classification

Once trained, the BERT-based model can be utilised for detecting anomalies in log sequences. The anomalous score of a log sequence can be determined based on the model's predictions for the MASK tokens. For VF-MLM, the anomaly score of a sequence is the loss as measured by the same criterion that is used during training, as shown in Equation (5.5).

To make an actual classification of normal or anomalous, an anomaly score decision threshold is needed for the sequence or element anomaly score, above which a sequence or element is classified as anomalous. The original LogBERT paper uses the test set to find the optimal threshold. While this does show the model's maximum theoretical performance, it is not indicative of real-world performance, as in a real-world situation the labels of the set one needs to classify are unknown.

Therefore, the decision was made to select an optimal threshold on a development set and use that threshold to measure classification performance on the test set. A development set needed to be created instead of just using the validation set, since by design the validation set only contains non-anomalous sequences.

The development set is created by concatenating the validation set with a certain ratio of randomly sampled anomalous sequences from the test set. In this case, a 0.5 sample ratio was taken. The selected sequences are then removed from the test set, so as a consequence, the test set size is reduced.

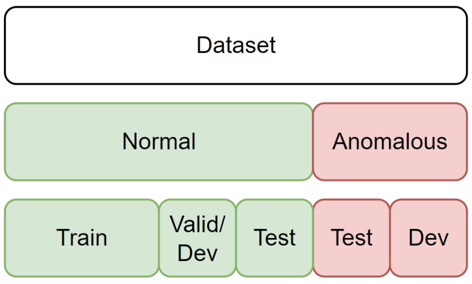


Figure 5.6: The data split used. Red and green show that a set consists of anomalous and normal sequences, respectively. The normal sequences in the development set and validation set are the same. The development set was added compared to LogBERT [26].

5.3. Unseen Logkey Heuristic

The two methods BERT-based methods are compared with a heuristic approach as a baseline. This approach was made to evaluate the effectiveness of this heuristic when data instability increases. By evaluating the effectiveness of a simple heuristic, the difficulty of detecting anomalies in the evaluated situations can be gauged.

The unseen logkey heuristic is rather simple. Instead of using (deep) Machine Learning, the anomaly score is created using a simple heuristic. For the unseen logkey heuristic, the anomaly score of a sequence is the percentage of unseen logkeys in a sequence. For example, if a sequence of length 10 contains one unseen logkey, the anomaly score for this sequence is 0.1 (10%).

To compare the heuristic as fairly as possible to the BERT-based approaches, the procedure is kept as close as possible to the procedure of the BERT-based methods. Therefore, as with the BERT-based approaches, an optimal decision threshold is found on the development set. The performance of the method is thereafter measured on the test set.

Algorithm 3 Anomaly score prediction using the unseen logkey heuristic

```
$S$                                        ▷ List of all sequences to be classified
$V$                                        ▷ All unique logkeys present in the train set (Vocabulary)

for $s_i \in S$ do
    $n \leftarrow |s_i|$                   ▷ Current sequence length
    $x \leftarrow 0$                       ▷ Initialise unseen logkey count
    for $l \in s$ do                       ▷ Iterate over all logkeys in sequence
        if $l \in V$ then
            $x \leftarrow x + 1$
        end if
    end for
    anomaly_score$_i \leftarrow \frac{x}{n}$   ▷ Calculate Anomaly Score for sequence $i$
end for
return anomaly_score
```

6 Evaluation

In this section, the experiment design is shown, including information about the datasets and evaluation metrics. Thereafter, the experiments and their results are presented. After the experiment design, there is a section dedicated to each research sub-question.

6.1. Experiment Design

6.1.1. Datasets

The proposed method is evaluated on the most frequently used public datasets in this field: The HDFS dataset [71], BGL dataset [53], and a small version of the Thunderbird [53] dataset. The public datasets are summarised in Table 6.2, and the case study data is described in Chapter 7.

To make the results comparable, the same pre-processing steps and settings are used as in LogBERT [26]. In a nutshell, this includes the usage of the Drain [30] parser, and grouping using a sliding window of size w_{size} with steps of size w_{step}. If a sequence is longer than $l_{\max}$, it is truncated. If it is shorter than $l_{\min}$, it is discarded. Statistics of the raw datasets can be found in Table 6.2. The settings used for grouping are shown in Table 6.1. Finally, statistics of the datasets after the pre-processing are shown in Table 6.3.

Dataset	w_{size} (mins)	w_{step} (mins)	$l_{\min}$	$l_{\max}$
HDFS [71]	-	-	10	512
BGL [53]	5	1	10	512
TBird Small [26]	1	0.5	10	512

Table 6.1: Pre-processing dataset window and sequence parameters. HDFS uses session window grouping. These are the same settings as used in LogBERT [26]
.

Hadoop Distributed File System (HDFS) The Hadoop Distributed File System (HDFS) [71] dataset is derived from the HDFS that is installed on a high-performance computing cluster featuring 203 nodes that execute a large number of standard MapReduce jobs. The dataset consists of over 24 million logs gathered over two days, and each log sequence corresponds to heterogeneous log events for particular file blocks functioning as session identifiers. Several event sequences represent abnormal execution paths that primarily relate to performance problems, such as write exceptions, and these anomalous logs have been manually labelled. The HDFST dataset is labelled on sequence-level instead of element-level. Furthermore, for the HDFS dataset, instead of using a sliding window to create sequences, session IDs present in each log message are used to group logs together to form sequences. On average, these log sequences have a length of 19 logs. This is short compared to the other datasets and the maximum sequence length of VoBERT and LogBERT, which is 512.

BlueGene/L Supercomputer System (BGL) The BlueGene/L Supercomputer System (BGL) dataset [53] comprises over four million log events that were collected for over 200 days from a Blue-

Gene/L (BGL) supercomputer at the Lawrence Livermore National Labs. These log events have a severity field that enables them to be classified into different categories. In addition to the severity field, system administrators manually labelled the logs. The anomalies that occur in these logs can be attributed to both hardware and software problems. For BGL, a sliding window of 5 minutes is used to generate log sequences. Using this sliding window, the average sequence length before cutting off sequences longer than $l_{\max}$ windows is 562. After trimming these sequences to max $l_{\max}$ in length, the average sequence length becomes 201.

Thunderbird (TBird) The Thunderbird (TBird) [53] dataset also is a big log dataset, collected from a supercomputer system. TBird mini is a reduced-size dataset introduced by [26]. They select the first 20,000,000 log messages from the full Thunderbird dataset. A sliding window is then applied to the Thudnerbird mini dataset to generate log sequences. The average sequence length before cutting off sequences longer than $l_{\max}$ windows is 326. After trimming these sequences to max $l_{\max}$ in length, the average sequence length becomes 164.

Dataset	Data Type	Label Granularity	Number of logs	Of which anomalous
HDFS [71]	Logs	Sequence	11,172,157	284,818 (3%)
BGL [53]	Logs	Element	4,747,963	348,460 (1%)
TBird [53]	Logs	Element	211,212,192	3,248,239 (2%)
TBird small [26]	Logs	Element	20,000,000	758,562 (4%)

Table 6.2: Datasets. 'Number of logs' refers to the number of logs before they are grouped in sequences.

Dataset	Sequences (total)	Sequences (anomalous)	Mean sequence length	Vocabulary size
HDFS [71]	558,223	8,419	19	46
BGL [53]	16,744	1,313	201	857
TBird small [26]	76,208	46,224	164	1,167

Table 6.3: Pre-processed datasets. Using the specified pre-processing pipeline with the settings in Table 6.1

6.1.2. Evaluation Metrics

This work evaluates the anomaly detection performance of the log sequences based on the following metrics: precision, recall, F1 score, and Mathews Correlation Coefficient (MCC) [48]. The MCC score is a more robust metric for evaluating binary classification, which only yields a high score if the prediction could achieve good results in all four confusion matrix categories (true positives, false negatives, true negatives, and false positives) proportionally both to the size of positive and negative elements in the dataset. It is well suited for evaluating binary classification performance on unbalanced datasets, which is often the case when doing anomaly detection. The MCC score produces a more informative and truthful metric in evaluating binary classifications than accuracy and F1 score. According to a log anomaly detection survey [37], previous researches use evaluation metrics such as the F1 score, which is known not to perform when data sets are highly imbalanced [37]. Given that, this research will use the MCC instead.

To compute the aforementioned metrics, choosing a threshold is necessary. To provide a more complete performance overview, a metric without the need for a threshold is also provided: Area Under the Precision-Recall Curve (AUPRC). AUPRC provides a measure of the overall performance which allows for the comparison of the performance of different models, irrespective of the chosen threshold. While the Area Under the Receiver Operating Characteristic (AUROC) is also often used for this purpose, the AUPRC is better suited for situations with unbalanced data [61]. Since log anomaly detection data is highly unbalanced, the AUPRC is used in favour of the AUROC in this study.

Recall (Equation (6.1)) is the ratio of alerts correctly classified as malicious of all malicious alerts.

$$Recall = \frac{TP}{FN + TP} \tag{6.1}$$

Precision (Equation (6.2)) is the ratio of alerts correctly classified as malicious of all alerts classified as malicious.

$$Precision = \frac{TP}{FP + TP} \tag{6.2}$$

The F1 score is the harmonic mean of the Accuracy and Recall:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{6.3}$$

The MCC metric is calculated as in Equation (6.4). MCC values are in the range [-1, 1], and signify the correlation of the predicted output and the labels. An MCC score of 1 is the highest score and means a perfect positive correlation between the predicted output and labels. An MCC score of 0 means that there is no correlation between the output and labels at all, which makes the predictions equivalent to a random guess. -1 signifies a perfect negative correlation, which means that there is an absolute discordance between the output and the labels.

$$MCC = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6.4}$$

6.1.3. Experiment Setup

All experiments were run on a PC with the following specifications:

- CPU: 12th Gen Intel Core i9-12900k
- RAM: 32 GB
- GPU: NVIDIA GeForce RTX 4080 (16 GB)

For reproducibility, the code used to conduct the experiments and the implementation of VoBERT is publicly available: `https://github.com/daanh99/VoBERT`.

6.2. RQ1: Element-Level Evaluation

In this section, sub-research question 1 is answered. This research question is:

RQ1: What influence does each individual logkey have on the sequence anomaly score?

An issue in evaluating current literature was identified in a deep log anomaly detection [21]. Anomaly detection methods are usually evaluated on sequence-level instead of element-level [21]. This means that the whole sequence is classified as normal or anomalous instead of its elements. However, some elements in a log or alert sequence may be normal, even if the sequence as a whole is considered anomalous because it contains a few anomalous elements. Furthermore, evaluation on element-level provides additional insight into the performance of log sequence anomaly detection methods. Additionally, it is useful for analysts to know which specific elements in a log sequence are predicted as anomalous. Hence, this work also evaluates log sequence anomaly detection on element-level.

Element-level evaluation can only be done for datasets labelled on the granularity of single events rather than sequences [37]. The label granularity of the used datasets can be seen in Table 6.2.

6.2.1. Per Element Masking

To perform element-level evaluation, it is necessary to use per-element masking instead of the ratio-masking approach used in LogBERT [26]. The difference between these two methods is explained in Section 5.1.1. An experiment was done comparing the sequence-level anomaly detection performance using ratio masking versus using per-element masking, as well as reproducing the results in the original LogBERT paper. The results can be seen in Table 6.4. It can be seen that the reproduction is within 1 percent point of the performance reported by LogBERT.

	Masking Method	Precision	Recall	F1-Score	MCC	AUROC
Reported in Logbert [26]	Ratio	89.40	92.32	90.83	-	-
Reproduction	Ratio	90.42	92.69	91.54	0.89	0.96
Reproduction	Per element	90.21	93.72	91.93	0.90	0.96

Table 6.4: Performance of the LogBERT [26] reproduction on the full BGL dataset. Evaluated on sequence-level. Parameters: $\alpha = 0.1$ (Equation (4.5)), masking ratio = 0.5, top-g candidates = 15. Note that for a fair comparison, the threshold is optimised on the test set, which is also done in the LogBERT paper [26].

6.2.2. Results

Since HDFS is not labelled on element-level, the element-level evaluation experiments were only done for BGL and TBird.

BGL In Table 6.5, it can be seen that for the BGL dataset, element-level MCC score is higher than sequence-level MCC score in all situations. Anomalous Sequence Only (ASO) element-level MCC score is measured by only considering sequences the model predicted as anomalous. For both LogBERT and VoBERT, ASO element-level evaluation MCC score is lower than regular element-level evaluation for the original train-test split (Instability = Min). For both methods, ASO element-level is higher than regular element-level for the train-test split with maximum data instability (Instability = Max). For both LogBERT and VoBERT, the mean MCC score of the evaluated instability levels is the lowest at sequence level, followed by regular element-level and is highest for ASO element-level.

TBird For the TBird dataset using MLM, the mean MCC score of the evaluated data instability levels is highest at sequence-level. The mean regular and ASO element level MCC scores are comparable. When using VoBERT for this dataset, the mean MCC score of the ASO element-level is highest, and the sequence-level MCC score is comparable to that of the regular element-level. The most significant difference between sequence-level and element-level MCC scores can be observed for the train-test split with maximum data instability (Instability = Max).

