A Lightweight Anomaly Detection Approach in Large Logs Using Generalizable Automata

Ameneh Kazemimiraki

A Thesis

in the Department of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science in Electrical and Computer Engineering

at

Concordia University

Montreal, Quebec, Canada

March 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Ameneh Kazemimiraki

Entitled: A Lightweight Anomaly Detection Approach in Large Logs Using Generalizable Automata

submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Dongyu Qiu________________________ Chair

Dr. Olga Ormandjieva________________________Examiner

Dr. Dongyu Qiu________________________Examiner

Dr. Abdelwahab Hamou-Lhadj________________________Supervisor

Dr. Otmane Ait-Mohamed ________________________Supervisor

Approved by: ________________________________

Dr. Jun Cai________________ 2022________________________

Dr. Mourad Debbabi, Dean, Gina Cody School of Engineering and Computer Science

Abstract

A Lightweight Anomaly Detection Approach in Large Logs

Using Generalizable Automata

Ameneh Kazemimiraki

In this thesis, we focus on the problem of detecting anomalies in large log data. Logs are generated at runtime and contain a wealth of information, useful for various software engineering tasks, including debugging, performance analysis, and fault diagnosis. Our anomaly detection approach is based on the multiresolution abnormal trace detection algorithm proposed in the literature. The algorithm exploits the causal relationship of events in large execution traces to build a model that represents the normal behaviour of a system using varying length n-grams and a generalizable automaton. The resulting model is later used to detect deviations from normalcy.

In this thesis, we investigate the application of this algorithm in detecting anomalies in log data. Logs and execution traces are different. Unlike traces, logs do not exhibit a causal relationship among their events, raising questions as to the effectiveness of automata to model log data for anomaly detection. Logs are unstructured data and hence require the use of parsing and abstraction techniques.

We propose a process, called LogAutomata, which uses the multiresolution abnormal trace detection algorithm as its primary mechanism. When applying LogAutomata to a large log file generated from the execution of Hadoop Distributed File System (HDFS), we show that the multiresolution algorithm can be a very effective way to detect anomalies in log data.

Acknowledgments

I would like to convey my heartfelt gratitude to Dr. Wahab Hamou-Lhadj, and Dr. Otmane Ait-Mohamed who provided me with the incredible opportunity to conduct this research. I appreciate their faith in me, their constant support, keeping me motivated, and the fact that they keep their doors open for constructive feedback and discussion.

I would also like to thank everyone in Professor's Lab for their friendship, inspiration, and thought-provoking conversations. I would also like to express my gratitude to Concordia University for their support and assistance, particularly the Concordia University Library, which provided a wealth of literature and guidance on report writing.

I would also like to express my gratitude to my parents, siblings and friends for their encouragement and loving support during my achievements, and setbacks. I would like to express my gratitude to my husband as well for his always support during my ups and down of life. Without them, none of this would be possible.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Motivation

In the last decades, software systems have become more data-driven and complex. Cloud computing and parallel processing frameworks such as Hadoop and Spark are widely used in many domains [1] [2]. With the increase in the scale and complexity of the systems, intrusion detection is an essential component for detecting performance degradation, crashes, security attacks, and other undesirable behaviours [3]. Not addressing these issues may lead to revenue loss and user dissatisfaction [4].

An Intrusion Detection System (IDS) is defined as a hardware or software tool that is used to prevent unauthorized system activities [2]. Initially, intrusion detection systems were used to detect security attacks that happen at the network level, rendering these tools important for deployed systems [3].

Intrusion detection techniques are grouped into two categories: signature-based detection and anomaly-based detection techniques [5]. The former detects known problems based on a database of pre-defined patterns. For example, in security, a signature-based system keeps track of known attack signatures and uses them to detect if a running system matches one of these signatures [5] [6].

Anomaly detection techniques, the focus of this thesis, work differently. They rely on modelling the normal behaviour of the system. The resulting model is then used to detect abnormal behaviours. Several methods have been proposed to model the normal behaviour including statistical methods [7], pattern recognition [6], Hidden-Markov Models [5] [8] [9], and machine and deep learning methods [1] [2].

1.2 Logs and Anomaly Detection

Logging is a mechanism that is used by developers to record runtime data that is later used to analyze the system behaviour [10]. Developers and system administrators commonly use logs to understand the state of the system [4] [11] [12] [13]. When a software system fails, developers use logs to troubleshoot the system in order to understand the causes of the failure and to provide fixes [8-10]. Therefore, logs, generated by software systems, are valuable data sources to diagnose, detect, and correct problems that occur in operation [4] [14].

Log messages can be the only data sources that describe the system's behaviour. Commonly, a log is a message that is written by developers using a logging statement (e.g., print ()), and a log file is composed of several log events [14] [15]. Even though logs play an essential role in debugging and anomaly detection, the traditional analysis of logs, which consists of searching for specific keywords is impractical [1]. This is because log files can be considerably large [16] [17] hindering viable analysis of their content. Additionally, search-based strategies are not adequate for detecting anomalies using logs [1]. We need to develop techniques and tools that can automatically detect anomalies in large logs.

There are studies for the detection of anomalies in logs. A common approach is to use machine learning techniques to model the normal behaviour of the system. The resulting models are then used to detect sequences that do not follow the normal models [18] [19]. Other anomaly detection techniques rely on the concept of log differencing [20], a technique that uses state machines to model the structure of logs and uses this structure to detect logs that deviate from normalcy.

In this thesis, we investigate the application of the multiresolution abnormal trace detection algorithm proposed by Jiang et al. [21] to the detection of anomalies in large logs. The algorithm takes as input execution traces that are generated from normal executions (e.g., by running the system in a controlled environment). It extracts varying length n-grams of events in the traces that are then used to build an automaton, which characterizes the normal behaviour of the system. The automaton is deployed in operation to detect if a trace generated at runtime is normal or not. Jiang et al. proposed a clever way to control the degree of generalization of the automaton using a threshold alpha. By varying alpha, one can build a model that reduces false positives, i.e., false alarms.

The work of Jiang et al., however, focuses on execution traces. Logs and execution traces are different in many ways. Traces are usually used to record a program's control flow [22]. Examples of traces include traces of function calls, system call traces, inter-process communication traces, etc. Unlike traces, logs are generated from statements written by developers. They do not exhibit a causal relationship among their events. In addition, logs contain messages entered by developers using natural language, which may be ambiguous and unprecise. This is because there are no known standards on how and where to log [17], which affect the quality of the generated log events. To make things worse, logs are largely unstructured data, and require the use of techniques for extracting structures. These techniques are known as log parsing techniques [17].

1.3 Thesis Contribution

The principal contribution of this thesis is to answer the following question:

Considering the differences between logs and traces, can the multiresolution abnormal trace detection algorithm proposed by Jiang et al. [21] be used to detect anomalies in log data and, if so, what would be the accuracy?

To answer this question, we propose a process, called LogAutomata, which uses the multiresolution abnormal trace detection algorithm as its primary mechanism. When applying LogAutomata to a large log file generated from the execution of Hadoop Distributed File System (HDFS), we show that LogAutomata detects anomalies with a precision, recall and F1-score of 93%, 81%, and 86%, which is a very promising result.

1.4 Thesis Outline

The rest of the thesis is structured as follows:

In Chapter 2, we present the background and related work relevant to the thesis. We discuss the concept of logs in more details. We also review recent studies on anomaly detection systems, followed by a broad discussion.

In Chapter 3, we present our approach, called LogAutomata, which is based on Jiang et al. [21] generalizable automata to model the normal behaviour of the system using log data. We start the chapter with an overview of the approach and continue with describing each component of our method in details. In this chapter, we also discuss the evaluation of our approach on logs generated from HDFS. We discuss the results and conclude with lessons learned.

In Chapter 4, we revisit the contribution of this thesis. We conclude with comments about our project and present opportunities for future research.

