TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technology

Department of Software Science

ITC70LT

Orkhan Gasimov 194241 IVCM

Compare and Contrast Analysis of Log Parsing Algorithms: Perspective of Efficiency and Pattern Detection Rate

Master Thesis

Supervisor: Risto Vaarandi, PhD

Co-Supervisor: Mauno Pihelgas, PhD

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Orkhan Gasimov

Abstract

Over the last couple years, many researchers have been working to find solutions for the challenges related to log mining process and the associated log parsing algorithms. In addition, there have been many studies, the main focus of which were the comparison of these log mining/parsing algorithms with each other to find out the most effective ones that were also producing much more accurate outputs. However, during the analysis of the past academic research papers related to the log parsing algorithms and their comparisons, the author of the thesis found out that too little attention was paid to the efficiency, in which the processing speed of these log mining algorithms would be compared to each other so that to find out how much CPU time and memory resources these algorithms consume.

Consequently, the focus area of this thesis was the compare and contrast analysis of log parsing algorithms, and measurement of their efficiency and pattern detection rates. The efficiency was defined as the processing speed of log mining algorithms, in other words, total CPU time and memory consumptions, likewise, the pattern detection rate was described as the quality of the parsed log messages by the log parsing algorithms.

One of the key factors, which makes this thesis different than other studies, is that while executing the log mining algorithms to parse the log messages, the author of the thesis contacted the authors of the tested log parsing algorithms to find out their optimal input values that would help the algorithms to identify rarely occurring log lines, thus, generate output with much higher quality in a more efficient way. The second key factor, which makes this thesis different than other studies, is that the author conducted the memory consumption analysis of log parsing algorithms, which was not tested or analyzed in the previous similar research studies.

Table of Contents

List of Tables

Chapter 1. Introduction

The role of logs, which are, in simple terms, collections of records about the events occurring in the software systems, in their applications and processes, has increased significantly over the current century. There has been an obvious transition of traditional services into digitalized ones during the last decade, which can be seen from the increase in the usage of the Internet, usage of smart devices, application of IoT (Internet of Thing) devices almost in every field that surrounds people, etc. In other words, these developments have changed almost every aspect of people's daily lives. For instance, the way a person contacts with his friends, socializes with people, does his daily purchases and so on [1]. In order to provide better and secure digital services for people, detect anomalies in the case of a system failure and identify the user behavior, developers and engineers are collecting detailed records regarding the system runtime information in the form of log files.

Today, however, there still exist certain challenges in the analysis of log data. One of the challenges is that the size of logs continues to grow since modern software systems generate a huge volume of them. As a result, the manual analysis of the log data by engineers becomes impractical. The second challenge of the log analysis is that most of them are unstructured and just raw textual data because developers usually write unformatted log messages in the source code of their applications [2]. Consequently, there is a need to parse or transform logs into system events in order to be useful for engineers to provide maintenance of services. Finally, after being parsed, there is a need for logs to be summarized into certain patterns from the events [3].

Over the last years, many researchers have been trying to resolve these issues by developing data mining algorithms for the analysis of unstructured log data [4], by making the overall log analysis process more automatic rather than manual [5, 6, 7]. Additionally, some of the research papers [8, 17, 18] were mainly focused on the comparison of these log mining/parsing methods to identify which one of them is more effective and provides more accurate results.