			Sequence-level			Element-level			Element-level ASO		
Dataset	Method	Instability	F1	MCC	PRC	F1	MCC	PRC	F1	MCC	PRC
HDFS	LogBERT	Min	68.42	68.16	0.63	-	-	-	-	-	-
		Max	2.26	4.27	0.02	-	-	-	-	-	-
		Mean	21.69	25.02	0.22	-	-	-	-	-	-
	VoBERT	Min	7.89	17.31	0.15	-	-	-	-	-	-
		Max	2.23	3.91	0.02	-	-	-	-	-	-
		Mean	4.53	9.94	0.06	-	-	-	-	-	-
	Heuristic	Min	41.87	50.74	0.62	-	-	-	-	-	-
		Max	2.25	4.33	0.02	-	-	-	-	-	-
		Mean	15.42	23.19	0.22	-	-	-	-	-	-
BGL	LogBERT	Min	85.49	83.72	0.94	92.98	92.15	0.89	93.11	81.16	0.91
		Max	25.56	17.45	0.27	25.89	23.15	0.22	84.78	53.14	0.77
		Mean	45.34	42.27	0.53	59.43	57.65	0.55	88.95	67.15	0.84
	VoBERT	Min	49.10	45.02	0.45	68.34	66.84	0.39	86.97	61.22	0.68
		Max	23.41	10.53	0.12	22.65	17.48	0.11	82.66	44.52	0.67
		Mean	36.34	30.17	0.25	45.49	42.16	0.25	84.82	52.87	0.68
	Heuristic	Min	92.56	91.93	0.94	-	-	-	-	-	-
		Max	22.29	10.13	0.42	-	-	-	-	-	-
		Mean	43.48	39.35	0.56	-	-	-	-	-	-
TBird	LogBERT	Min	78.74	72.92	0.98	61.06	64.26	0.47	61.46	61.61	0.47
		Max	18.79	2.67	0.26	12.14	14.69	0.06	23.51	18.15	0.14
		Mean	56.52	44.90	0.62	36.60	39.48	0.27	42.48	39.88	0.30
	VoBERT	Min	53.26	38.12	0.43	33.25	39.45	0.14	47.37	47.19	0.25
		Max	48.01	29.98	0.42	30.55	37.35	0.13	59.94	59.64	0.40
		Mean	53.98	39.57	0.47	31.90	38.40	0.14	53.66	53.42	0.32
	Heuristic	Min	98.46	97.95	1.00	-	-	-	-	-	-
		Max	40.53	9.26	0.51	-	-	-	-	-	-
		Mean	63.09	49.27	0.69	-	-	-	-	-	-

Table 6.5: Performance comparison between VoBERT, LogBERT, and the unseen logkey heuristic Section 5.3. Instability: Min= As in train/test split used in original LogBERT [26] paper, Max= Most unstable situation generated by the method described in Section 6.3.1, Mean= Mean of all instability levels. ASO = Anomalous Sequences Only (as predicted by the model). PRC = AUPRC.

6.2.3. Discussion

Comparing element-level and sequence-level evaluations, the former outperforms on the BGL dataset but underperforms on the TBird dataset when looking at the mean MCC score. We speculate that different characteristics of the datasets could cause this.

For BGL, the reason that element-level evaluation performance is higher than sequence-level might be that the model is often correct on the element-level for a particular sequence, but the sequence anomaly threshold r is not correct for that particular sequence. In that situation, the model has a higher element-level than sequence-level performance.

For TBird, the reason that element-level evaluation performance is lower than sequence-level might be that the model is often wrong on the element-level for a particular sequence, but the sequence anomaly threshold r is still set in such a way that the prediction on sequence-level is correct. In this situation, the model would have a higher sequence-level than element-level performance.

While the element-level evaluation provides insight into which specific logs are anomalous in a sequence, it comes at a high computational cost. The time complexity of making a classification goes from $\mathcal{O}(1)$ to $\mathcal{O}(n)$ in terms of sequence length n. In practice, a good compromise might be only to make element-level predictions on sequences that the model has predicted as being anomalous. This would make sense in a real-world scenario since log analysts are naturally more interested in anomalous than normal log sequences. Furthermore, element-level evaluation is important to improve explainability.

Note that in Table 6.5, the LogBERT performance on the original BGL train-test split (BGL, LogBERT, No) is different from the performance reported in Table 6.4. The reason for this difference is that for the experiments in Table 6.4 the threshold is optimised for the test set in order to allow for direct comparison to the LogBERT paper. However, all other experiments (including Table 6.5) optimise the threshold on a separate development set to better represent the model's performance in real-world scenarios. This explains the slightly lower performance in Table 6.5.

6.3. RQ2: Unstable Logkeys

In this section, sub-research question 2 is answered. This research question is:

RQ2: How to make an anomaly-detecting method adapt to logkeys that were not seen during training?

To answer this research question, traditional MLM and VF-MLM will be compared by evaluating LogBERT and VoBERT on three public datasets. First, the performance will be compared on the train/test split as used in the LogBERT paper [26]. Then, the performance of both methods will be measured for other train/test splits with increasing data instability. The data instability is increased by only reassigning sequences in the train and test set while keeping the train and test set sizes fixed. We show that VoBERT is more stable than LogBERT when data instability is increased and that the performance of LogBERT is not actually as good as presented in [26].

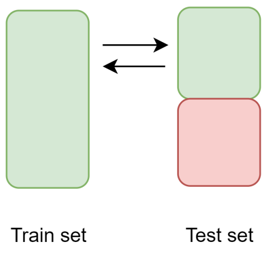


Figure 6.1: Data redistribution process. Green represents normal sequences, and red represents anomalous sequences.

6.3.1. Data Redistribution Algorithm

To increase the log data instability, the test data should contain more sequences that contain unseen logkeys, i.e., logkeys present in the test set that never occurred in any sequence in the train set. For the three public datasets, we found that a large portion of the anomalous sequences in the test set already contained unseen logkeys. This contrasts the normal sequences, of which almost none contain unseen logkeys.

To increase the number of sequences, the percentage of normal test sequences containing unseen logkeys must be increased since the number of anomalous test sequences can not be changed by much in our case. The number of anomalous test sequences containing unseen logkeys can never be decreased. That would mean that sequences containing logkeys occurring only in the anomalous test must be moved to the train set, which is not allowed to contain anomalous sequences by design. The number of anomalous test sequences containing unseen logkeys can not be increased much since the three public datasets used in the evaluation is already close to 100% of the sequences.

The percentage of normal test sequences containing unseen logkeys is increased by reassigning sequences between the train set and the normal part of the test set, as shown in Figure 6.1. To achieve this, a data redistribution algorithm was created that starts from the original train/test split as used in [26] and gradually increases the data instability by increasing the percentage of normal sequences containing unseen elements. The test and train set sizes are kept fixed during this process. The algorithm data redistribution algorithm is shown in Algorithm 4.

The size of the train and test set are equal to the sizes used in the LogBERT paper and not changed when data is redistributed. For all datasets, the train size is rather small. The train test split sizes are shown in Table 6.6.

Dataset	Train Size (seqs)	Test Size (seqs)	Normal seqs ≥ 1 unseen	Anomalous seqs ≥ 1 unseen
BGL	6.697 (37%)	11.360 (63%)	0.13%	86.75%
TBird	6.000 (6%)	93.320 (94%)	0.88%	99.16%
HDFS	4.855 (0.86%)	561.787 (99.14%)	0.01%	36.33%

Table 6.6: Train and Test set sizes as used in LogBERT [26]. Normal and anomalous sequences containing unseen logkeys in the test set are shown. For consistency, these sizes are also used in this thesis. Note that the size is expressed as the number of sequences, not individual logkeys. Additional statistics can be seen in Appendix A.

Algorithm 4 Data redistribution algorithm. This algorithm increases the percentage of sequences containing ≥ 1 unseen element while keeping the training and test sizes fixed, starting with the original train/test split. V_S: Vocabulary of set of sequences S
.

```
train                                              ▷ The train set
test_normal                                        ▷ The normal portion of the test set
test_abnormal                                      ▷ The abnormal portion of the test set

save_to_file(train, test_normal, test_abnormal)    ▷ Save the original split

$all\_normal \leftarrow$ train $\cup$ test_normal
$train\_vocab \leftarrow$ unique_elements($train$)

$N \leftarrow |\text{train}|$
$\Delta N \leftarrow |all\_samples| - |train|$
for $e_i \in train\_vocab$ do
    $R \leftarrow$ occur($all\_normal$, $e_i$)     ▷ Which sequences of all_normal contain element $e_i$
    if $|R| > 0$ then
        $S_i \leftarrow R$
    end if
end for
splits $\leftarrow 10$                              ▷ The number of train/test splits to generate
$S \leftarrow S$.sort($|S_i|$)
$n \leftarrow 0$
while $n \leq \Delta N$ do
    $S_{\text{head}} \leftarrow$ first_N_unique_items($S$, $n$)   ▷ Returns minimal set of sequences $S$ so that $V_S \geq n$
    $train_i \leftarrow all\_normal$
    Remove $S_{head}$ from $train_i$
    $test\_normal_i \leftarrow s_{head}$
    if $|train_i| > N$ then                        ▷ Reduce train size to N
        selected, not_selected$\leftarrow$ sample($train_i$, size=N)
        $train_i \leftarrow$ selected
        $test\_normal_i \leftarrow$ not_selected
    end if
    save_to_file(train$_i$, test_normal)
    $n \leftarrow n + (\frac{1}{\text{splits}} \times \Delta N)$
end while
```

6.3.2. Increasing Data Instability

Statistics describing the different train/tests for the datasets obtained using this method can be seen in Appendix A. In the figures, the x-axis identifies the different train/test splits, and $x = 0$ is the original train/test split as used in the LogBERT [26] paper. For the BGL set, Figure A.1c shows the percentage of sequences of the test set that contain at least one unseen element. This is the statistic that is used as a measure of how dynamic the logkey train/test split is in the rest of this experiment. The vocabulary size, i.e. the number of unique elements in each set is shown in Figure A.1b. Finally, Figure A.1a shows the percentage of test set vocabulary that is unseen, i.e. the percentage of unique test set elements that do not occur in the train set. Figures A.2 and A.3 show the same statistics, but for TBird and HDFS, respectively.

It is important to note that for all three datasets, the original train/test split used by LogBERT [26] greatly favours the traditional MLM method used by LogBERT. For example, for BGL it can be seen in Figure A.1a and Figure A.1c that the normal sequences in the test set contain almost no unseen logkeys, while almost all anomalous sequences in the test set contain unseen logkeys. The same is true for the other datasets tested in this study.

This is highly beneficial for the traditional MLM method used by LogBERT since LogBERT inherently has a very high chance of classifying sequences with unseen logkeys as anomalous. For the original train/test split, using this heuristic results in a good performance.

LogBERT has a high chance of classifying sequences containing unseen logkeys as anomalous because if it encounters an unseen logkey it is embedded as an [UNK] token (explained in Section 4.2). Because by design an [UNK] token is never observed by the model during training, the model is very unlikely to predict a masked [UNK] token correctly. Even if the other masked tokens in a log sequence are regular logkey tokens, the prediction of these tokens is made harder because the [UNKNOWN] token provides no valuable information on which to base the prediction. This effect is increased if more logkeys in a sequence are [UNK] tokens.

For all of the increasingly unstable train/test splits for the datasets, the performance of LogBERT and VoBERT is computed according to the metrics described in Section 6.1.2. The best threshold is found by optimising for the biggest Mathews Correlation Coefficient (MCC) score on the development set, and the MCC score obtained by using that threshold on the test set is plotted.

6.3.3. Results

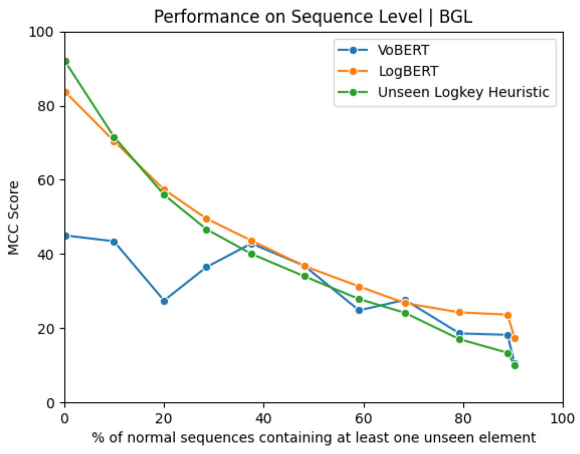
The results of running both LogBERT and VoBERT on increasingly unstable log data can be seen in Figure 6.2. The MCC score at the leftmost point (closest to $x = 0$) represents the MCC score when the model is run on the original train/test split as used in LogBERT [26]. The settings of the models used can be seen in Table 6.7

Dataset	Loss (for VF-MLM)	Loss (for MLM)	top-g (for MLM)	Mask ratio
BGL	MSE	Cross-Entropy Loss	15	0.5
HDFS	MSE	Cross-Entropy Loss	6	0.65
TBird	MSE	Cross-Entropy Loss	15	0.5

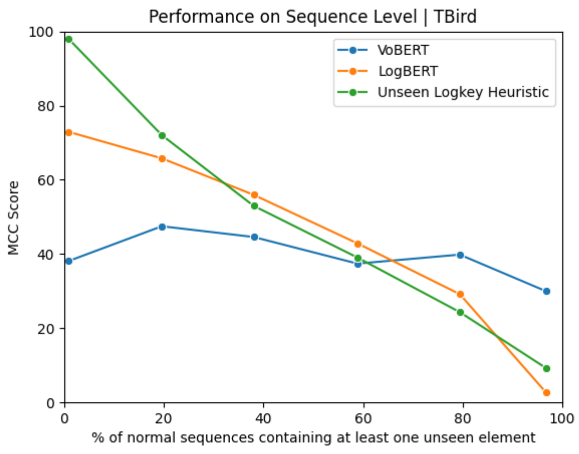
Table 6.7: Settings used for the experiments per dataset.

For all datasets, it can be observed that LogBERT is susceptible to significant performance degradation when log data instability increases. When comparing the initial train/test split presented in the LogBERT paper [26] and the reassigned train-test split of similar size with maximum data instability, the performance of LogBERT (MLM) decreased with 94%, 79%, and 95% for the HDFS, BGL, and TBird dataset, respectively. If we make the unseen logkey heuristic less effective by applying more iterations of the data redistribution algorithm, we can see that the performance of LogBERT and the unseen logkey heuristic converge. This suggests that LogBERT's higher performance relies on the unseen logkey heuristic. This conclusion is strengthened by the fact that LogBERT's performance converges with VoBERT's performance when the unseen logkey heuristic becomes less effective. VoBERT can be seen as LogBERT with the added ability to deal with unseen logkeys properly instead of always classifying them as anomalous. This difference inherently prevents VoBERT from using the unseen logkey heuristic.