Chapter 2. Background and Related Work

2.1 Overview of Logs

Software developers resort to logging to record important information. Log files can help with many software engineering tasks including debugging, program comprehension, performance analysis, anomaly detection, and regulatory compliance. A log file consists of log events, and a log event is output from the execution of a logging statement (e.g., ‘print’ function), inserted by developers in the source code. Each log event consists of a set of fields to record the system state and significant system event [23] [24] based on the logging convention that is used by the developers of the system [14]. Figure 2.1 shows an example of a raw log event. This event contains timestamp (081109 203519), process id (143), a verbosity level (INFO), the logging function (dfs.DataNode$DataXceiver), a log message, and dynamic variables (Receiving block blk_-16 src: /10.250.10.6:40524 dest: /10.250.10.6:50010). This log event was generated from a logging statement in the HDFS code.

There is no standard format to represent log files [25] [26]. At a high-level, a log event contains three parts [27]: a header, a static part, and a dynamic (variable) part. For example, in the log event of Figure 2.1, the header is the concatenation of the timestamp (081109 203527), process id (148), log verbosity level (INFO), and logging function (dfs.DataNode$DataXceiver) [28]. The log message part consists of “Receiving block (variable) src: (variable) dest: (variable)”. The static parts are “Receiving block ”, “src:”, “dest:” while the dynamic parts (the value of the logged variables) are “blk_-16”, “/10.250.10.6:40524” and “/10.250.10.6:50010”. The static and the dynamic parts form the log message.

Raw Log from HDFS

```
081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-
16 src: /10.250.10.6:40524 dest: /10.250.10.6:50010
```

$\downarrow$

Structured Log

Timestamp	081109 203527
Process id	148
Log verbosity level	INFO
Logging function	dfs.DataNode$DataXceiver
Event Template	Receiving block <*> src: <*> dest: <*>
Parameters	[blk_-16, /10.250.10.6:40524, /10.250.10.6:50010]

Figure 2-1 An example of a Raw and Structured log event

Although logs are valuable data source, working with raw log data is known to be challenging. This is because raw logs are mostly unstructured and contain free text written by developers. Hence, the prerequisite of any viable log analysis technique is to structure the logs first [29] [17]. There are many studies that aim to convert raw log data into a structured format (e.g. [4] [14] [30]). This is called log parsing. The idea of log parsing is to differentiate between the dynamic part and the static part of log events. More precisely, log parsing is used to convert unstructured logs into a structured format by extracting log templates that characterize the content of log events and help the automate log analysis [4] [31]. Log templates consist of static part and dynamic part of a log when the value of dynamic part replaced by <*>. Log templates also called log keys or message types [10] [28] and to illustrate, the log template corresponding to the example in Figure 2.1 is

Receiving block <*> src: <*> dest: <*> and as explained, variables were replaced with the symbol <*>

One way to extract log templates from raw log events would be to use regular expression [10]. This solution, however, may turn to be ineffective when distinguishing between the static and the dynamic parts of a log event. First, there may be many templates within the same log file, requiring the development of several regular expressions. Additionally, as the system evolves, these regular expressions should constantly evolve as well.

In recent years, many log parsing techniques have been proposed including LKE [15], LogSig [32], SHISO [33], IPLoM [35], and Drain [27]. These techniques vary in their design and effectiveness for scaling up to large logs. A comprehensive survey of log parsing tools is presented by El-Masri et al. [17].

2.2 Review of Log-Based Anomaly Detection Techniques

Existing log-based anomaly detection techniques can be grouped depending on the method used including finite state machines (FSM) and invariant mining, clustering, and Principal Component Analysis (PCA)-based approaches, and these of deep learning algorithms.

2.3 Finite state Machine and Invariant Mining Analysis

FSM-based approaches model the behaviour of the system that produces the logs and then the behavioural change of the system is used to find anomalies [12]. Fu et al. [15] presented a log analysis method that operates on unstructured logs to address anomaly detection in distributed systems and performed an empirical evaluation on Hadoop and SILK datasets. In their approach, they first converted logs to templates (called keys in their study) and then constructed an FSM from a training set of log key sequences. The resulting FSM represents the normal behaviour of the system. They calculated the time taken to transit from one state to another state to record potential performance degradation in the system.

Beschastnikh et al. [30] created a model called Synoptic that performs based on temporal invariant mining. The aim is to infer accurate models from the system behaviour. Their generated model is similar to a finite state machine. Synoptic parses raw logs and then mines three types of temporal invariants in each of them. After that, it leverages those temporal mined invariants to make a model. The authors argued that the three types of invariants result in accuracy improvement. In addition, to explore the space of the model, Synoptic uses refinement as a second criterion.

Amar et al. [11] proposed the use of finite state machines to compare logs. In their work, 2KDiff algorithm and nKDiff algorithms are presented, which are based on the classical k-Tails algorithm [36]. The researchers showed that 2KDiff can compare two logs only, and nKDiff is used to compare one log file to many logs. They evaluated their approach on mutated logs that were generated based on the models. They also conducted a user study to demonstrate the effectiveness of their approach.

Beschastinlk et al. [37] proposed CSight, a tool based on Communicating Finite State Machines (CFSM) that can detect anomalies in traces generated from distributed or concurrent systems. CSight mines the set of temporal invariants and makes a model that satisfies temporal invariants. The input of the tool is a system's execution traces with a requirement of having vector timestamps, and they provided a tool that automatically adds vector timestamps to system logs. Their evaluation of logs from three different network systems and a user's study on bug finding shows that CSight infers models accurately.

Goldstein et al. [12] suggested a method for comparison between service execution to identify behavioural changes of complex systems. The authors mine logs using FSM, enhanced with performance-related data to capture behavioural models from normal and abnormal execution of a system. The evaluation on real telecommunication logs showed that the method is helpful to identify undesired behaviours.

2.4 Clustering based related predictions

Vaarandi [6] did one of the primary clustering works that focuses on clustering log messages into different groups and flagging objects that do not belong to any clusters as anomalies. The author presents a clustering algorithm that detects frequent patterns from logs. In order to implement the clustering algorithm, an experimental tool called SLCT (Simple Logfile Clustering Tool) has been developed, and the experiment shows the usefulness of SLCT for building log file profiles.

He et al. [1] represented the application of three common supervised anomaly detections and three unsupervised anomaly detection methods: for supervised learning, they applied logistic regression, decision trees, and Support Vector Machine, and for unsupervised learning, PCA, log clustering, and invariant mining have been investigated.

Xu et al. [14] used a combination of source code analysis with information retrieval method to detect abnormal logs sequences. Their approach, though useful, requires the presence of source code. Lin et al. [10] identified a log-based system problem with their proposed log clustering-based service called, LogCluster. To cluster the logs, log sequences are converted to a vector with related weight, while different log events have different weights based on their importance on the identification of the problem. After vectorization, they applied similarity metric between two sequences, and log sequences are grouped to different clusters using a clustering technique. LogCluster checks whether a logging cluster is a recurrent cluster to identify a problem.

Log3C [4] was designed as a novel clustering approach to identify system problems that lead to the decline of the system Key Performance Indicators (KPIs) using log sequences and system KPIs. He et al. [15] proposed a clustering-based approach that parses the data and vectorizes each log sequence and gives weight to different templates by Inverse Document Frequency (IDF) weighting. IDF is a method in text mining that gives higher weight to rare templates and lower weight to more frequent ones.

2.5 Use of deep learning algorithm

There is a recent interest in exploring deep learning algorithms on logs. We briefly review a few examples based on the deep neural network model:

Du et al. [23] proposed DeepLog, which is a neural network model based on the application of Long Short-Term Memory (LSTM). DeepLog models system logs as a natural language sequence. The authors consider logs as a sequence of words with related natural language rules and grammatical restrictions. They leverage log parsing and store both templates (static value) and parameter values (dynamic value) of logs into two separate vectors, and their LSTM model check first the template to evaluate if its normal. Moreover, the authors performed an incremental online update of the Deeplog model to be adaptable to the new log patterns over time.