Most of these research papers, however, were generally concentrated on the evaluation criteria such as the accuracy, where they measured the ability of log mining method to identify variable and constant parts of log messages, the robustness, where they analyzed the accuracy of results generated by these tools in the case of logs with different sizes obtained from various systems, and the usability, where it was identified how easy it is to use and apply these log parsing tools. On the contrary, too little attention was paid to the efficiency, where scientists would measure the processing speed of these log mining algorithms, how much time and resources they consume, etc. Moreover, the ones, which focus on the analysis of the performance of log parsing algorithms, have not been thorough and/or fair enough. This is also the case for a recent paper from the University of Hong Kong that claims to address the shortcomings of previous works and offer a detailed performance comparison [8]. For example, the labeled datasets used for the accuracy measuring experiments [8] are not available for the download and use for other research studies. Furthermore, initially, it was claimed in the paper that 16 different datasets were used during the experiments, where some of them were very large. Nevertheless, during the conducted performance testing experiments, the authors utilized just 3 quite specific datasets, which do not represent frequently used real-world log types. Another drawback of the study is that the datasets used in the experiments were preprocessed, which corrupts the results and an adequate picture about what the performance of various algorithms can be on real log files. What is more important, however, is that some log mining tools were tested with badly chosen input parameter settings, which led to the poor performance of these tools and unfair comparison [19]. Finally, in the research study [31] by by Copstein et al., it was mentioned that the authors by focusing on the work of study [8], tried to replicate the conducted experiments, where they used the implementation of tools and annotated dataset, which were provided publicly by the original study. During their replication study, however, while analyzing Android logs, the authors found out that one of the log parsing algorithms did not manage to generate the similar results described in the study [8]. In addition, at the end of the replication experiments, the authors confirmed that some of the obtained outcomes were significantly different than the original study, in other words, results demonstrated approximately more than 10 percentage points of variance over the original experiments.

Consequently, the objective of the author in this thesis, firstly, was to conduct the comparative analysis of currently available log parsing algorithms and obtain fair and clear results regarding the efficiency of them, where the efficiency is defined as the processing speed of log mining algorithms, in other words, total CPU time and memory consumptions while parsing log datasets of various sizes from different systems. Secondly, to find out the quality of the parsed log messages, where the quality is defined as the pattern detection rate of the log parsing algorithms.

In comparison with other similar studies about the log parsing algorithms, this thesis is different since the author of it contacted the authors of the tested log mining algorithms in order to find out the optimal input values while running the log parsing algorithms, which would help, firstly, to identify rarely occurring log message lines, secondly, produce results with much higher quality in a more efficient way. That allows for avoiding unfair comparison of different tools due to poor input parameter settings (for example, such mistakes have been made in [8]). Furthermore, another important distinction of this thesis paper is that the author conducted the memory consumption analysis of log parsing algorithms, which was not tested or analyzed in the previous studies like the paper [8]. Finally, unlike previous studies (for example, [8]), experiments for measuring efficiency are conducted on large log files from production environments, which have not been preprocessed in any way. As a result, the main contributions of this thesis paper are the identification of a log parsing algorithm or algorithms that generated results with much higher quality and achieved best pattern detection rates and demonstrated best performance and efficiency outcomes during the conducted experiments, likewise, the provision of the support for engineers to parse raw log data files into structured ones by saving CPU time and memory resources of their devices.

The thesis consists of four chapters. The chapter 1 gives a brief overview of the importance of log files for the computer systems and services that people benefit in their daily lives, discusses some existing challenges related to the analysis of log data and what measures have been taken to resolve these issues, likewise, the focus area of the study is mentioned in this chapter as well, which is the compare and contrast analysis of log parsing algorithms mainly from the perspective of the efficiency and the pattern detection rates. The literature review and the methodology used in the research are discussed in the chapter 2. In the chapter 3, the author gives detailed information about the conducted experiments, such as what log parsing algorithms have been used, which datasets have been chosen, etc. Additionally, the results of the experiments, in other words, the performance of log mining algorithms are also discussed in this chapter. This chapter answers questions such as how much CPU time it takes for a particular algorithm to parse a raw log data, what the memory consumption is, how accurate are results and so on. In the fourth and the final chapter of the paper, the author provides a summary of the research study, revisits key points, and makes recommendations on which automatic log mining tools should be used to parse raw log files in order to obtain more accurate results in a more efficient manner.