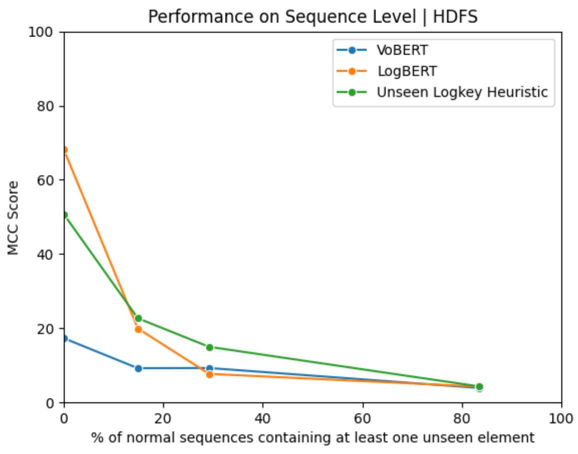
Note that the unseen logkey heuristic significantly outperforms LogBERT and VoBERT for the BGL



(a) BGL dataset Mathews Correlation Coefficient for increasing log data instability (sequence-level).



(b) Thunderbird dataset Mathews Correlation Coefficient for increasing log data instability (sequence-level).



(c) HDFS dataset Mathews Correlation Coefficient for increasing log data instability (sequence-level).

Figure 6.2: VoBERT, LogBERT, and the unseen logkey heuristic predictive performance (Mathews Correlation Coefficient) for increasingly unstable data generated with the data redistribution algorithm (Section 6.3.1).

and TBird dataset when the specific train-test split of the LogBERT paper [26] is used. It achieves an MCC score of close to 100% for TBird and around 90% for BGL. It can be concluded that the train-test splits used in LogBERT [26] are not representative of anomaly detection performance on the evaluated datasets since the simple unseen logkey heuristic obtains a comparable or higher MCC score. Additionally, it is observed that the performance of LogBERT drops significantly for other train-test splits.

BGL For the BGL dataset, it can be seen that the performance graph of LogBERT (MLM) is very similar to that of the unseen logkey heuristic. The performance of VoBERT (VF-MLM) is lower for the original dataset, which makes sense as VF-MLM can inherently not use the unseen logkey heuristic. After around 40%, the performance of VoBERT follows the performance of LogBERT and the unseen logkey heuristic.

TBird For the TBird dataset, the performance of VoBERT is more stable than that of logBERT and the unseen logkey heuristic. As with the BGL dataset, the performance of LogBERT is very similar to that of the unseen logkey heuristic. The performance of VoBERT, however, clearly outperforms LogBERT and the unseen logkey heuristic when the data instability becomes greater than 70% of sequences containing unseen logkeys.

HDFS For the HDFS dataset, the data suggest the performance reported in the LogBERT paper [26] relied on anomalous sequences containing unseen logkeys and the absence of sequences containing unseen logkeys in the test set. As the amount of normal sequences containing unseen logkeys in the test set increases, the MCC score drops sharply. After 30% of normal sequences in the test set contain unseen logkeys, the MCC score of LogBERT, VoBERT, and the unseen logkey heuristic drops below 20%.

6.3.4. Discussion

The results show that the performance of LogBERT degrades significantly when data instability is increased. The proposed more robust solution, VoBERT, is shown to remain more stable when data instability is increased. However, the performance of LogBERT (using traditional MLM) is higher than SequanoBERT (using VF-MLM) for the train-test splits with a lower data instability for all datasets. These are the train-test splits with a large difference between the percentage of normal and anomalous sequences containing unseen logkeys. For this performance discrepancy, we have identified the following potential reasons:

Unseen Logkey Heuristic The suspected main cause for the worse performance of VoBERT in situations with limited or no data instability (e.g. few unseen logkeys) could be that LogBERT greatly benefits from using the unseen logkey heuristic in these situations. This could mean that LogBERT is actually not able to learn the difference between normal and anomalous log sequences but instead relies on the unseen logkey heuristic (classifying sequences containing [UNKNOWN] tokens as anomalous). VoBERT is inherently designed not to use this heuristic, as there are no [UNKNOWN] tokens in VF-MLM.

The results also align with this hypothesis since the performance of LogBERT decreases when the unseen logkey heuristic becomes less effective. At the same time, the performance of VoBERT stays relatively stable, which is expected since VoBERT can not use the unseen logkey heuristic by design. As expected, the performance eventually converges in the most unstable environments. For all three evaluated datasets (HDFS, BGL, TBird), the anomalous part of the test set already contains a high (>90%) percentage of sequences containing at least one unseen logkey. With every iteration of the dataset redistribution algorithm, the percentage of unseen logkeys in the normal part of the test set is increased. By decreasing the difference in the number of unseen logkeys between the normal and anomalous part of the test set, the unseen logkey heuristic becomes less and less effective.

[37] found that it is far from challenging to achieve competitive detection rates on one of the most frequently used evaluation datasets: HDFS [72]. Namely, the combination of a sequence length heuristic and using the simple Stide algorithm [22] that moves a sliding window of a given length over the data and looks for sub-sequences that have not been seen before in the training data could produce competitive performance [37]. The survey's findings affirm the hypothesis that, at least for HDFS, the reported results in the LogBERT heavily rely on the unseen logkey heuristic instead of the model actually learning to discern normal and anomalous sequences.

This is also in line with the sharp drop in performance observed in Figure 6.2c. Since the average sequence length for the HDFS dataset after pre-processing is only 19 logkeys, this is probably too short of a sequence to learn anything meaningful. In any case, for this specific dataset, it can be concluded that using a heuristic such as sequence length and Stride as described in [37] is the most appropriate approach.

Proxy Choice The percentage of sequences containing at least one unseen logkey might not be the best proxy for log data instability. Other proxies could also be imagined, such as % of all raw logkeys in the test set that is unseen (while counting duplicate logkeys). It could be that the effect might be better observed using an alternative proxy for log data instability.

Amount of BERT Layers For LogBERT, which uses traditional MLM, the initial embeddings are randomly generated before training starts. However, the embeddings are updated along with the rest of the model during backpropagation in the training phase. The embeddings remain static in VF-MLM, so VoBERT might need an extra BERT layer to compensate for the loss of trainable parameters.

6.4. Sensitivity Analysis

After the main experiments were completed, a sensitivity analysis for VoBERT was conducted. In the sensitivity analysis, the influence of loss functions, train set size, and parsing as pre-processing steps.

Category	Setting	Value 1	Value 2	Value 3
VF-MLM Loss Function	Loss Function	L1 Loss (MAE)	L2 Loss (MSE)	CosineSimilarity
	Include VHM	Yes	No	
Pre-processing	Training size	As in LogBERT [26]	80%-20% train-test	
	Parsing	Yes	No	

Table 6.8: Variables and their settings evaluated as sensitivity analysis

6.4.1. Loss Function

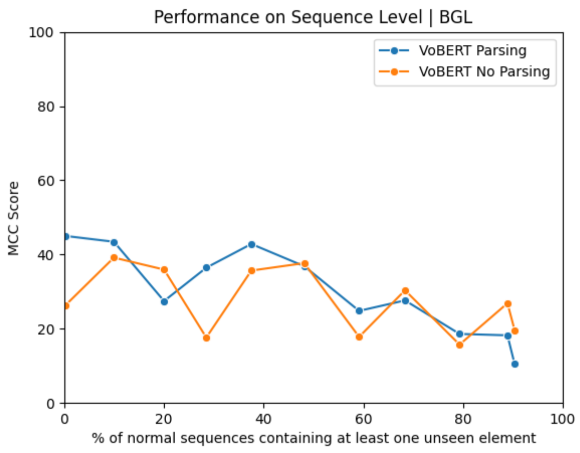
The results of testing different loss functions for VF-MLM can be seen in Table 6.9. All experiments were run on the original train/test split as used in LogBERT [26]. As visible from the table, the best results were obtained using Mean Squared Error (MSE) as a loss function without Volume of Hypersphere Minimisation (VHM). This setting is thus used for all other experiments.

The loss function used in LogBERT [26] is L1 loss with VHM. It should be noted that due to time constraints, no hyperparameter tuning was performed for the alternative loss functions. The lack of this will likely negatively influence the alternative loss functions.

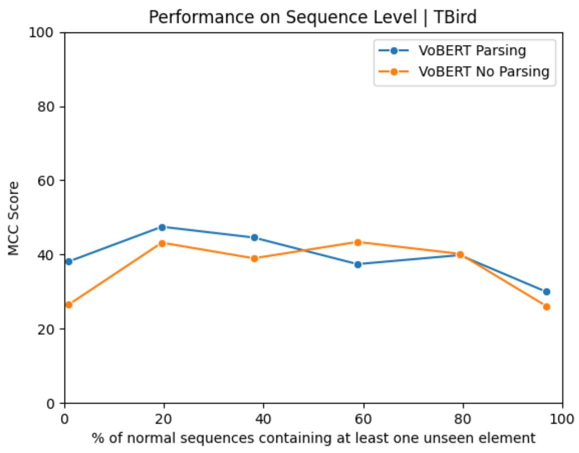
Loss Function	F1	MCC	AUROC
VHM + L1 (MAE)	57.92	53.24	0.88
VHM + L2 (MSE)	51.94	47.27	0.86
VHM + CosineSimilarity	32.06	26.95	0.70
L1 (MAE)	57.41	52.92	0.88
L2 (MSE)	60.10	55.58	0.90
CosineSimilarity	23.78	11.54	0.53

Table 6.9: Sequence-level Performance on the BGL dataset of VF-MLM tested for different loss functions. All experiments run on the original BGL train-test split as in LogBERT [26] with threshold selection on a development set.

6.4.2. Parsing



(a) Mathews Correlation Coefficient (MCC) of VoBERT with and without parsing as a pre-processing step on BGL dataset (sequence-level).



(b) Mathews Correlation Coefficient (MCC) of VoBERT with and without parsing as a pre-processing step on TBird dataset (sequence-level).

Figure 6.3: Sensitivity analysis for the effect of parsing on VoBERT. Anomaly detection performance for increasingly unstable data generated with the data redistribution algorithm (Section 6.3.1).

One of the pre-processing steps of VoBERT is to parse the logkeys. To test the influence of this pre-processing step on the predictive performance of VoBERT, additional experiments were conducted using the same settings as the main results presented in Section 6.3, but without the parsing step during pre-processing. The results are presented in Figure 6.3. In this graph, it can be seen that the average performance does not vary significantly when not using parsing. However, using parsing, the MCC score is slightly higher for both TBird and BGL.

			Parsing (Sequence-level)			No Parsing (Sequence-level)		
Dataset	Method	Instability	F1	MCC	AUPRC	F1	MCC	AUPRC
BGL	VoBERT	Min	49.10	45.02	0.45	31.33	26.21	0.29
		Max	23.41	10.53	0.12	27.42	19.67	0.16
		Mean	36.34	30.17	0.25	33.09	27.52	0.29
TBird	VoBERT	Min	53.26	38.12	0.43	46.36	26.50	0.29
		Max	48.01	29.98	0.42	46.08	26.15	0.41
		Mean	53.98	39.57	0.47	52.04	36.40	0.40

Table 6.10: Comparing VoBERT performance with and without parsing as a pre-processing step. Stable As in train/test split used in original LogBERT [26] paper, Unstable: Most unstable situation generated by the method described in Section 6.3.1. Mean: mean performance of all train-test splits.

6.4.3. Train-test split size

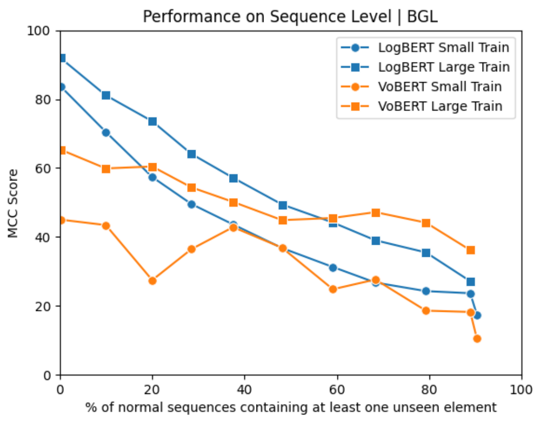


Figure 6.4: Sensitivity analysis of train sizes. Train-test splits used: Small train= 37%-63%, Large train= 80%-20%.

The LogBERT [26] paper uses a small train set compared to the test set, as seen in Table 6.6. While this might be sufficient for LogBERT since it can use the unseen logkeys heuristic, VoBERT might benefit from a more traditional split such as (80%-20%). Therefore, the experiment was repeated with an 80%-20% train-test split for the BGL dataset. The results of this experiment are shown in Figure 6.4.

As seen in Figure 6.4, the performance of both LogBERT and VoBERT is better across all data instability levels when using the larger trainset. The increase in performance of VoBERT is more extensive when using the large train size than the increase in performance of LogBERT. Consequently, the performance gap between LogBERT and VoBERT is smaller when using a larger trainset (80%-20%) for the BGL dataset.

For train-test splits where the unseen logkey heuristic is effective, the larger performance increase of VoBERT could be explained by the fact that LogBERT relies on the unseen logkey heuristic, so it does not benefit significantly from extra training data. However, it should be noted that VoBERT's performance increase could also be greater than LogBERT simply because it has a lower starting performance, which is easier to increase than a higher starting performance.

			Small Train Size			Large Train Size		
Dataset	Method	Instability	F1	MCC	AUPRC	F1	MCC	AUPRC
BGL	VoBERT	Min	49.10	45.02	0.45	75.50	65.34	0.88
		Max	23.41	10.53	0.12	62.47	45.50	0.78
		Mean	36.34	30.17	0.25	68.22	54.37	0.82
	LogBERT	Min	85.49	83.72	0.94	94.11	91.93	0.96
		Max	25.56	17.45	0.27	50.68	27.15	0.75
		Mean	45.34	42.27	0.53	69.00	56.32	0.85

Table 6.11: Sensitivity analysis of train-test split sizes for BGL and TBird.