Another LSTM model [38] was introduced by Zhang et al., namely, LogRobust, to tackle the problem of instability in anomaly detection. Based on the authors' view, log data is not stable over time, and log data contains log events that are not seen previously. To plan the work, the authors extracted the semantic information of logs embedded in log events and represented extracted information as semantic vectors. Then, these vectors are given as input to a Bidirectional Long Short-Term Memory (Bi-LSTM) classification model to find anomalies. Bi-LSTM model learns the information in existing log sequences in context, and captures the importance of different log templates, and uses the captured characterizations to handle unstable log events and sequences.

Chapter 3. Anomaly Detection Approach

3.1 Overview

In this chapter, we elaborate on our method for modelling a system to be able to detect anomalies later. Our approach adapts Jiang et al.'s multiresolution abnormal trace detection algorithm [21] to the detection of anomalies in log files. We first start by describing the multiresolution abnormal trace detection algorithm. We then continue with introducing LogAutomata. The evaluation of LogAutomata using log files generated from the Hadoop Distributed File System (HDFS) is presented, followed with a discussion.

3.2 Jiang et al.'s Multiresolution Abnormal Trace Detection Algorithm

Jiang et al. [21] presented an algorithm to detect abnormal sequences in execution traces. Unlike logs, execution traces consist of sequences of events that exhibit a causal relationship. Examples of execution traces included traces of routine calls, component invocations, inter-process communication, etc. Jiang et al.'s approach takes traces generated in a lab environment (a training set) and creates a model based on varied-length n-grams and automata that represents the normal behaviour of the system. Once deployed, the model is used to detect whether new traces (a testing set) comply with the automaton or not. The algorithm flags traces that deviate significantly from the model as anomalies. Jiang et al.'s approach comprises three major steps, presented here and discussed in more detail in the following subsections:

- Step 1. Extracting varied-length n-grams from the traces using a threshold α that controls the degree of generalization of the automaton.

- Step 2. Constructing the automaton using the n-grams of Step 1.
- Step 3. Detecting anomalies using the automaton constructed in Step 2.

Jiang's approach starts by extracting varied-length n-gram elements from the traces of the training set to be used as states of the automaton (see Appendix A.1 for more details about the varied-length n-gram extraction algorithm). To illustrate this step, let us consider a training set of four traces AEDBC, DBCA, DBCEA, DBC. The distinct elements of these traces form the alphabet set T = {A, B, C, D, E}.

Given these traces as input, the n-gram extraction algorithm starts with building a table of 1-grams with the frequency of their appearance in the traces. Table 3.1 shows the results of the extracted 1-grams (k =1) from the sample traces AEDBC, DBCA, DBCEA, DBC. As we can see, "A" appears three times in the training set (more precisely in traces AEDBC, DBCA and DBCEA), "B" appears four times (appears once in each trace), etc. In the next iteration, the algorithm builds a set of 2-grams (k=2) with their frequencies (e.g., "AE" appears once and this is in Trace AEDBC).

To control the length of the final n-grams, the algorithm introduces a condition based on a threshold alpha, which varies from 0 to 1. A k-gram element X, which will build, using (k-1)-grams Y and Z is considered in the next iteration (i.e., when computing (k+1)-grams) only and only if frequency(X) > alpha*min (frequency (Y), frequency(Z)). The frequency refers to the number of times an element appears in all the traces of the training set.

If we set alpha = 0.6, we can see in Table 3.1 that AE (1), CA (1), CE (1), ED (1), and EA (1) do not meet this condition and therefore they are not considered in the next iteration, i.e., when building a 3-gram table. The largest n-gram is "DBC" (k = 3), which is repeated 4 times in the traces of the training set.

Table 3.1. An example of an n-gram extraction table using traces AEDBC, DBCA, DBCEA, DBC and alpha = 0.6

k = 1	k = 2	k = 3
A (3)	AE (1)	**DBC (4)**
B (4)	**BC (4)**	
C (4)	CA (1)	
D (4)	CE (1)	
E (2)	**DB (4)**	
	ED (1)	
	EA (1)	

The next step of Jiang et al's approach is to construct the automaton. The construction happens by sorting the extracted n-grams first. They are sorted firstly based on their length and then, based on their frequency when the most frequent one ranked firstly, and after sortation, the automata is built by treating the largest n-grams as single elements. In our case, DBC is the largest n-gram. The sample traces AEDBC, DBCA, DBCEA, DBC can be viewed as AEDBC, DBCA, DBCEA, DBC where DBC is now represented as one element. The automaton construction algorithm (shown in Appendix A.2.) starts by building a table where the rows and columns consists of the n-grams of Table 3.1 that meet the threshold condition (shown in bold in the table). The value of cell X, Y is set to 1 if there is at least one trace in the training set (after replacing the n-grams) where element Y appears after X. It is set to 0 otherwise. For example, in Table 3.2 which shows the automaton table that is generated from the sample traces and the n-gram table of Table 3.1, Cell (A, E) is set to 1 because AE appear in trace AEDBC.

Table 3.2. Automaton construction table example

Extracted n-grams	A	D	B	C	E	DB	BC	DBC
A	0	0	0	0	1	0	0	0
D	0	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0
E	1	0	0	0	0	0	0	1
DB	0	0	0	0	0	0	0	0
BC	0	0	0	0	0	0	0	0
DBC	1	0	0	0	1	0	0	0

The generated automaton using alpha = 0.6 is shown in Figure 3.1.

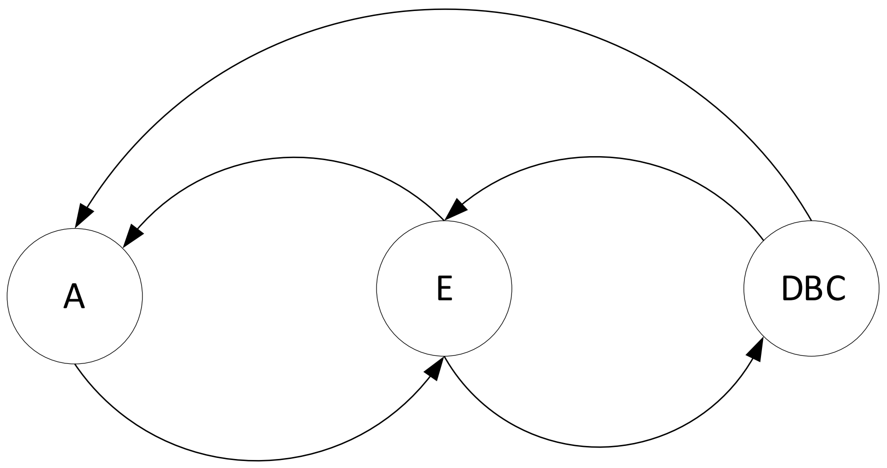


Figure 3-1 Generated automaton from traces AEDBC, DBCA, DBCEA, DBC using alpha = 0.6

The threshold alpha not only controls the largest length of the final n-grams, but also the degree of generalizability of the resulting automaton. If alpha = 1 then the longest n-grams become the distinct elements of the traces, i.e., “A”, “B”, “C”, “D”, and “E”. The n-gram table for alpha = 1 is shown in Table 3.3. The generated automaton is shown in Figure 3.2. This automaton is too flexible since it will consider all sorts of traces that combine the distinct elements of the training set as valid traces (e.g., AEAEAEAE), which may reduce the accuracy of the approach.

Table 3.3. An example of an n-gram extraction table using traces AEDBC, DBCA, DBCEA, DBC and alpha = 1

k = 1
A (3)
B (4)
C (4)
D (4)
E (2)

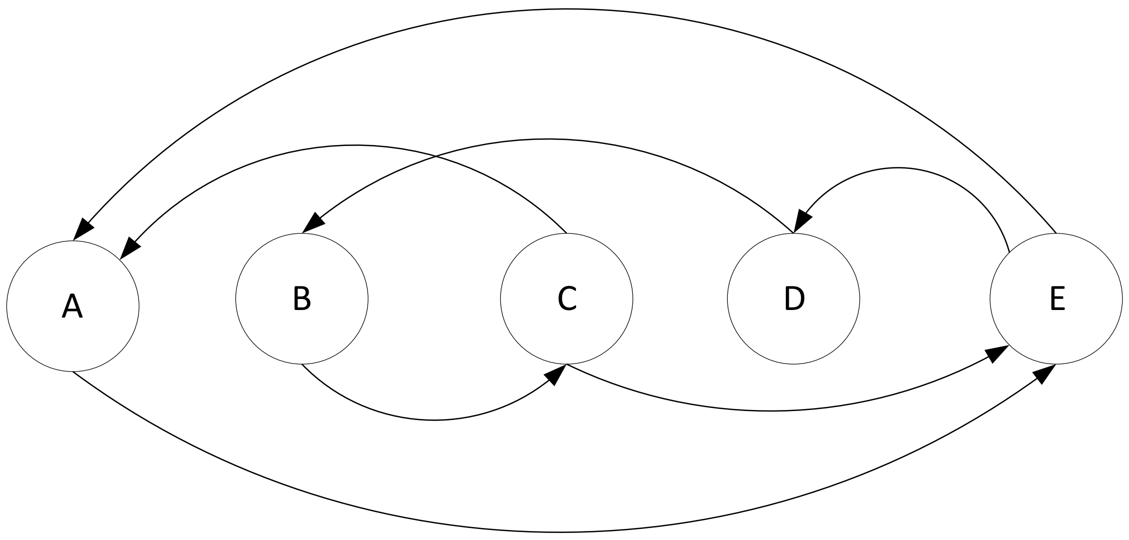


Figure 3-2 Generated automaton from traces AEDBC, DBCA, DBCEA, DBC using alpha = 1

On the other hand, setting alpha = 0 will lead to each trace as a state of the automaton, resulting in an automaton that is too rigid, failing to recognize unseen cases. The challenge is to find the setting for alpha that can improve the accuracy of the anomaly detection algorithm while reducing false positives. In our case, we determine alpha using a validation set of log data.

3.3 LogAutomata: Application of the Multiresolution Anomaly Detection Algorithm to Log Data

Our approach for applying the multiresolution anomaly detection algorithm [21] to log data is shown in Figure 3.3 and consists of three phases: Training, validation, and testing. During the training phase, LogAutomata takes a set of normal log files (i.e., log files that do not contain anomalies) and builds ten automata by varying alpha from 0 to 1. The role of the validation is to select the alpha value that yields to the best accuracy when used on a sample of normal and anomalous logs. Finally, using the selected automaton, we experiment with a larger set of normal and anomalous logs to evaluate the performance of the model in the testing phase.

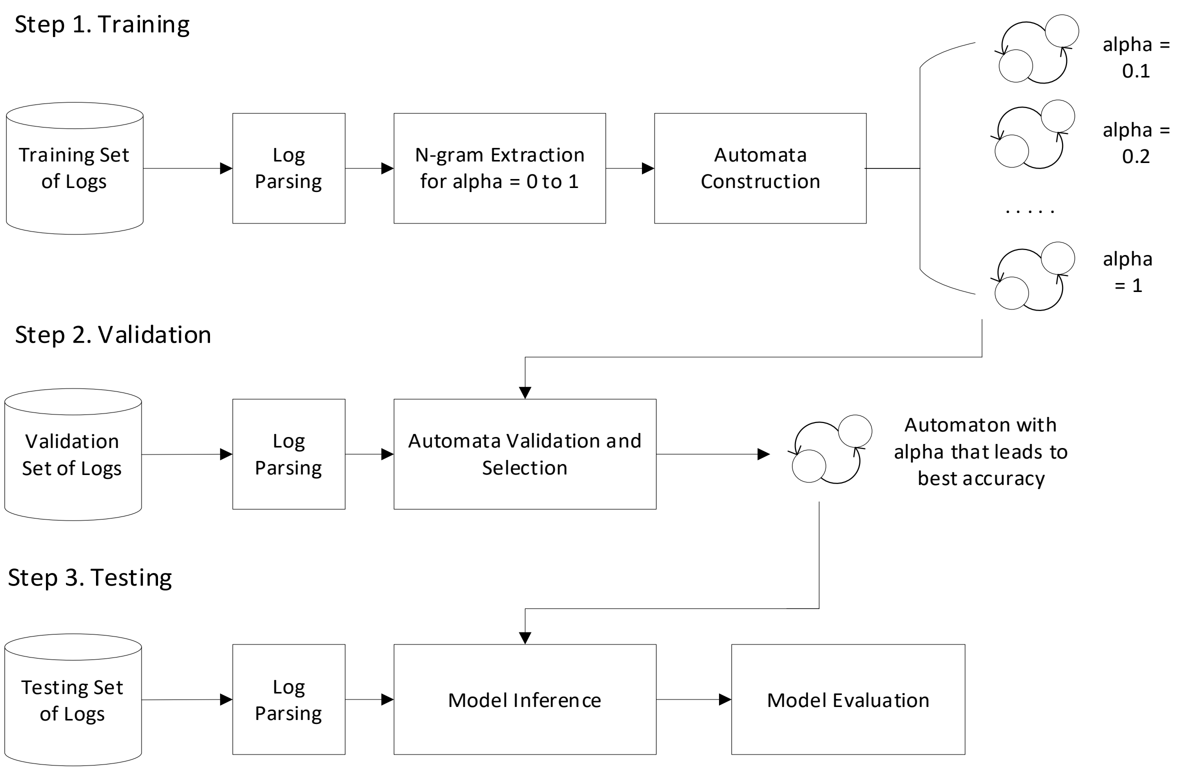
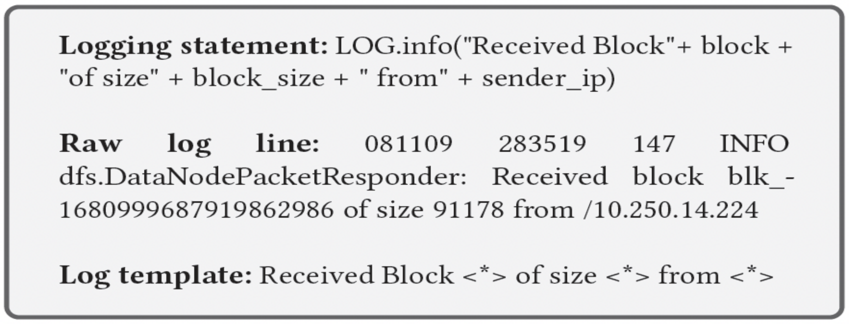


Figure 3-3 LogAutomata approach

All these phases require an important step, which is parsing logs. Unlike execution traces, logs are unstructured, making it difficult to extract meaningful information from large raw log files. This is mainly caused because the practice of logging is largely ad hoc with no known guidelines and best practice. There are also many logging libraries at the disposal of software developers. These libraries use different logging formats.


Logging statement: LOG.info("Received Block"+ block + "of size" + block_size + " from" + sender_ip)

Raw log line: 081109 283519 147 INFO dfs.DataNodePacketResponder: Received block blk_-1680999687919862986 of size 91178 from /10.250.14.224

Log template: Received Block <*> of size <*> from <*>

Figure 3-4 Example of log parsing

Log parsing consists of automatically converting unstructured raw log events into a structured format that would facilitate future analysis. Log parsing techniques focus on parsing the log message. The log header (timestamp, log level, process ID, logging function) usually follows the same structure within the same log file [28]. Parsing a log message is further reduced to the problem of automatically distinguishing the static text from the dynamic variables. An example of parsing a log event is shown in Figure 3.4. The example shows the logging statement in the code, the generated log event, and the parsed log event.