7 Case Study: Security Events

In this case study, we answer sub-question 3:

> *Is it possible to apply the log anomaly detection technique to detect anomalies in real-world security event logs?*

7.1. Case Study Motivation

There has been a significant increase in the number of cyber-attacks and unauthorised activity on computer networks in recent years [50]. As more and more services become accessible through the internet, it is vital to ensure information security and protect against these attacks. A common method is using an Intrusion Detection System (IDS). IDSs are vital for ensuring information security and against unauthorised activity on computer networks. According to [5], Intrusion detection is "the process of monitoring the events occurring in a computer system or network and analysing them for signs of intrusions, defined as attempts to compromise the confidentiality, integrity, availability, or to bypass the security mechanisms of a computer or network." IDSs are hardware or software products that automate this monitoring and analysis. Intrusion detection alert analysis is an attractive and active topic [58]. Many researchers have been working in this field recently, and there exist several recent surveys, literature reviews, and Systemic Mapping Studies such as [58, 51, 36, 41, 50].

A significant challenge within IDSs is the high number of alerts they generate, many of which are false positives [44]. In fact, [3] found that 99% of IDS alerts are false positives and concluded that analysts often spend a significant amount of time manually reviewing alarms to determine their validity. This significant manual effort often leads to alert fatigue [28] and decreased responsiveness from security analysts [16].

[18] found that companies deal with an average of 174,000 alerts per week, of which only 12,000 could be manually inspected. Furthermore, Cisco [1] reported in 2019 that in 41% of 3,540 organisations included in the research, over 10,000 alerts per day were generated. Just 50.7% of the generated alerts were manually inspected because of the limited amount of security operators available. Of the manually inspected alerts, only 24.1% were actual attacks; the rest were false positives. Another reason for a large number of alerts is that in a monitored network, often multiple different IDS systems, sensors, and other detection tools that all generate alerts are installed. This often leads to many duplicated alerts [69].

There has been tremendous progress in automating parts of the alert analysis workload in recent years. Various methods have been proposed to reduce false positives. Some of these methods involve different configurations of IDSs, while most of them propose the post-processing of alerts [58]. Despite these efforts, the large number of alerts is still a problem [50].

Advancements in deep learning have led to sequential models capable of capturing temporal and contextual information for alert and log anomaly detection [37]. The problem of alert anomaly detection is essentially a problem of discrete sequence anomaly detection. Recently, there has been a surge in sequential deep learning methods [37]. Apart from Security Event (alert) analysis, these methods aim at a similar task: Log event anomaly detection. They can capture the temporal and context information of event sequences and are well-suited for alert and log anomaly detection.

From a structural point of view, logs can be seen as special alerts that only consist of the alert description attribute. Oftentimes, IDS systems ingest logs and, through subsequent analysis, can then flag which logs are suspicious in the form of alerts.

While much research is done into Alert analysis, there are still many open challenges in this area. Over the years, many traditional machine learning models have been created to identify anomalous alerts [41]. These methods extract features from alert sequences before feeding these features to an ML model. Since there is often a significant imbalance between available training data for normal and anomalous alerts, it is often not possible to use a supervised approach.

Therefore, many one-class classification or unsupervised models are used for alert analysis, such as one-class Support Vector Machine (SVM) [42] or principal component analysis (PCA). However, traditional machine learning models do not capture the temporal sequence information of the alert sequence. Furthermore, since temporal sequence information is often lost in the feature extraction step, it is often not possible to show what specific alerts have let to a classification as normal or anomalous. Explainability techniques can help to partly alleviate this problem by providing local explainability on a feature basis but are not able to pinpoint the influence of specific alerts. Missing this information is a significant disadvantage since it can be very helpful to security analysts, for example, to do Root Cause Analysis [60].

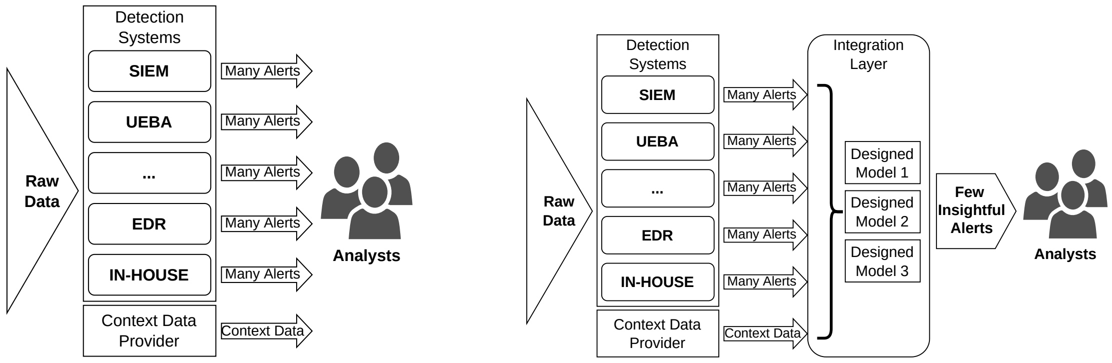
The effectiveness of traditional and vocabulary-free MLM is evaluated on real-world alert data collected at a large international bank (50,000+ employees), hereafter called 'The bank'. To our knowledge, this is the first work evaluating the effectiveness of log analysis techniques for alert analysis.

7.2. Case Study Problem Context

The problem that alert analysis tries to solve is detecting malicious behaviour in a stream of alerts generated by different IDSs. The goal is to timely detect and warn Security Analysts when an attack is happening and provide helpful information about what alerts are related to this detected attack. This will reduce alert fatigue for security analysts by reducing the number of false positive alerts.

Apart from reducing false positives, a large benefit of having an Alert Analysis system in place is the ability to combine multiple alerts and aggregate them into one important alert, often called a hyper-alert. This can augment detection by enabling the company to successfully recognise attacks by combining several alerts that are not sufficiently suspicious on their own, but their joint presence is and yields a detection that would otherwise not have occurred.

A typical situation with and without alert analysis in place can be seen in Figure 7.1



(a) A typical Intrusion Detection setup without Alert Analysis. **(b)** A typical Intrusion Detection setup with Alert Analysis.

Figure 7.1: A typical Intrusion Detection setup without (left) and with (right) alert analysis.

A typical alert analysis set-up is fed by multiple IDSs. Such a setup is called heterogeneous alert analysis, as opposed to homogeneous alert analysis. With heterogeneous alert analysis, the alerts of all of these systems are often collected centrally. This is also the situation in this case study, depicted in Figure 7.1.

The alert analysis pipeline typically consists of three tasks: Normalisation, Grouping, and Sequence Classification. An overview of the common anomaly detection with deep learning pipeline can be seen in Figure 7.2. This case study focuses on the final step: Sequence Classification.

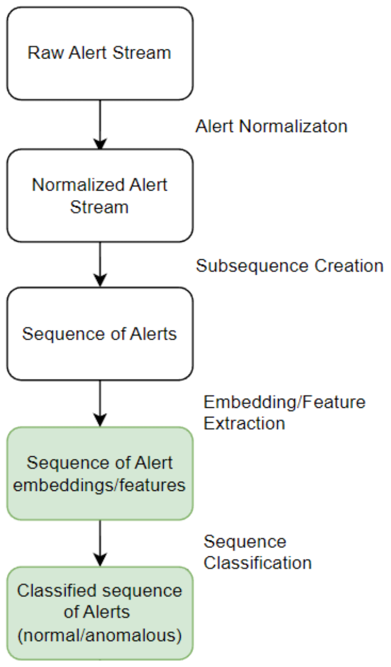


Figure 7.2: A typical Alert Anomaly Detection pipeline. The focus of this work is highlighted in green.

Since there is a big unbalance between available training data of alerts, a semi-supervised approach is well suited. That is why the assumption is made that anomalous alerts are likely attacks and normal alerts are more likely not.

7.2.1. Formal Definition

The typical security event analysis pipeline consists of several steps, explained in this section.

Normalisation

- Input: $L = (l_1, l_2, ..., l_n)$ where L is the sequence consisting of all raw alerts, and l_i is the i-th alert in this collection.
- Output: $A = (a_1, ...a_n)$ where A is the sequence consisting of all normalised alerts, and a_i is the i-th alert in this collection.

First, alerts from different IDSs are merged into one stream and transformed into a common format. This process is called Alert Normalisation. Oftentimes, the same network is monitored by multiple kinds of IDSs from different vendors or other security systems such as firewalls and antivirus programs. These systems all produce alerts in a different format [64]. To enable alert analysis across multiple alert sources (known as heterogeneous analysis/correlation), these raw alerts must be converted to the same format. For this reason, alert standards have been developed, such as the intrusion detection message exchange format (IDMEF) [17].

The output of the normalisation step is a collection of alerts in a common format containing different attributes. The specific attributes of alerts in this case study are specified in Table 7.1.

Grouping

- Input: $A = (a_1, ...a_n)$ where A is the sequence consisting of all normalised alerts, and a_i is the i-th alert in this collection.
- Output: $S = (s_1, ..., s_n)$ where s_i is a sub-sequence of alerts, and S is a collection of grouped sub-sequences of alerts.

The goal of the grouping step is to group related alerts together so that these groups can be further analysed in subsequent steps. This is often done in two main ways: Time splitting and Correlation. They can be done subsequently, or combined into a single step.

Time splitting In this step, the stream of alerts is split into sequences based on the timestamp of the alerts. The time window selection problem is the challenge of selecting the optimal time window length. This is dependent on multiple factors, such as the environment the IDS is deployed in, the rate at which new alerts come in, and the type of attack that the solution aims to detect. Some attacks are spread over multiple days, such as Advanced Persistent Threats (APTs). For these types of attacks, longer time windows are better suited. However, longer time windows might make sequences too long in some environments. All alerts are split into sequences of a fixed time period of length l. The input of this step is a sequence of alerts A. The output, S, is a set of sub-sequences of A, where the alerts in A are split based on what time window they fall in, as shown in Equation (7.1).

$$S = \{s_1, ..., s_n\} \tag{7.1}$$

Where s_i is a sub-sequence of alerts. Each $s_i \subseteq A$. Each sub-sequence consists of alerts of which the timestamp falls within the range of the time window of that sub-sequence.

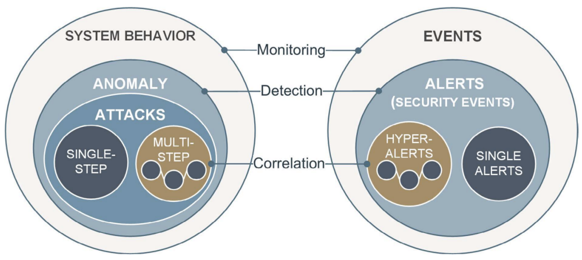


Figure 7.3: The role of Monitoring, Detection, and Correlation in Alert Analysis [36]

Alert Correlation In the alert correlation step, a sequence of raw alerts is grouped into subgroups of related alerts. Alert Correlation aims to decrease the number of alerts by bundling correlated alerts into subgroups and increase the quality of information about alerts by providing context and insights into which and how alerts are related [25]. In figure 7.3, the role of Alert Correlation in Alert Analysis can be seen. This is an active research field, and many recent surveys, literature reviews, and Systemic Mapping Studies exist, such as [58, 51, 36, 41].

There are three main categories of alert correlation methods: similarity-based, step-based, and mixed strategies [41]. Similarity-based correlation is based on defining some metric with which the distance between two alerts is computed. Then, a clustering algorithm can be used to create clusters of similar alerts. Step-based correlation groups alert together that belong to the same chain of events. This could be used to reconstruct a user's actions. Reconstructing a chain of events can either be done using predetermined information or statistical relationships. Strategies based on predetermined information could, for example, use attack scenarios and vulnerability knowledge bases as sources of predetermined information [41]. Statistical relationship-based methods do not use predetermined knowledge but instead use statistical models such as Bayesian networks and regression analysis [41].

The output of this step is a collection of alert sub-sequences S, where each sub-sequence s_i is a sequence of related alerts $s_i \in S$. The exact meaning of "related" depends on the choice of alert correlation method.

Sequence Classification

- Input: $S = (s_1, ..., s_n)$ where each s_i is the i-th sub-sequence of grouped alerts.
- Output: $S_{pred} = \{(s_1, y_1), ..., (s_n, y_n)\}$ where y_i is a label (malicious or benign) for alert sub-sequence s_i.

The final step of the alert analysis pipeline is often to detect relevant alerts. The sequences of alerts produced by the previous step need to be classified as either requiring further action or as safe to be ignored. Since, in most setups, real alerts requiring further action are much rarer than those that do not, this process can also be seen as anomaly detection. In this view, the relevant alerts requiring further action are seen as anomalies and the alerts that do not are seen as normal. The output of this task is a binary label for every sequence.

$$S_{pred} = \{(s_1, y_1), ..., (s_n, y_n)\} \tag{7.2}$$

7.3. Case study Current Situation

The current situation at the bank was first researched and is summarised in this section. The bank has built its own Alert Analysis module that is actively employed on top of its traditional SoC set-up in a production environment. The Alert Analysis module built by the bank is called the Integration Layer and is based on a traditional ML technique: Gradient Boosting. As shown in Figure 7.4, the Integration Layer (IL) collects alerts from various sources and reports its findings back to the SIEM system so that the Incident Response Team (IRT) can handle suspicious alerts. The IRT labels the suspicious alerts as either an actual threat or a false alarm, and the Alert Analysis module is re-trained on this new data.

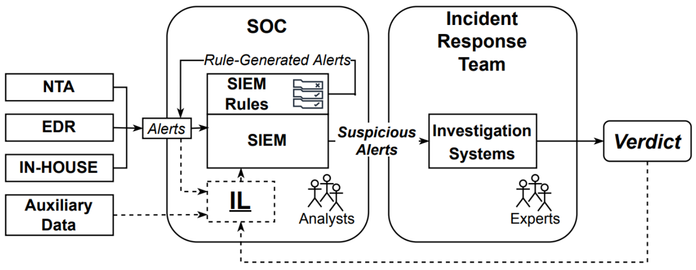


Figure 7.4: The alert analysis situation at the bank. The traditional infrastructure is shown with full lines, and the custom Alert Analysis-related infrastructure, called Integration Layer (IL), is shown with dashed lines.