The result of parsing this log event takes the form of a template where the static and dynamic content are clearly identified. In the case of the above example, the extracted template is "Received block <*> of size <*> from <*>"

Automate log parsing is an active research topic in recent years due mainly to the increasing diversity in the types of logs that are generated by various applications. There exist many log parsers tools that were proposed in the literature. In this thesis, we select to use Drain[1], a powerful log parsing tool with a very high accuracy that was developed by He et al. [27], what is available online as an open-source product.

Drain is an online parser which accurately parses logs in a streaming and timely manner. As it is an online parser, it does not need an offline training step and therefore, making it very practical. The tool leverages parsing rules using heuristics to extract log templates from raw log data. To this end, it applies a fixed-depth tree structure in order to put each log in corresponding log group. The way it deals with a new raw log is that it first preprocesses raw logs according to regular expressions based on domain knowledge and then assigns each log event to a log group. One way

[1] https://logparser.readthedocs.io/en/latest/tools/Drain.html

to group log events would be to compare each log event with the log events of the entire file. This is time consuming and impractical. Instead, Drain uses a clever algorithm that is based on a fixed-length parse tree to speed up the group assignment. The tool assigns a new log event to the most similar log group. If none exists, the tool creates a new log group. The details of the algorithm are presented in [27].

3.4 Experiment

3.4.1 Dataset

To assess the proposed anomaly detection model, we used a dataset of labelled logs made publicly available in the LogHub Github repository[2]. The repository is maintained by the LogPai research team, and the log datasets are discussed by He et al. in [24]. The repository provides both labeled and unlabeled datasets of logs. These logs are generated from various software systems including Linux, Hadoop, Android, etc.

In this thesis, we assessed our model using log data generated from the Hadoop Distributed File System (HDFS)[3]. HDFS is a file system created for the purpose of storing large files and more for batch processing [39]. It is written in Java, and it has more than 11 million textual console logs, collected from a Hadoop cluster that runs on 203 Hadoop nodes [14] [24]. The collected logs are unchanged without any modification, saying that the collected logs are gathered as ‘they are' [14]. There are two versions of HDFS log files on LogHub repository [24]. In our work, we use HDFS-1, the labeled version of the data, where normal and anomalous logs are clearly identified[4]. The characteristics of the log data is presented in Table 3.4.

[2] https://github.com/logpai/loghub
[3] https://github.com/logpai/loghub/blob/master/HDFS
[4] ttps://zenodo.org/record/1596245#.XMMZ1dv7S- Y

Table 3.4 Characteristics of the HDFS log data used for the evaluation

Data	**Size**
Total number of log events	11,175,629
Size of log file	1.58 GB
Total number of normal log events	10,887,379
Total number of anomalous log events	288,250

The HDFS logs were created using the Log4j package[5], and in log4j the format of log is as below:

- Date and time format
- The line number from where the logging requests
- The logging priority
- The logging name being set via getLogger()
- The message to log

Examples of log events from the HDFS log dataset are shown below. Each log event contains a log header and a log message.

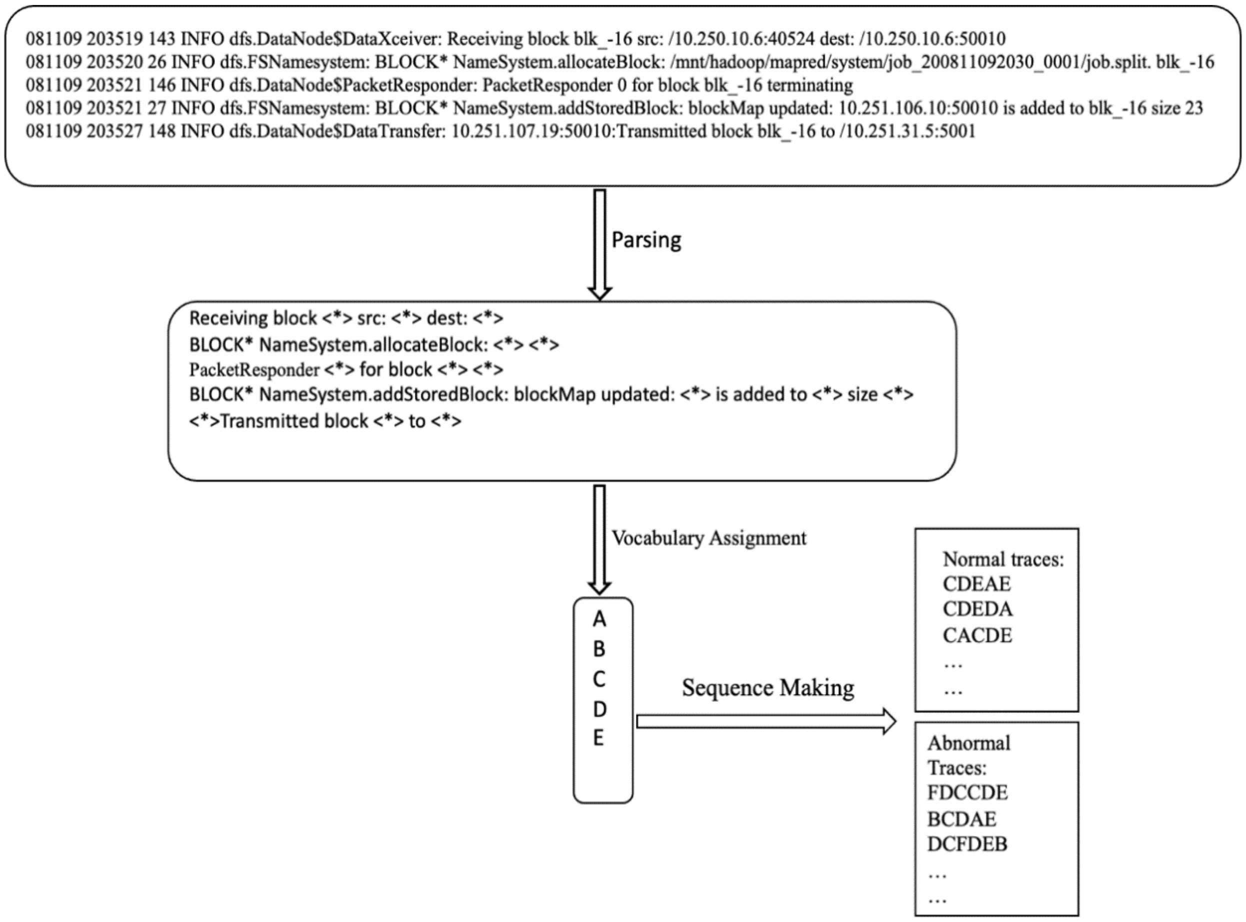
- *081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-16 src: /10.250.10.6:40524 dest: /10.250.10.6:50010*

[5] https://en.wikipedia.org/wiki/Log4j

- *081109 203532 147 INFO dfs.DataNode$DataXceiver: Receiving block blk_-1385756122847916710 src: /10.251.203.166:42786 dest: /10.251.203.166:50010*
- *081109 203532 28 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /user/root/rand/_temporary/_task_200811092030_0001_m_000221_0/part-00221. blk_-1385756122847916710*
- *081109 203628 150 INFO dfs.DataNode$PacketResponder: PacketResponder 1 for block blk_-1385756122847916710 terminating*
- *081109 203628 150 INFO dfs.DataNode$PacketResponder: Received block blk_-1385756122847916710 of size 67108864 from /10.251.203.166*

We are working with files of distributed systems. A distributed system has various components spread across multiple devices[6], and HDFS's characteristic is that when data is received by it, the information is divided to separate blocks and these blocks are stored in a set of containers (DataNodes)[7]. A block identifier, block_id, serves as an identifier in HDFS log event, and all logs containing the same block_id convey information about that specific block operation such as allocating, writing, replicating, and deleting the single block [1].