7.3.1. Normalisation

The goal of this task is to collect alerts from different sources and transform them into a common format. The situation at the bank can be seen in Figure 7.4. The bank collects alerts generated by various systems: Security Information and Event Management (SIEM), Network Traffic Analysis (NTA), Endpoint Detection and Response (EDR), and a custom security system they developed in-house. All of these systems are created by different vendors, highlighting the importance of alert normalisation. After normalising an alert, its original source is added as a new attribute for use in subsequent steps.

In addition to normalising all alerts from Intrusion Detection Systems, the Alert Analysis module the bank has developed (IL in Figure 7.4) also fetches auxiliary data to enrich the alerts. For example, the bank fetches information about the users' Internet connections from the corporate web-proxy server. This information is added to the alerts as a new attribute.

The output of this task is a sequence of alerts with attributes as specified in Table 7.1. A more detailed version of this table can be found in Appendix B.

Attribute	Type
Alert id	String
Alert description	String
Alert source (which IDS)	String
Start date	Timestamp
End date	Timestamp
Source IP	String
Destination IP	String
Domain	String
Email Address	String
Corporate Key	String
Technique List	List<String>

Table 7.1: Case-study Alert Attributes after Normalisation

7.3.2. Grouping

The bank performs alert grouping with a relatively simple method: Choosing one alert attribute and grouping all alerts with an equal value for this attribute. At the bank, this attribute is the Corporate Key associated with the alert. The bank defines an attack as: "*A kill-chain related to the compromise of a corporate user during a given period*". Thus, alerts are grouped per individual user, which can be identified using a so-called Corporate Key (CK). The CK is a unique string identifying an individual employed by the bank. Not all alerts have a CK attribute, or it is not trustworthy. Instead of dropping all such alerts, the bank can infer the CK for most alerts based on some other attributes' values. This is done by making some simplifying assumptions, such as that during a single day, only one individual can use a single workstation. Using this assumption, a missing CK value can be inferred using the "Source IP" attribute.

After grouping the alerts using the CK, the alerts are further split using sliding windows of length l with step size t. At the bank, t = one alert. This means that for every incoming alert, a new time window is created. For most features, the time window is the previous day from 00:00 to 23:59. However, some features have longer time windows, such as per week. The grouping step is performed in the Correlator module in Figure 7.5.

7.3.3. Sequence Classification

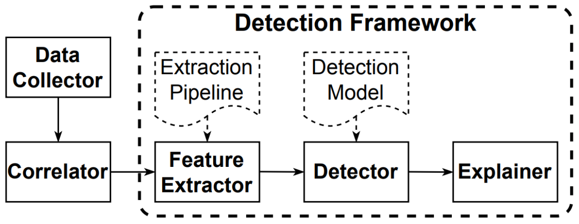


Figure 7.5: The steps used by the alert anomaly detection model currently deployed at the bank.

The bank uses a traditional supervised Machine Learning approach based on feature extraction and a gradient boosting model. The classification step consists of three substeps, feature extraction, classification, and explaining. These steps are happening in the Feature Extractor, Detector, and Explainer module in Figure 7.5, respectively.

Feature Extraction For every sequence produced by the grouping step, features are extracted. These summarising features describe an alert sequence using various numerical statistics. Every alert sequence is aggregated into one row of statistics in a table, with the columns being the names of the various statistics. Some example features are shown in Table 7.2.

Feature	Window	Value range
Number of alerts from the EDR system	1 day	Integer, [0, ck_occurrence]
Number of alerts from the SIEM system	1 day	Integer, [0, ck_occurrence]
Did "Deprecated User-Agent" IN-HOUSE model trigger	1 day	Binary, 0, 1
Sum of bytes-out in Proxy data per CK	1 day	Integer, $[0, \infty)$
Percentage uncategorised domains / of 'dst' domains	1 day	Float, [0.0, 100.0]

Table 7.2: Some examples of features used by the bank. CK: Corporate Key.

It is important to note that by aggregating the alert sequences into these feature rows, the temporal information about the alerts and their ordering is lost.

After the alert sequences are converted into features, feature selection is applied in two phases. First, variables exhibiting significant intercorrelation are eliminated. Second, Principal Component Analysis (PCA) - a strategy for decreasing the dimensionality of data - is employed to ensure that the minimal collection of features is chosen to account for the maximum extent of data fluctuation.

Classification For classification, a gradient boosting [23] algorithm is used. Specifically, CatBoost [56]. The model is a supervised learning method, meaning that all data needs to be labelled. This is challenging due to the unbalanced nature of alert anomaly detection. Attacks occur very rarely, making it difficult to obtain labelled alerts corresponding to actual attacks. At the bank, this problem is addressed by using red-team exercises as examples of alerts that correspond to actual attacks. Additionally, the bank uses Synthetic Minority Oversampling Technique (SMOTE) [11] to create additional attack samples artificially.

To fine-tune XGBoost's hyperparameters, such as the number of trees, maximum depth, and learning rate, the bank uses a grid search using a 3-fold cross-validation method over the training data. In addition, they use random sub-sampling of 80% and apply L1 regularisation.

Explaining The bank recognises the value of explainability and uses SHAP [46] to provide security analysts with local explainability. This comes in addition to the global explainability that XGBoost provides by default in the form of a feature importance view depicting relative feature importance.

However, due to the fact that the sequential nature of the data is lost in the feature extraction step, explainability can not be provided on the individual alert level.

7.4. Adapting Alerts to Logs

The performance of both traditional MLM (evaluated with LogBERT) and VF-MLM (evaluated with VoBERT) were evaluated on the case-study dataset. These techniques are explained in more detail in Chapter 4 and Chapter 5, respectively. Both methods were compared to the CatBoost baseline model developed by the bank, explained in the previous section. Since both models were not modified for the case-study, the explanation of these anomaly detection methods is not repeated here.

The models were initially designed for log anomaly detection, which means they need to be adapted for alerts. Although similar, there are some key distinctions between log and alert anomaly detection. The differences between logs and alerts are summarised in Table 7.3.

A straightforward adaptation approach is chosen: We view alerts as logs by treating the alert message as the log message, ignoring the other alert attributes besides the timestamp. Possible more sophisticated methods for this adaptation are discussed in Chapter 8.

From a purely structural point of view, the alert description attribute can be seen as a log string, ignoring all other alert attributes. When also considering what alerts and logs represent, the alert message could be seen as a log string describing suspicious behaviour.

	Alerts	Logs
Source	IDS	Processes
Meaning	Suspicious behaviour	All behaviour
Structure	Structured attributes	Free text

Table 7.3: Key differences between Logs and Alerts

7.5. Case Study Results

In order to evaluate the proposed methods on real-world alert data, a case study is conducted at a large bank with 50,000+ employees.

7.5.1. Data

The data used for evaluation was collected between 2021-06-05 and 2023-05-14. Statistics of the dataset can be found in Table 7.5.

Pre-processing The data from the bank is pre-processed using the same steps as for the public datasets. The specific settings used for grouping alerts are listed in Table 7.4. For parsing, Drain [30] is used.

Dataset	w_{size} (mins)	w_{step} (mins)	l_{min}	l_{max}
Private	60	6	10	512

Table 7.4: Pre-processing dataset window and sequence parameters. HDFS uses session window grouping. These are the same settings as used in LogBERT [26]

Dataset	Data Type	Label Granularity	Number of logs	Of which anomalous
Private	Alerts	Element	399,061	6,466 (2%)

Table 7.5: Case study dataset statistics. 'Number of logs' refers to the number of alerts before they are grouped in sequences.

7.5.2. Experiment Setup

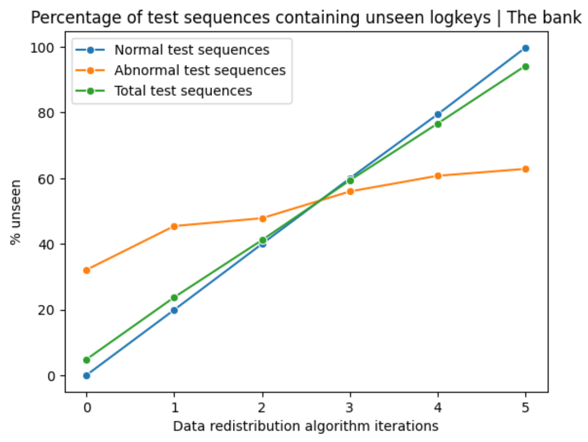
All case study experiments were run on Azure Databricks, using a Standard_NC6s_v3 GPU worker node with 128GB of VRAM.

7.5.3. Evaluation Metrics

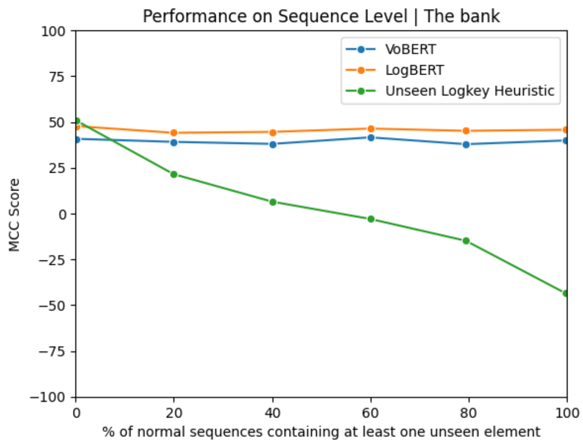
The evaluation metrics used for the case study are equal to those used for the main study, which are described in Section 6.1.2.

7.5.4. Results

The Mathew Correlation Coefficient (MCC) performance of VoBERT and LogBERT measured at different train-test sets can be seen in Figure 7.6b. These varying train-test splits are of equal size; the sequences are only reassigned between the train and test set in such a way that the number of normal sequences containing at least one unseen element increases. The dataset at $x = 0$, on which 0 data redistribution algorithm (see Section 6.3.1) iterations have been applied, is a randomly generated train-test split. All subsequent splits are generated by running the data redistribution algorithm an increasing number (x) of times.



(a) Percentage of sequences in the test set that contains at least one unseen logkey. The data redistribution algorithm is described in Section 6.3.1.



(b) VoBERT, LogBERT, and the unseen logkey heuristic predictive performance (Mathews Correlation Coefficient) for increasingly unstable data generated with the data redistribution algorithm (Section 6.3.1).

Figure 7.6: Performance of VF-MLM, MLM, and a heuristic for different train test splits are shown in Figure 7.6b. Statistics of the train-test splits are shown in figure 7.6a.

During the experiments on public datasets, it was found that some train-test splits are very easy to classify correctly: Just using a simple heuristic is enough. The simple heuristic that was able to classify the train-test splits of the public datasets with a high MCC score was the "unseen logkey heuristic". This heuristic is based on unseen logkeys. The anomaly scores it assigns to each sequence in the test set is the ratio of logkeys in the sequence that did not occur in the train set. In other words, the anomaly score of each sequence is the ratio of unseen logkeys when using the unseen logkey heuristic.

The main results of the experiment are shown in two plots in Figure 7.6. The first plot (Figure 7.6a)

Dataset	Sequences (total)	Sequences (anomalous)	Mean sequence length	Vocabulary size (parsed)
Private	20,992	718	148	457

Table 7.6: Statistics describing the pre-processed case-study dataset. The data was pre-processed using the specified pre-processing pipeline in Table 6.1

describes the train/test splits used for evaluating the methods. For each number of the dataset re-distribution algorithm (Algorithm 4), the percentage of sequences containing unseen logkeys is shown. This percentage is shown for both the normal sequences (normal) as well as the anomalous sequences (abnormal) in the test set, and when looking at the test set as a whole (total).

Note that the percentage of sequences containing unseen logkeys in the entirety of the test set (total) is not the sum of such sequences in the normal and abnormal part of the test set, since the number of normal and abnormal sequences in the test set is not equal. The number of normal sequences is much higher: 20,000 normal sequences vs 718 anomalous sequences in the test set.

While the anomalous sequences in the test set contain relatively more unseen logkeys than the normal part of the test set for the initial train-test split, this difference is not as significant as for the public datasets. After three iterations of the dataset redistribution algorithm, this balance shifts and the normal sequences in the test set contain a larger percentage of sequences with unseen logkeys. Additional statistics describing the train-test splits can be seen in appendix A.

The second plot (Figure 7.6a) shows the MCC scores of both deep anomaly detection methods as well as the unseen logkey heuristic obtained at the train/test splits described in the first plot. The train/test splits are characterised by increasing data instability. As in the main study, the log data instability is measured by the percentage of unseen logkeys in the regular part of the test set. This is the statistic shown on the x-axis of Figure 7.6a.

The MCC scores of all evaluated anomaly detection methods are quite low. They are, at most, around 50. In contrast to the evaluation of these methods on the public datasets, the MCC scores of both VF-MLM and MLM remain stable across the increasing data instability. The unseen logkey heuristic works just as well as the deep anomaly detection methods on the initial split but decreases to a negative correlation with the labels when data instability increases (MCC < 0). In contrast to the public datasets, the unseen logkey heuristic never greatly outperforms MLM or VF-MLM.

			Sequence-level		
Dataset	Method	Instability	F1	MCC	AUPRC
Private	LogBERT	Min	45.47	47.73	0.50
		Max	45.25	45.74	0.45
		Mean	44.62	45.61	0.47
	VoBERT	Min	39.50	40.79	0.41
		Max	39.63	39.89	0.43
		Mean	38.62	39.14	0.40
	Heuristic	Min	45.21	50.82	0.70
		Max	17.58	-43.51	0.40
		Mean	27.74	2.91	0.47

Table 7.7: Case study results. Min= Minimal data instability, obtained by randomly shuffling the train-test (80%-20%) split. Max= Maximum data instability, created with five iterations of the data redistribution algorithm (Section 6.3.1).

7.5.5. Discussion

From the results, it can be seen that the MCC score remains stable across the increasing normal sequences containing at least one unseen element. This contrasts with the public datasets that were evaluated; on those datasets, the MCC score decreased significantly when the data instability increased. This points to the fact that the unseen logkey heuristic (classifying sequences with unseen logkeys as anomalous) is not effective on the dataset used by the bank, even when the normal sequences in the test set contain no unseen elements at all. For the unseen logkey heuristic to be ineffective when there are no unseen logkeys in any of the normal log sequences, there also needs to be a lack of unseen logkeys in the anomalous sequences. This is indeed the case, as can be seen in figure 7.6a.

By analysing a real-life dataset, this case study highlights that the positive picture painted in some of the papers in this field is sometimes too optimistic. It is paramount to evaluate the complexity of the evaluation datasets, and the specific train-test splits used. For the most frequently used public log datasets, a simple heuristic can perform well under specific circumstances. LogBERT [26] was evaluated under these circumstances, which has led to a reported performance evaluation that is not representative of the real performance.

We show that both this simple heuristic, as well as LogBERT and VoBERT, did not perform as well on real-world data as on the publicly available datasets. The average MCC score of LogBERT evaluated on the public datasets dropped from 75% to 48% when compared to the real-world data. This is a troubling finding since most current literature evaluates their approaches to these three public datasets, which under some circumstances might not be challenging enough to evaluate complex anomaly detection methods.

8 Limitations and Future Work

In this chapter, the study's limitations and promising research directions for future work are discussed per research question.

8.1. RQ1: Element-level Evaluation

Hybrid masking methods In this work, ratio and per-element masking are compared. Per-element masking is computationally very expensive. However, a mix between ratio and per-element masking could also be evaluated. One such method is n-gram making, in which multiple predictions are made for each sequence, just like per-element masking. However, instead of masking only one element per prediction, n consecutive elements are masked. Compared to per-element masking, this lowers the number of predictions required for a sequence of length l from l to $\lceil l/n \rceil$. n-gram masking is visualised in Figure 8.1.

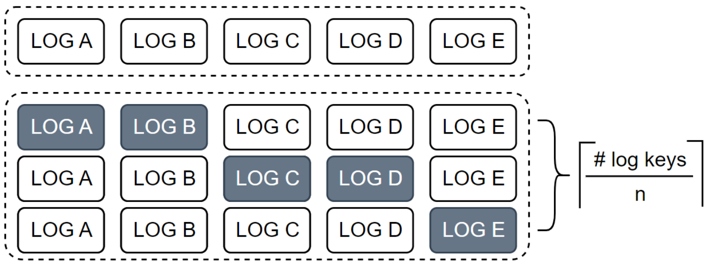


Figure 8.1: n-Gram masking.

8.2. RQ2: Unstable Logkeys

Logkey Evolution The evaluation done in this work evaluates LogBERT (based on MLM) and VoBERT (based on VF-MLM) across a varying degree of data instability, measured as the percentage of normal sequences from the test set containing at least one logkey that was not seen in the training set. This is a general measure of data instability, but it does not directly relate to a specific kind of data instability, such as log key evolution (slight changes in logkey formulation but retaining similar meaning). The performance was not explicitly evaluated for this type of data instability. Still, it would be interesting to do so since the main strength of VF-MLM is the ability to classify normal sequences containing unseen logkeys correctly. Having logkeys that are slightly changed would be a helping factor for this task. Therefore, it is interesting to evaluate this type of log data instability specifically in future work.

Limited hyperparameter Tuning As part of the sensitivity analysis, multiple loss functions were evaluated. However, due to the time constraints of this project and the long time it takes to train and evaluate a BERT model from scratch, all loss functions are evaluated using the same hyperparameters. This probably has a negative influence on some of the loss functions since the hyper-parameters are not optimised for those loss functions. Specifically, the performance using cosine similarity was lower than expected since this metric is often used to compare semantic vectors. Sub-optimal hyperparameters for this loss function could cause this. It might thus be the case that e.g., cosine similarity could achieve the highest predictive performance if the hyper-parameters are sufficiently tuned. Hence, we suggest future researchers to do more extensive hyperparameter tuning on VoBERT when using cosine similarity as a loss function.

Alternative methods for the Semantic Embedding Layer Only one semantic embedding method for the semantic embedding layer of VoBERT is evaluated: Sentence Bert. Sentence Bert is pre-trained on natural language and is not fine-tuned on log data. Using an encoder specifically suited for log language to embed the logs instead of sentence-bert will probably improve predictive performance since the quality of the semantic embeddings will be improved.

Apart from finetuning the selected model on log data, other models could also be tried. Promising candidates include the combination of FastText [34] and TF-IDF [2], as well as SimCSE [24]. SimCSE is a state-of-the-art sentence BERT-based contrastive learning framework for producing sentence embeddings. Evaluating VoBERT performance using these alternative models as the semantic embedding layer is a promising future direction.

Different number of BERT layers With traditional MLM, the initial embeddings are randomly generated before training starts. However, the embeddings are updated along with the rest of the model during backpropagation in the training phase. Since for MLM, the embeddings are also trained along with the model but remain static in VF-MLM, the latter method might need an extra BERT layer to compensate for this. Looking at the effect of adding more layers to the BERT model could be an interesting direction for future research. Since extra trainable parameters are added, special care should be taken to avoid overfitting.

Different instability proxies The percentage of sequences containing at least one unseen logkey might not be the best proxy for log data instability. Other proxies could also be imagined, such as the percentage of all raw logkeys in the test set that is unseen (while counting duplicate logkeys). It could be that the effect might be better observed using one of these other proxies for log data instability. Evaluating the performance using different instability proxies is a promising future direction.

8.3. RQ3: Security Events Case Study

When dealing with alerts, the “alert message” is seen as a log key, and the timestamp is used as is, ignoring all other attributes. This leads to the loss of information stored in the other attributes, such as source IP, the IDS system that generated the alert, the timestamp, etc. Hence, it could be interesting to see the effect of incorporating these additional attributes into the semantic embeddings used by VF-MLM. Three distinct methods for accomplishing this are proposed:

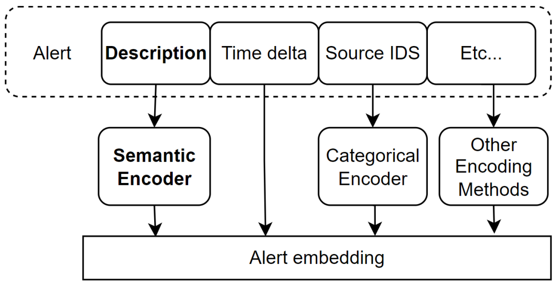


Figure 8.2: Separately encoding string and numerical features. The bold feature and embedding method are what was used in this research.

Embed string and numerical features separately The above-mentioned semantic embedding method could be used to create an embedding for the text fields of an alert, such as the description. Other attributes could be embedded in a way fitting for the attribute type (categorical/numerical). For instance, the alert source field is a categorical variable that denotes what IDS created the alert. The possible values are known beforehand and are often small (i.e. ≤ 10). This could be embedded using one-hot encoding. Numerical attributes do not need conversion and could just be taken as is. Finally, all embeddings could be combined by e.g. concatenation, summation, or calculating the mean. This way of embedding the alerts is shown in Figure 8.2. This could be implemented using the LogAI framework [15] developed by SalesForce. This framework provides different embedding methods out of the box and is publicly available on GitHub [1].

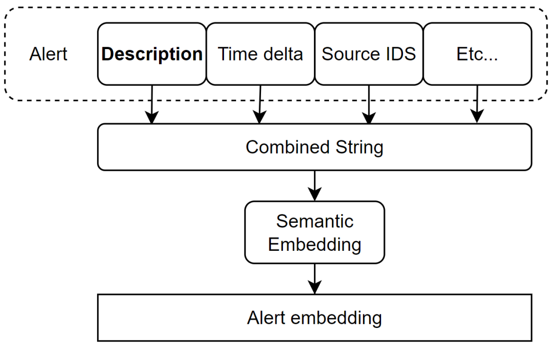


Figure 8.3: Creating a combined string representation of an alert, and then encoding that string.

Embed string and numerical features together An alternative option could be to convert the entire alert and all its attributes into a text string representation. This string could then be embedded using standard NLP sentence embedding methods. It could, for example, be encoded using BERT. This method of embedding an alert is shown in Figure 8.3.

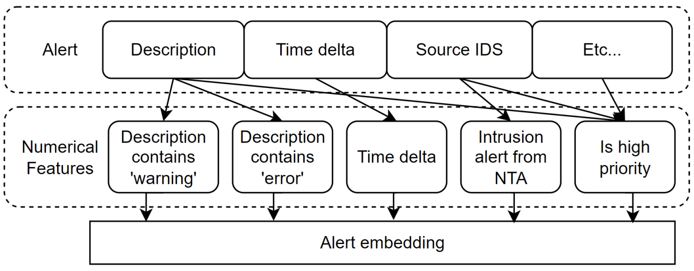


Figure 8.4: Extracting numerical features from the alerts and then using those features directly as the embedding.

Feature based embeddings Finally, an option could be to convert alerts into a vector consisting of numerical features. For this approach, a feature selection step is needed, that extracts numerical features from an alert. The features can be combined into one vector describing the alert, which can then be used as the embedding. This way of embedding the alerts is shown in Figure 8.4.

[1]`https://github.com/salesforce/logai`

9 Conclusion

With the ever-increasing digitalisation of society and the explosion of internet-enabled devices with IoT, keeping services and devices secure is becoming ever more important. Logs play a critical role in sustaining system reliability. Manual analysis of logs has become increasingly difficult, accelerating the development of automated methods for log anomaly detection. Despite significant progress in automating log analysis, current state-of-the-art methods face challenges dealing with unstable log data, which means that the content of log messages evolves over time. A study examining the stability of logging statements found that around 20% to 45% of logging statements changed throughout their lifetime [35]. Typically, minor modifications to an existing log key can yield a new but semantically similar logkey. In real-world settings, models can often not be retrained in time to accommodate these changes [75]. Current approaches interpret this as an entirely new log key [21, 31, 43, 71, 26], causing the effectiveness of most of the current state-of-the-art methods to be significantly limited by the instability of log data [75].

To address this issue, we answered the research question: "How can anomalies in unstable sequential log data be detected?". We highlighted the question's relevance by showing the inability of a state-of-the-art Bidirectional Encoder Representations from Transformers (BERT) based model called LogBERT [26] to deal with unstable log data. It was also shown that the high performance reported in the LogBERT paper was not representative of real-world performance but instead relied on a simple heuristic that only works under specific circumstances.

To answer the research question, we proposed Vocabulary-Free BERT (VoBERT): A BERT-based model that uses a novel pre-training task that we named Vocabulary-Free Masked Language Modeling (VF-MLM). We have developed VF-MLM by adapting traditional Masked Language Modelling (MLM) and removing the fixed vocabulary constraint. This is enabled by the vital insight that a fixed vocabulary, while needed for traditional NLP tasks, is not required when using BERT for anomaly detection tasks. To drop the fixed-vocabulary constraint, we removed the final layer of a traditional MLM-based model and leveraged a second sentence encoder model to generate semantic initial embeddings. Doing so increases VoBERT's robustness by enabling it to classify out-of-vocabulary logkeys correctly.

We showed that VoBERT is more stable across varying levels of log data instability. On the frequently used Thunderbird log dataset, increasing data instability to 97% unseen logkeys caused the MCC score of LogBERT to drop from 64% to 14%, while the MCC score of VF-MLM only dropped from 39% to 37%. For the datasets that allowed it, we evaluated on element-level, providing a more granular assessment of its performance.

To assess the generalisation of the experimental results to real-world scenarios, a case study was conducted evaluating the anomaly detection models on real-world security event data collected at a large bank (50,000+ employees). It was found that the simple heuristic LogBERT used did not work for this real-world data. VoBERT showed performance on par with LogBERT on this real-world security event dataset.

This work provides important insights into the challenges of unstable log data and is a proof of concept for the novel pre-training task VF-MLM. Therefore, it is not only a significant contribution to our understanding of handling unstable log data but also a stepping stone towards more innovative, effective solutions in the domain of anomaly detection.

References

[1] Chief Information Security Officer (CISO). *Anticipating the Unknowns - Benchmark Study.* 2019. URL: https://ebooks.cisco.com/story/anticipating-unknowns/ (visited on 01/31/2023).

[2] Akiko Aizawa. "An information-theoretic perspective of tf–idf measures". In: *Information Processing & Management* 39.1 (2003), pp. 45–65.

[3] Bushra A Alahmadi, Louise Axon, and Ivan Martinovic. "99% False Positives: A Qualitative Study of SOC Analysts' Perspectives on Security Alarms". In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security), Boston, MA, USA.* 2022, pp. 10–12.

[4] Jay Alammar. *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning).* 2021. URL: https://jalammar.github.io/illustrated-bert/ (visited on 05/12/2023).

[5] Rebecca Gurley Bace, Peter Mell, et al. "Intrusion detection systems". In: (2001).

[6] Peter Bodik et al. "Fingerprinting the datacenter: automated classification of performance crises". In: *Proceedings of the 5th European conference on Computer systems.* 2010, pp. 111–124.

[7] Jakub Breier and Jana Branišová. "Anomaly detection from log files using data mining techniques". In: *Information Science and Applications.* Springer. 2015, pp. 449–457.

[8] Marta Catillo, Antonio Pecchia, and Umberto Villano. "AutoLog: Anomaly detection by deep autoencoding of system logs". In: *Expert Systems with Applications* 191 (2022), p. 116263.

[9] Raghavendra Chalapathy and Sanjay Chawla. "Deep learning for anomaly detection: A survey". In: *arXiv preprint arXiv:1901.03407* (2019).

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly detection for discrete sequences: A survey". In: *IEEE transactions on knowledge and data engineering* 24.5 (2010), pp. 823–839.