[6] https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html
[7] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html



```
081109 203519 143 INFO dfs.DataNode$DataXceiver: Receiving block blk_-16 src: /10.250.10.6:40524 dest: /10.250.10.6:50010
081109 203520 26 INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /mnt/hadoop/mapred/system/job_200811092030_0001/job.split. blk_-16
081109 203521 146 INFO dfs.DataNode$PacketResponder: PacketResponder 0 for block blk_-16 terminating
081109 203521 27 INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: 10.251.106.10:50010 is added to blk_-16 size 23
081109 203527 148 INFO dfs.DataNode$DataTransfer: 10.251.107.19:50010:Transmitted block blk_-16 to /10.251.31.5:5001
```

```
Receiving block <*> src: <*> dest: <*>
BLOCK* NameSystem.allocateBlock: <*> <*>
PacketResponder <*> for block <*> <*>
BLOCK* NameSystem.addStoredBlock: blockMap updated: <*> is added to <*> size <*>
<*>Transmitted block <*> to <*>
```

Figure 3-5 Example of parsing HDFS log events using DRAIN

The application of Drain to parse the HDFS log dataset resulted in different log templates. We labelled the templates with alphabet letters (“A”, “B”, “a”, etc.). We replaced each log event in the dataset (normal and abnormal event) with its corresponding alphabet letters, as shown in Figure 3.5.

Using the fact that the logs with the same block_id pictures the execution flow of that particular session [1] [2], we used the dynamic variable “block-id” to group log events that belong to the same flow of execution. To form the sequence of normal (and abnormal) log events, we first detect block_ids, then group message with the same block_id and create the sequence of events happening in each block_id. Therefore, we derive an execution sequence out of each block-id for both the normal and abnormal data file**.** The results are 575,061 sequences of events that characterise normal and abnormal behaviours of HDFS. In total, we extracted 558,223 normal sequences and 16,838 anomaly sequences, and then perform our experiment with unique normal sequence and unique abnormal sequence. These sequences are used to train, validate, and test the generalizable automaton.

3.4.2 Data Splitting

We need to split the data into training, validation, and testing sets in order to apply LogAutomata. Following the common practice in machine learning, our training set contains 70% of normal log events selected randomly. The validation set contains 10% of the normal logs and 10% of the abnormal logs. The testing set is composed of the remaining 20% of the normal logs and 90% of the abnormal logs.

3.4.3 Evaluation Metrics

To evaluate our approach, we use precision, recall, and F1-score [5] which are the accuracy metrics being used mostly in literatures.

- Precision = $\frac{TP}{TP+FP}$
- Recall = $\frac{TP}{TP+FN}$
- F1-score = $\frac{2 \times Precision \times Recall}{Precision+Recall}$

where:

- True positive (TP) relates to the sequences that are abnormal in the ground truth, and the model decision assigns accurately an abnormal label to them.
- False Positive (FP) relates to the sequences that are normal in the ground truth, and the model decision inaccurately assigns an abnormal label to them.
- False Negative (FN) relates to the sequences that are abnormal in the ground truth, and the model decision inaccurately assigns a normal label to them [5].

3.4.4 Results

We trained 10 automata by varying alpha from 0 to 1 with bonds of 0.1. Using the validation set, we measure the accuracy of each automaton. The aim is to determine the value of alpha that yields the best accuracy in terms of precision, recall, and F1-score. The results are shown in Table 3.5.

Table 3.5 Results of the validation phase

Alpha	Precision	Recall	F1-score
0	0.22	1.00	0.36
0.1	0.28	0.96	0.43
0.2	0.56	0.88	0.68
0.3	0.7	0.77	0.74
0.4	0.92	0.43	0.58
0.5	0.98	0.4	0.56
0.6	0.98	0.39	0.56
0.7	0.98	0.32	0.48
0.8	0.97	0.26	0.42
0.9	0.97	0.26	0.42
1	0.97	0.25	0.39

Table 3.6 The number of n-grams generated using various alpha values

Alpha	**Number of extracted n-grams**
0	1,234,058
0.1	56,653
0.2	1,665
0.3	708
0.4	300
0.5	42
0.6	39
0.7	31
0.8	28
0.9	26
1	19

Based on the results of the validation set, we found that alpha=0.3 gives the highest F1-score for HDFS logs. Table 3.6. shows the number of n-grams that are generated for value of alpha. With alpha = 0.3, the automaton contains 708 states (i.e., the number of n-grams).

Therefore, we use the generated automaton with alpha = 0.3 as the model that captures the normal behaviour of the HDFS logs, used during the testing phase. The results of classifying the logs in the testing set are shown in Table 3.7. The approach has 86% accuracy, which is considered excellent.

Table 3.7 Results of LogAutomata on HDFS testing set with alpha = 0.3

Measure	**Result**
Precision	0.93
Recall	0.81
F1-Score	0.86

When it comes to the time it takes for classification of a log event, the value of alpha is deterministic, and the processing time to classify a log event differs based on the value of alpha. For smaller alpha, it takes more time because the model extracts more n-grams. However, the condition becomes looser, and the process speeds up with the improvement of alpha as expected.

3.5 Discussion

Generalization of the automata: The excellent results obtained by LogAutomata is attributed to the automaton generalization power of Jiang et al.'s algorithm. Based on the precision and recall of Table 3.6, the approach yields a very low false positive rate (only 7%). This is important for the adoption of the approach. A high false positive rate may deter users from using the approach because of the high number of alerts that the system would produce. This said, it is important to determine how much generalization we should have in order to obtain good detection accuracy. The risk with generalization is to end up with an automaton that is either too loose, which may affect the true positives (abnormal logs classified as normal) or too strict, which detects everything as anomaly (high false positive rate). In this thesis, an alpha value of 0.3 has shown to yield best accuracy. We expect this to differ from one system to another. Therefore, we propose that a tool that implements LogAutomata should allow enough flexibility to users to adjust alpha as they see best fit. For example, they can decide to experiment with different validation sets and data sizes so as to obtain an alpha value that increases the confidence of detecting many anomalies. Another approach would be to adjust alpha dynamically as the system under observation is running. This would mean that we need to embed LogAutomata with a system that learns over time, perhaps through the use of online learning techniques such as those used in reinforcement learning.

Precision vs. recall: Our results favor precision over recall (93% precision as compared to 81% recall). In other words, LogAutomata was successful in keeping the number of false positives low, at the detriment of detecting less anomalies (a high false negative rate). This can also be changed by varying the value of alpha. As shown in Table 3.5, a different alpha may increase the number of false positive while reducing the number of false negatives at a risk of having a lower overall F1-score. The decision of whether we should focus on precision or recall depends on the application of the anomaly detection. In our case, we aim to detect performance degradation in HDFS. As such, missing some anomalies may not be a major issue compared to the detection of security attacks. In general, finding the right trade-off between precision and recall is not an easy task. In this thesis, we use F1-score as a measure of accuracy to balance precision and recall.

3.6 Threats to Validity and Limitations

We now discuss the threats to the validity of our results and recommendations.

Construct Validity: Construct validity threats concern the accuracy of the observations with respect to the theory. In this thesis, we reuse Jiang et al.'s algorithm for multiresolution anomaly detection using generalizable automata. The algorithm stems from strong theoretical background in automata. Additionally, we followed traditional machine learning steps for training, validating, and testing our models. Thus, we argue that there is no major threat to the construct validity of our results.