[11] Nitesh V Chawla et al. "SMOTE: synthetic minority over-sampling technique". In: *Journal of artificial intelligence research* 16 (2002), pp. 321–357.

[12] Mike Chen et al. "Failure diagnosis using decision trees". In: *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE. 2004, pp. 36–43.

[13] Song Chen and Hai Liao. "BERT-Log: Anomaly Detection for System Logs Based on Pre-trained Language Model". In: *Applied Artificial Intelligence* 36.1 (2022), p. 2145642. DOI: 10.1080/08839514.2022.2145642. eprint: https://doi.org/10.1080/08839514.2022.2145642. URL: https://doi.org/10.1080/08839514.2022.2145642.

[14] Zhuangbin Chen et al. "Experience report: Deep learning-based system log analysis for anomaly detection". In: *arXiv preprint arXiv:2107.05908* (2021).

[15] Qian Cheng et al. *LogAI: A Library for Log Analytics and Intelligence.* 2023. DOI: 10.48550/ARXIV.2301.13415. URL: https://arxiv.org/abs/2301.13415.

[16] Maria Cvach. "Monitor alarm fatigue: an integrative review". In: *Biomedical instrumentation & technology* 46.4 (2012), pp. 268–277.

[17] Herve Debar, David Curry, and Benjamin Feinstein. *The intrusion detection message exchange format (IDMEF).* Tech. rep. 2007.

[18] Demisto. *The State of SOAR.* 2018. URL: https://start.paloaltonetworks.com/the-state-of-soar-report-2018 (visited on 01/31/2023).

[19] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[20] Min Du and Feifei Li. "Spell: Streaming parsing of system event logs". In: *2016 IEEE 16th International Conference on Data Mining (ICDM).* IEEE. 2016, pp. 859–864.

[21] Min Du et al. "Deeplog: Anomaly detection and diagnosis from system logs through deep learning". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 1285–1298.

[22] S. Forrest et al. "A sense of self for Unix processes". In: *Proceedings 1996 IEEE Symposium on Security and Privacy*. 1996, pp. 120–128. DOI: 10.1109/SECPRI.1996.502675.

[23] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[24] Tianyu Gao, Xingcheng Yao, and Danqi Chen. "Simcse: Simple contrastive learning of sentence embeddings". In: *arXiv preprint arXiv:2104.08821* (2021).

[25] Ibrahim Ghafir et al. "Hidden Markov models and alert correlations for the prediction of advanced persistent threats". In: *IEEE Access* 7 (2019), pp. 99508–99520.

[26] Haixuan Guo, Shuhan Yuan, and Xintao Wu. "Logbert: Log anomaly detection via bert". In: *2021 international joint conference on neural networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[27] Yalan Guo et al. "Anomaly detection using distributed log data: A lightweight federated learning approach". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2021, pp. 1–8.

[28] Wajih Ul Hassan et al. "Nodoze: Combatting threat alert fatigue with automated provenance triage". In: *network and distributed systems security symposium*. 2019.

[29] Pinjia He et al. "An evaluation study on log parsing and its use in log mining". In: *2016 46th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE. 2016, pp. 654–661.

[30] Pinjia He et al. "Drain: An online log parsing approach with fixed depth tree". In: *2017 IEEE international conference on web services (ICWS)*. IEEE. 2017, pp. 33–40.

[31] Pinjia He et al. "Towards automated log parsing for large-scale log data analysis". In: *IEEE Transactions on Dependable and Secure Computing* 15.6 (2017), pp. 931–944.

[32] Shilin He et al. "Experience report: System log analysis for anomaly detection". In: *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)*. IEEE. 2016, pp. 207–218.

[33] Zhiheng Huang, Wei Xu, and Kai Yu. "Bidirectional LSTM-CRF models for sequence tagging". In: *arXiv preprint arXiv:1508.01991* (2015).

[34] Armand Joulin et al. "Fasttext. zip: Compressing text classification models". In: *arXiv preprint arXiv:1612.03651* (2016).

[35] Suhas Kabinna et al. "Examining the stability of logging statements". In: *Empirical Software Engineering* 23 (2018), pp. 290–333.

[36] Igor Kotenko, Diana Gaifulina, and Igor Zelichenok. "Systematic Literature Review of Security Event Correlation Methods". In: *IEEE Access* (2022).

[37] Max Landauer et al. "Deep Learning for Anomaly Detection in Log Data: A Survey". In: *arXiv preprint arXiv:2207.03820* (2022).

[38] Max Landauer et al. "System log clustering approaches for cyber security applications: A survey". In: *Computers & Security* 92 (2020), p. 101739.

[39] Van-Hoang Le and Hongyu Zhang. "Log-based anomaly detection with deep learning: How far are we?" In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 1356–1367.

[40] Van-Hoang Le and Hongyu Zhang. "Log-based anomaly detection without log parsing". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 492–504.

[41] Diana. Levshun and Igor Kotenko. "A survey on artificial intelligence techniques for security event correlation: models, challenges, and opportunities". In: *Artificial Intelligence Review* (2023).

[42] Kun-Lun Li et al. “Improving one-class SVM for anomaly detection”. In: *Proceedings of the 2003 international conference on machine learning and cybernetics (IEEE Cat. No. 03EX693)*. Vol. 5. IEEE. 2003, pp. 3077–3081.

[43] Yinglung Liang et al. “Failure prediction in ibm bluegene/l event logs”. In: *Seventh IEEE International Conference on Data Mining (ICDM 2007)*. IEEE. 2007, pp. 583–588.

[44] Richard P Lippmann et al. “Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation”. In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX’00*. Vol. 2. IEEE. 2000, pp. 12–26.

[45] Jian-Guang Lou et al. “Mining Invariants from Console Logs for System Problem Detection.” In: *USENIX annual technical conference*. 2010, pp. 1–14.

[46] Scott M Lundberg and Su-In Lee. “A unified approach to interpreting model predictions”. In: *Advances in neural information processing systems* 30 (2017).

[47] Adetokunbo AO Makanju, A Nur Zincir-Heywood, and Evangelos E Milios. “Clustering event logs using iterative partitioning”. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2009, pp. 1255–1264.

[48] Brian W Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975), pp. 442–451.

[49] Salma Messaoudi et al. “A search-based approach for accurate identification of log message formats”. In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 167–177.

[50] H Sardana Milan and Kamalpreet Singh. “Reducing false alarms in intrusion detection systems–a survey”. In: *International Research Journal of Engineering and Technology (IRJET) e-ISSN* 2395 (2018), p. 0056.

[51] Julio Navarro, Aline Deruyver, and Pierre Parrend. “A systematic survey on multi-step attack detection”. In: *Computers & Security* 76 (2018), pp. 214–249.

[52] Kim Anh Nguyen, Sabine Schulte im Walde, and Ngoc Thang Vu. “Integrating distributional lexical contrast into word embeddings for antonym-synonym distinction”. In: *arXiv preprint arXiv:1605.07766* (2016).

[53] 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)*. IEEE. 2007, pp. 575–584.

[54] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[55] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[56] Liudmila Prokhorenkova et al. “CatBoost: unbiased boosting with categorical features”. In: *Advances in neural information processing systems* 31 (2018).

[57] Jiaxing Qi et al. “LogEncoder: Log-based Contrastive Representation Learning for anomaly detection”. In: *IEEE Transactions on Network and Service Management* (2023).

[58] Ali Ahmadian Ramaki, Abbas Rasoolzadegan, and Abbas Ghaemi Bafghi. “A systematic mapping study on intrusion alert analysis in intrusion detection systems”. In: *ACM Computing Surveys (CSUR)* 51.3 (2018), pp. 1–41.

[59] Nils Reimers and Iryna Gurevych. “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Nov. 2019. URL: `http://arxiv.org/abs/1908.10084`.

[60] James J Rooney and Lee N Vanden Heuvel. “Root cause analysis for beginners”. In: *Quality progress* 37.7 (2004), pp. 45–56.

[61] Daniel Rosenberg. *Unbalanced Data? Stop Using ROC-AUC and Use AUPRC Instead.* 2022. URL: `https://towardsdatascience.com/imbalanced-data-stop-using-roc-auc-and-use-auprc-instead-46af4910a494` (visited on 06/05/2023).

[62] Lukas Ruff et al. “Deep one-class classification”. In: *International conference on machine learning.* PMLR. 2018, pp. 4393–4402.

[63] Piotr Ryciak, Katarzyna Wasielewska, and Artur Janicki. “Anomaly detection in log files using selected natural language processing methods”. In: *Applied Sciences* 12.10 (2022), p. 5089.

[64] Alireza Sadighian et al. “Ontids: A highly flexible context-aware and ontology-based alert correlation framework”. In: *International Symposium on Foundations and Practice of Security.* Springer. 2014, pp. 161–177.

[65] Xinying Song et al. “Fast wordpiece tokenization”. In: *arXiv preprint arXiv:2012.15524* (2020).

[66] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[67] Jin Wang et al. “LogEvent2vec: LogEvent-to-vector based anomaly detection for large-scale logs in internet of things”. In: *Sensors* 20.9 (2020), p. 2451.

[68] Wei Wang et al. “Robust Unsupervised Network Intrusion Detection with Self-supervised Masked Context Reconstruction”. In: *Computers & Security* (2023), p. 103131.

[69] Xiaoyu Wang et al. “MAAC: Novel Alert Correlation Method To Detect Multi-step Attack”. In: *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom).* IEEE. 2021, pp. 726–733.

[70] Zhiwei Wang et al. “Multi-scale one-class recurrent neural networks for discrete event sequence anomaly detection”. In: *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining.* 2021, pp. 3726–3734.

[71] Wei Xu et al. “Detecting large-scale system problems by mining console logs”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* 2009, pp. 117–132.

[72] Wei Xu et al. “Largescale system problem detection by mining console logs”. In: *Proceedings of SOSP.* Vol. 9. Citeseer. 2009, pp. 1–17.

[73] 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).* IEEE. 2020, pp. 1215–1220.

[74] Bo Zhang et al. “Anomaly detection via mining numerical workflow relations from logs”. In: *2020 International Symposium on Reliable Distributed Systems (SRDS).* IEEE. 2020, pp. 195–204.

[75] Xu Zhang et al. “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.* 2019, pp. 807–817.

[76] Jieming Zhu et al. “Learning to log: Helping developers make informed logging decisions”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* Vol. 1. IEEE. 2015, pp. 415–425.

[77] Jieming Zhu et al. “Tools and benchmarks for automated log parsing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP).* IEEE. 2019, pp. 121–130.

A

Dataset Statistics

A.1. Overview

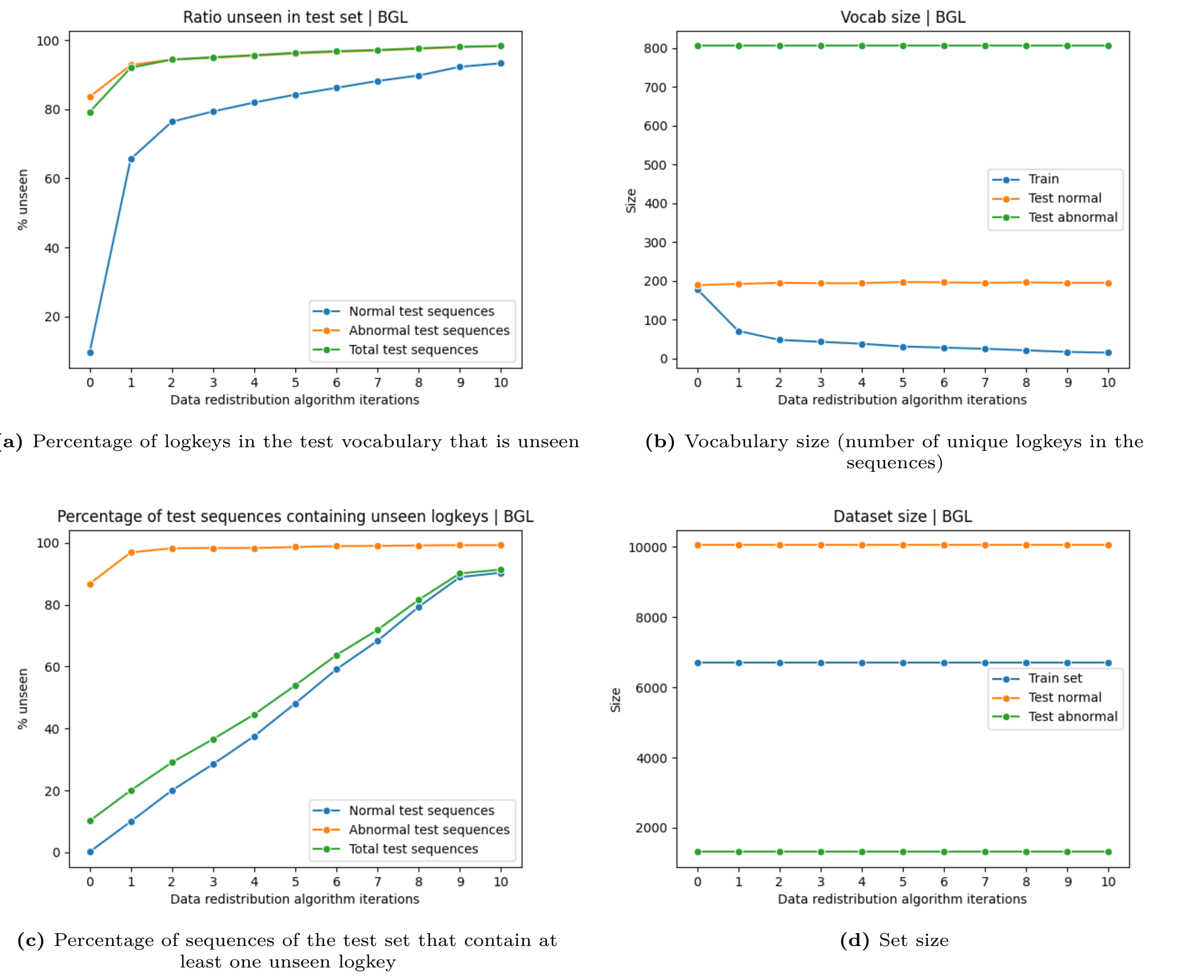
Dataset	Instability	Part	% Vocab Unseen	% Seqs Containing Unseen	Vocab Size	Size
BGL	Min	Train	-	-	179	6697
		Test Normal	9.52	0.13	189	10,047
		Test Anomalous	83.60	86.75	805	1,313
		Test Total	79.21	10.14		11,360
	Max	Train	-	-	15	6,697
		Test Normal	93.33	90.34	195	10,047
		Test Anomalous	98.26	99.24	805	1,313
		Test Total	98.36	91.36		11,360
TBird	Min	Train	-	-	843	6,000
		Test Normal	20.38	0.88	1,055	70,208
		Test Anomalous	19.75	99.16	886	23,112
		Test Total	27.84	25.22		93,320
	Max	Train	-	-	13	6,000
		Test Normal	98.77	96.75	1,058	70,208
		Test Anomalous	98.87	100.00	886	23,112
		Test Total	98.89	97.55		93,320
HDFS	Min	Train	-	-	15	4,855
		Test Normal	21.05	0.01	19	553,368
		Test Anomalous	67.39	36.33	46	8,419
		Test Total	67.39	0.55		561,787
	Max	Train	-	-	5	4,855
		Test Normal	73.68	83.47	19	553,368
		Test Anomalous	89.13	82.63	46	8,419
		Test Total	89.13	83.46		561,787

Table A.1: Statistics describing the used public datasets with the train and test sizes equal to the sizes in LogBERT [26]. Instability: Min represents the original LogBERT paper [26] train/test split. Modified: Max represents the most extremely modified train/test split obtained by applying the data redistribution algorithm (see Section 6.3.1).