Internal Validity: Internal validity threats are related to the set of factors that may influence our results. The selection of the dataset is one of the common threats to validity for an anomaly detection approach. It is possible that the normal and abnormal logs we use in the evaluation section have common properties that we are not aware of and that may invalidate the results. To mitigate this threat, we used HDFS logs that were generated specifically for the purpose of anomaly detection. The authors of this dataset (see [14]) carefully labelled the HDFS dataset to clearly represent normal executions of the systems and those caused by anomalies. Additionally, the HDFS data is used in similar studies. This said, we acknowledge that we need to apply our approach to other datasets. Another internal threat to validity is the use of Drain. We did not reimplement the tool. We simply used the implementation provided by the authors (see [27]). Errors in this implementation may impact the log parsing step of our approach. Finally, the implementation of LogAutomata may contain errors that may impact the results. To mitigate this threat, we carefully checked the implementation and tested it against sample data. We performed our study on HDFS logs. We need to apply LogAutomata to other log data to claim generalizability. We defer this to future work.

Conclusion Validity: Conclusion validity threats correspond to the correctness of the obtained results. We provide a Github repository to LogAutomata implementation to allow other researchers the possibility of the assessment and reproducibility of our results.

Limitations: The size of the dataset that we used in this thesis is 1.58 GB. For larger log files, our approach may generate a large number of n-grams, which may hinder scalability and performance. Another important limitation of the approach is in fact that it is a supervised method, which requires a labelled dataset for training. This may be difficult to obtain in practice since the system needs to be run in a controlled environment to generate normal executions. Additionally, it may not always be possible to characterize the normal behaviour of the system in a lab environment. Adding to this, as the system evolves, we need to constantly retrain the model to reflect the normal behaviour of the system.

Chapter 4. Conclusion and Future Work

4.1 Conclusion

In this thesis, we presented LogAutomata, which is an application of Jiang et al. [21] multiresolution abnormal trace detection algorithm to the detection of anomalies in system logs. The LogAutomata process starts by turning unstructured log data into a structured format using a log parsing tool. We used Drain in this thesis, but other tools can readily be used as well. Algorithm. The next step is to generate varying length n-grams, which can then be used to build an automaton that characterizes the normal behaviour of the system. The approach uses a threshold alpha that controls the degree of generalization of the automaton. Using a validation set, we determine the proper alpha value for our dataset. The training model is then used detect anomalies in large logs. When applying LogAutomata to a large log file generated from the execution of HDFS, we obtained an F1-score of 86%, which shows that Jiang et al's multiresolution algorithm can be a very effective way to detect anomalies in log data. This is particularly important knowing that unlike execution traces used in Jiang et al's study, logs do not exhibit a causal relationship among their events. They are also unstructured data and hence require the use of parsing techniques to extract the information necessarily to build the training, validation, and testing sets.

4.2 Opportunities for Further Research

We should investigate the performance of the proposed approach on logs of other systems to be able to generalize the results of this study.

Another direction of future work is to investigate how the generalizable automata technique can be combined with other machine learning models. To this end, ensemble methods could be a future direction to improve the result as they often prove to perform better than leveraging individual learning methods. Ensemble methods combine several learning models instead of only one. for instance, Islam et al. [5] proposed and assessed a novel method of ensemble learning on sequential data. To create the model, they used weighted pruning Boolean combinations for selecting and combining their detectors in order to improve the performance of the anomaly detection model and reduce the false alarm rate. Their empirical result shows that the weighted pruning approach performs well and improves the accuracy of existing Boolean combination methods while reducing the time is taken to combine the detectors. We should investigate how the generalizable automata approach can be used as a classifier for the ensemble method.

Additionally, for larger log files, we may need to develop better processing techniques to reduce their size before using them to create the training model. For this purpose, we propose two directions. The first one is to investigate the use of sampling and abstraction techniques such as those studied in the area of trace abstraction (see [34] [40] [41]) to understand how they can be applied to reduce the size of logs while keeping as much of the needed information to characterize the normal behaviour of the system as possible.

The second direction is to understand how text messages (static part) of log events can be used to distinguish between normal and abnormal executions of the system. These messages are written by developers to reflect what the system is doing and why. We believe that these messages can be exploited for the purpose of anomaly detection. An example of using text mining techniques with applications to run-time data can be found in the work of Pirzadeh et al. [42]. The authors showed that natural language processing techniques are useful in extracting important information from execution traces. This work can inspire the analysis of log messages using text mining.

Bibliography

[1] S. He, J. Zhu, P. He, and M.R. Lyu, "Experience report: System log analysis for anomaly detection," in *Proc. of the IEEE 27th International Symposium on Software Reliability Engineering (ISSRE),* pp. 207-218, 2016.

[2] S. Lu, X. Wei, Y. Li, and L. Wang, "Detecting anomaly in big data system logs using convolutional neural network," in *Proc. of the IEEE 16th International Conference on Dependable, Autonomic and Secure Computing,* pp. 151-158, 2018.

[3] H.J. Liao, C.H.R. Lin, Y.C. Lin, and K.Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, *36*(1), pp.16-24, 2013.

[4] S. He, Q. Lin, J.G. Lou, H. Zhang, M.R. Lyu, and D. Zhang, "Identifying impactful service system problems via log analysis," in *Proc. of the ACM 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,* pp. 60-70, 2018.

[5] M.S. Islam, W. Khreich, and A. Hamou-Lhadj, "Anomaly detection techniques based on kappa-pruned ensembles," *IEEE Transactions on Reliability*, *67*(1), pp. 212-229, 2018.

[6] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. of the IEEE 3rd Workshop on IP Operations & Management (IPOM 2003),* pp. 119-126, 2003.

[7] N. Busany and S. Maoz, "Behavioral log analysis with statistical guarantees," in *Proc. of the IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 877-887, 2016.

[8] W. Khreich, S. S. Murtaza, A. Hamou-Lhadj, and C. Talhi, “Combining heterogeneous anomaly detectors for improved software security,” *Journal of Systems and Software*, *Volume 137*, pp.415-429, 2018.

[9] W. Khreich, B. Khosravifar, A. Hamou-Lhadj, and C. Talhi, “An anomaly detection system based on variable N-gram features and one-class SVM,” *Information and Software Technology*, *91*, pp.186-197, 2017.

[10] Q. Lin, H. Zhang, J.G. Lou, Y. Zhang, and X. Chen, “Log clustering based problem identification for online service systems,” in *Proc. of the 38th International Conference on Software Engineering Companion*, pp. 102-111, 2016.

[11] H. Amar, L. Bao, N. Busany, D. Lo, and S. Maoz, “Using finite-state models for log differencing,” in *Proc. of the ACM 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 49-59, *2018*.

[12] M. Goldstein, D. Raz, and I. Segall, “Experience report: Log-based behavioral differencing,” in *Proc. of the IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 282-293). IEEE, 2017.

[13] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, “Recurrent neural network attention mechanisms for interpretable system log anomaly detection,” in *Proc. of the First Workshop on Machine Learning for Computing Systems*, pp. 1-8, 2018.

[14] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, “Large-scale system problem detection by mining console logs,” in *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles,* pp. 117–132, 2009.

[15] Q. Fu, J.G. Lou, Y. Wang, and J. Li, “Execution anomaly detection in distributed systems through unstructured log analysis,” in *Proc. of the IEEE 9th International Conference on Data Mining,* pp. 149-158, 2009.

[16] A. Miranskyy, A. Hamou-Lhadj, E. Cialini, and A. Larsson, “Operational-log analysis for big data systems: Challenges and solutions,” *IEEE Software*, *33*(2), pp.52-59, 2016.

[17] D. El-Masri, F. Petrillo, Y-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, “A systematic literature review on automated log abstraction techniques,” *Information and Software Technology*, volume 122, pp.106-276, 2029.

[18] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, “Failure prediction in ibm bluegene/l event logs,” in *Proc. of the IEEE 7th International Conference on Data Mining (ICDM 2007),* pp. 583-588, 2007.