Dataset	Instability	Part	% Vocab Unseen	% Seqs Containing Unseen	Vocab Size	Size
Private	Min	Train	-	-	438	16,219
		Test Normal	0.24	0.02	413	4,055
		Test Anomalous	5.51	29.25	272	718
		Test Total	3.65	4.42		4,773
	Max	Train	-	-	211	16,219
		Test Normal	52.05	98.52	438	4,055
		Test Anomalous	31.25	62.67	272	718
		Test Total	53.64	93.13		4,773

Table A.2: Statistics describing the used case study dataset. Instability: Min represents a randomly initialised train-test split of size 80%-20% respectively. Modified: Max represents the most extremely modified train/test split obtained by applying the data redistribution algorithm (see Section 6.3.1).

A.2. BGL train-test splits



(a) Percentage of logkeys in the test vocabulary that is unseen

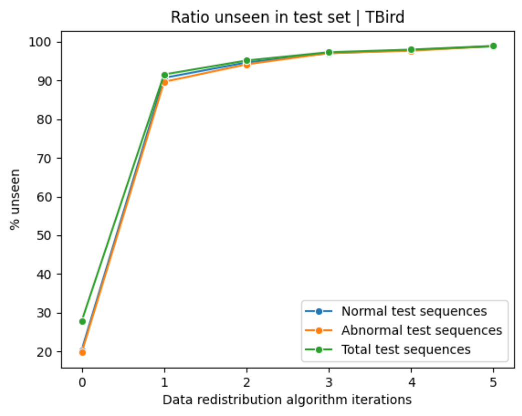
(b) Vocabulary size (number of unique logkeys in the sequences)

(c) Percentage of sequences of the test set that contain at least one unseen logkey

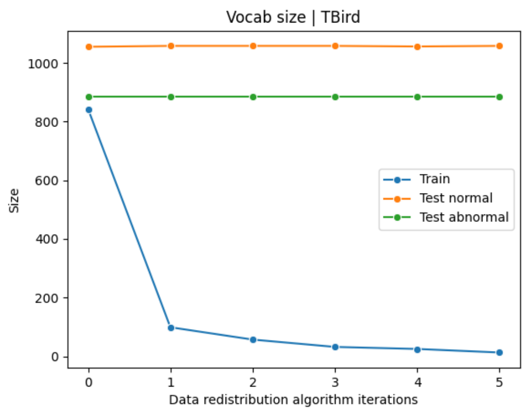
(d) Set size

Figure A.1: Statistics for varying the unseen ratio for the BGL dataset, generated with the data redistribution algorithm (Section 6.3.1). Train and test size are kept fixed, as well as the ratio of anomalous sequences in the test set. Every $x = i$ represents i consecutive iterations of the data redistribution algorithm.

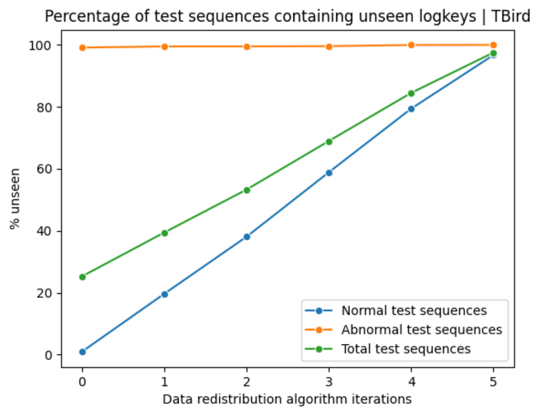
A.3. TBird train-test splits



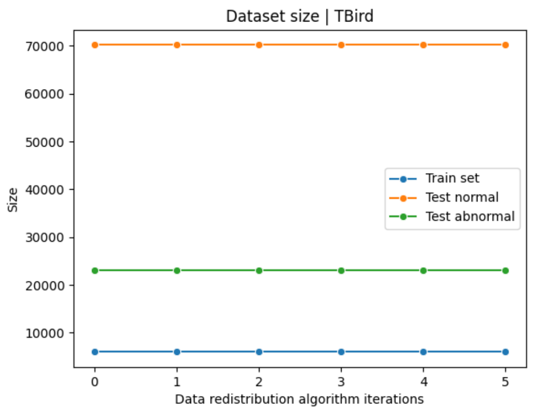
(a) Percentage of logkeys in the test vocabulary that is unseen



(b) Vocabulary size (number of unique logkeys in the sequences)



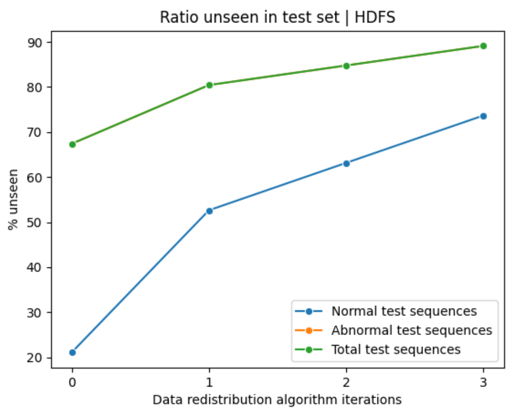
(c) Percentage of sequences of the test set that contain at least one unseen logkey



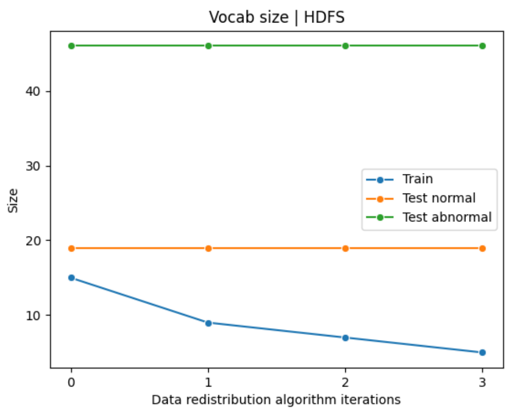
(d) Set size

Figure A.2: Statistics for varying the unseen ratio for the TBird dataset, generated with the data redistribution algorithm (Section 6.3.1). Train and test size are kept fixed, as well as the ratio of anomalous sequences in the test set. Every $x = i$ represents i consecutive iterations of the data redistribution algorithm.

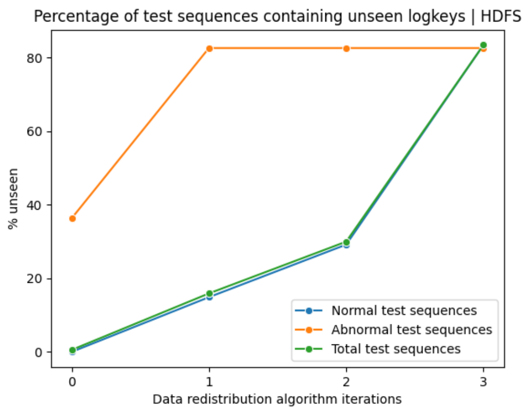
A.4. HDFS train-test splits



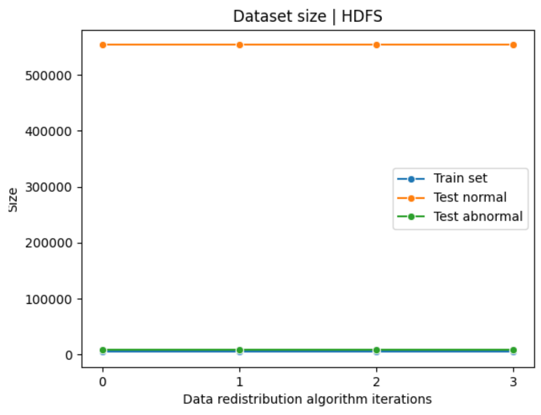
(a) Percentage of logkeys in the test vocabulary that is unseen



(b) Vocabulary size (number of unique logkeys in the sequences)



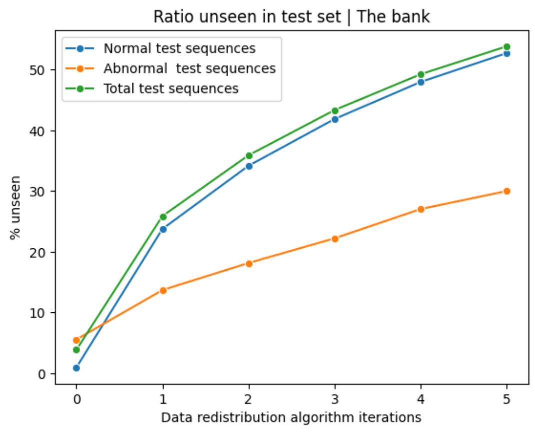
(c) Percentage of sequences of the test set that contain at least one unseen logkey



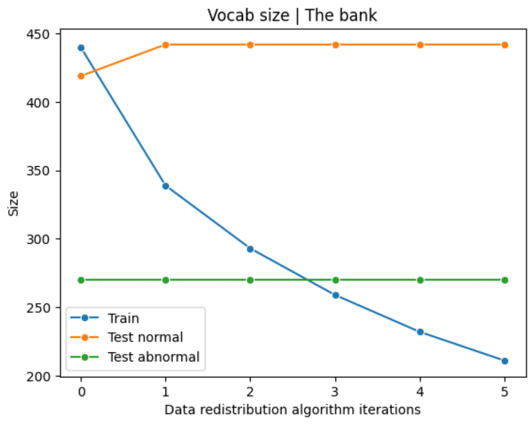
(d) Set size

Figure A.3: Statistics for varying the unseen ratio for the HDFS dataset, generated with the data redistribution algorithm (Section 6.3.1). Train and test size are kept fixed, as well as the ratio of anomalous sequences in the test set. Every $x = i$ represents i consecutive iterations of the data redistribution algorithm.

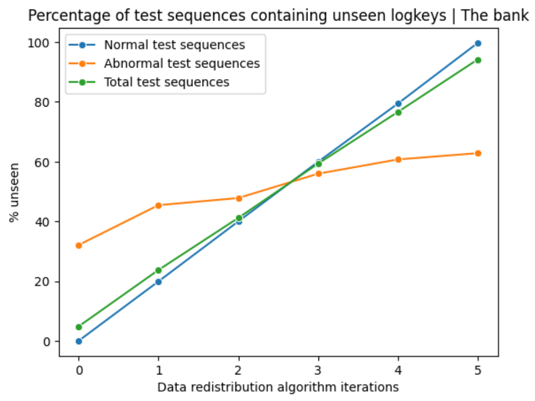
A.5. Case Study train-test splits



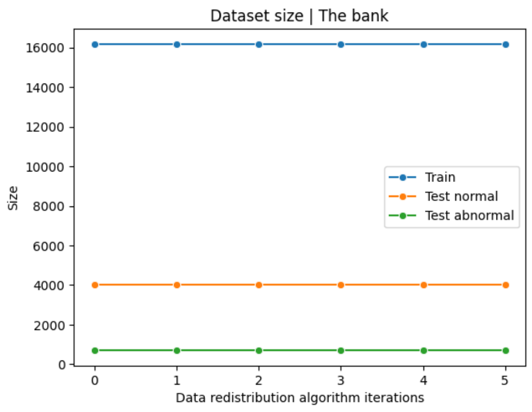
(a) Percentage of logkeys in the test vocabulary that is unseen



(b) Vocabulary size (number of unique logkeys in the sequences)



(c) Percentage of sequences of the test set that contain at least one unseen logkey



(d) Set size

Figure A.4: Statistics for varying the unseen ratio for the case study dataset, generated with the data redistribution algorithm (Section 6.3.1). Train and test size are kept fixed, as well as the ratio of anomalous sequences in the test set. Every $x = i$ represents i consecutive iterations of the data redistribution algorithm.

B Alert Details

Attribute	Type	Example
Alert id	String	abc1
Alert description	String	Suspicious connection blocked by network protection
Alert source (which IDS)	String	IDS_1
Start date	Timestamp	12-02-2023
End date	Timestamp	12-02-2023
Source IP	String	10.0.1.1
Destination IP	String	10.0.1.5
Email Address	String	John.Doe@company.com
Corporate Key	String	1234ABC
Domain	String	company.com
Technique List	String[]	[Technique1, Technique2]

Table B.1: Alert Attributes as present in the case study, after normalisation.