[19] S. Nedelkoski, J. Bogatinovski, A. Acker, J. Cardoso, and O. Kao, “Self-attentive classification-based anomaly detection in unstructured logs,” in *Proc. of the IEEE International Conference on Data Mining (ICDM)* (pp. 1196-1201). 2020.

[20] L. Bao, N. Busany, D. Lo, and S. Maoz, “Statistical log differencing,” in Proc. of the *IEEE/ACM 34th International Conference on Automated Software Engineering (ASE)*, pp. 851-862, 2019.

[21] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, “Multiresolution abnormal trace detection using varied-length n-grams and automata,” *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, *37*(1), pp.86-97, 2005.

[22] L. Alawneh, and A. Hamou-Lhadj, “Execution traces: A new domain that requires the creation of a standard metamodel,” in *Proc. of the International Conference on Advanced Software Engineering and Its Applications*, pp. 253-263, 2009.

[23] M. Du, F. Li, G. Zheng, and V. Srikumar, “Deeplog: Anomaly detection and diagnosis from system logs through deep learning,” in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security,* pp. 1285-1298, 2017.

[24] S. He, J. Zhu, P. He, and M.R. Lyu, “Loghub: a large collection of system log datasets towards automated log analytics,” *arXiv preprint arXiv:2008.06448, 2020*.

[25] J. Zhu, P. He, Q. Fu, H. Zhang, M.R. Lyu, and D. Zhang, “Learning to log: Helping developers make informed logging decisions,” in *Proc. of the IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 415-425, 2015.

[26] B. Chen, Z. Jiang, “Characterizing logging practices in java-based open-source software projects–a replication study in apache software foundation,” *Empirical Software Engineering*, *22*(1), pp.330-374, 2017.

[27] P. He, J. Zhu, Z. Zheng, and M.R. Lyu, “Drain: An online log parsing approach with fixed depth tree,” in *Proc. of the IEEE International Conference on Web Services (ICWS)*, pp. 33-40, 2017.

[28] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M.R. Lyu, “Tools and benchmarks for automated log parsing,” in *Proc. of the IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 121-130, 2019.

[29] W. Meng, Y. Liu, Y. Huang, S. Zhang, F. Zaiter, B. Chen, and D. Pei, “A semantic-aware representation framework for online log analysis,” in *Proc. of the 29th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1-7, 2020.

[30] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst, “Leveraging existing instrumentation to automatically infer invariant-constrained models,” in *Proc. of the ACM*

SIGSOFT 19th Symposium and the 13th European conference on Foundations of software engineering, pp. 267-277, 2011.

[31] Y. Zuo, Y. Wu, G. Min, C. Huang, and K. Pei, “An intelligent anomaly detection scheme for micro-services architectures with temporal and spatial data analysis,” *IEEE Transactions on Cognitive Communications and Networking*, *6*(2), pp.548-561, 2020.

[32] T. Tang, T. Li, and C.S. Perng, “LogSig: Generating system events from raw textual logs,” in *Proc. of the ACM 20th International Conference on Information and Knowledge Management*, pp. 785-794, 2011.

[33] M. Mizutani, “Incremental mining of system log format,” in Proc. of the *IEEE International Conference on Services Computing*, pp. 595-602, 2013.

[34] S. S. Murtaza, W. Khreich, A. Hamou-Lhadj, S. Gagnon, 2 “A Trace Abstraction Approach for Host-based Anomaly Detection,” *in Proc. of the 8th IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA'15),* 2015.

[35] A.A. Makanju, A.N. Zincir-Heywood, and E.E. Milios, “Clustering event logs using iterative partitioning,” in *Proc. of the ACM SIGKDD 15th International Conference on Knowledge Discovery and Data Mining*, pp. 1255-1264, 2009.

[36] A.W. Biermann, and J.A. Feldman, “On the synthesis of finite-state machines from samples of their behavior,” *IEEE Transactions on Computers*, *100*(6), pp.592-597, 1972.

[37] I. Beschastnikh, Y. Brun, M.D. Ernst, and A. Krishnamurthy, “Inferring models of concurrent systems from logs of their behavior with CSight,” in *Proc. of the 36th International Conference on Software Engineering*, pp. 468-479, 2014.

[38] X. Zhang, Y. Xu, Q. Lin, B. Qiao, H. Zhang, Y. Dang, C. Xie, X. Yang, Q. Cheng, Z. Li, and J. Chen, “Robust log-based anomaly detection on unstable log data,” in *Proc. of the 27th*

ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 807-817, 2019.

[39] J.G. Lou, Q. Fu, S. Yang, J. Li, and B. Wu, “Mining program workflow from interleaved traces,” in *Proc. of the ACM SIGKDD 16th International Conference on Knowledge Discovery and Data Mining*, pp. 613-622, 2010.

[40] A. Hamou-Lhadj, and T. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", in *Proc. of the 10th IEEE International Workshop on Program Comprehension (ICPC)*, pp. 159-168, 2002.

[41] H Pirzadeh, S Shanian, A Hamou-Lhadj, L Alawneh, A Shafiee, “Stratified sampling of execution traces: Execution phases serving as strata,” *Science of Computer Programming,* 78 (8), pp. 1099-1118. 2013.

[42] H. Pirzadeh, A Hamou-Lhadj, M Shah, “Exploiting text mining techniques in the analysis of execution traces,” in *Proc. of IEEE 27th International Conference on Software Maintenance (ICSM),* pp. 223-232, 2011.

Appendix A

A.1. Jiang's et al. n-gram Extraction Algorithm [21]

```
Input: a set of unique traces
Output: a set of varied-length n-grams

C1 = {the set of single components $c_1^i$ with $f(c_1^i) > 0$}
$k = 1$
do
   for each two elements $c_k^i$, $c_k^j$ from the set $C_k$,
      if the last $k-1$ component sequence of $c_k^i$ equals
         the first $k-1$ component sequence of $c_k^j$,
         then generate a new sequence
              $s = c_k^i$ plus the last component of $c_k^j$;
              count f(s), the number of times that $s$ appears
              in the trace data;
              if $f(s) > \alpha . \min(f(c_k^i), f(c_k^j))$,
                 then put s into the set $C_{k+1}$
   $k = k + 1$
while $C_k$ is not empty
return all $C_j$, for $1 \leq j \leq k-1$
```

A.2. Jiang's et al Automata Construction Algorithm [21]

```
Input: the set of unique traces and the sets of n-grams
Output: the matrix E(automaton)

Set $E[m][n] = 0$ for any two n-grams $m, n$
For each trace $T$
   Set $k = L$ and $l = T's$ length
   do
      for each $k$-gram $c_k^i$ selected from $C_k$ according to
      the sorted order (with the most frequent one first),

         search and replace all $c_k^i$ in T with the assigned state
         number;

         if the length of the replaced part equals $l$
            then break from the inner loop.
      $k = k - 1$
   while the length of the replaced part $\neq l$ and $k \geq 1$.

   from left to right, set $E[m][n] = 1$ if n-gram n follows
   another n-gram contiguously in the trace $T$

remove the unused n-grams/states from $E$
return the matrix $E$
```

A.3. Jiang's et al. Anomaly Detection Algorithm [21]

```
Input: the automaton and the new trace $T$
Output: true (normal) or false (abnormal)

set $R$=true, $l = T'$s length, and $m = 0$
for each n-grams $c_a^i \in C_a$ selected according to
rules 1 and 2,
   search and replace all $c_a^i$ in $T$ with the state number;
   $m = m + 1$;
   if the length of the replaced part equal to $l$,
      then break the loop;
   else if $m == N_a$
      then $R$=false.
if $R == true$, then
   from left to right, compare each state transition of $T$
   against the automaton;
   if any transition not found in the automaton,
      then $R$ = false.
return $R$
```