Chapter 2. Literature Review and Research Methodology

2.1 Research Methodology

The research methods that the author used in this thesis are:

- Analysis of the past academic research documents related to the automatic log parsing algorithms. The goal here was to identify which log mining algorithms currently exist in the market, how effective they are, what their strengths and weaknesses are, etc.
- Analysis of the past academic research papers related to the comparison of performances of log mining algorithms with each other while parsing raw log files. The objective that the author wanted to accomplish here was to understand how those log parsing algorithms were compared to each other, which evaluation criteria was considered, how experiments were conducted, what kind of datasets were used and so on.
- Conducting an experiment for comparative analysis of log mining algorithms specifically from the perspective of their efficiency and pattern detection rate, where, as it was stated earlier in the paper, the efficiency is defined as the processing speed of log parsing algorithms, in other words, how much CPU time and memory these algorithms consume while parsing raw log files of different size from various computer systems.

The documents referenced in the literature review part of this thesis were mainly found from the website called scopus.com. For the detailed preparation of the literature review, the following keywords were used: *log mining*, *log parsing*, *log cluster*, *log analysis*, *data mining*, etc. In terms of a search technique, the backward snowballing was used because some documents were not allowed to be downloaded. In other words, after getting the research papers that could be downloaded, the documents mentioned in their references were used since they had more detailed information about the topic of interest. Besides the scopus.com, some research papers were found with the help of Google search engine.

As it was stated earlier, the author conducted an experiment, in which, he compared the efficiency and pattern detection rates of log parsing algorithms with each other along with the quality of produced results. The datasets, that were used for experiments, were generated from the log data collections obtained from TalTech (Tallinn University of Technology) and from NATO

CCDCOE based on the NDA (non-disclosure agreement), which was signed between the respective parties.

2.2 Literature Review

In the research papers described in the section below, the authors are mainly focused on the specific log mining algorithms and their comparative analysis to similar methods from the perspective of parsing of unstructured raw log message files into structured ones. Most of these research studies start by describing the importance of event logs and emphasizes the roles that they play in the management of systems and networks. For instance, the paper [13] begins by highlighting the significance of monitoring of the way large enterprise systems work with the help of log files generated by the corresponding systems. According to the study, the system monitoring via log files plays a crucial role since it helps to ensure that such complex systems are producing expected results.

The remainder of the literature review is organized as follows: in subsections 2.2.1-2.2.7, a number of log mining algorithms are discussed, while subsection 2.2.8 describes comparative studies of different log mining algorithms.

2.2.1 IPLoM

In their paper [5], Makanju, Zincir-Heywood and Milios claim that the manual analysis of log files is becoming challenging day by day since they are increasing in size significantly. Consequently, the paper is concentrated on the automatic analysis of these log files and introduces a log mining algorithm called IPLoM or Iterative Partitioning Log Mining.

According to the study [5], IPLoM is a new clustering algorithm used to obtain event type patterns by mining the event logs, where each detected cluster corresponds to some event pattern. Overall, there are four steps for IPLoM to generate clusters from raw log files. During the first three steps, IPLoM works through hierarchical clustering process, in other words, the algorithm divides a log file into respective clusters. To be more specific, IPLoM starts its work by considering the entire log file as a single cluster. Next, during the first step, the initial cluster is split by assigning log file messages with a different number of words to different clusters. During the second step, each resulting cluster is then divided further in the following way – a word position is identified which has the least number of unique words, and the cluster is split by these words. On the third step, clusters from the previous step are divided by associations between pairs of words. In the final and fourth stage, IPLoM generates cluster descriptions for every leaf partition of the log data file. In other words, by producing these cluster descriptions, the algorithm finds event type patterns.

There are some differences of IPLoM from similar log mining algorithms. For instance, IPLoM is not based on the Apriori algorithm, which is a popular data mining method utilized by some other log mining algorithms [5]. This helps the algorithm to be more computationally efficient during the mining process of longer patterns compared to other algorithms [11]. Additionally, while it is mandatory to use a pattern support threshold for some other similar log mining algorithms, for IPLoM, it is optional, which increases the possibility for IPLoM to find all potential clusters.

The authors of the paper found out that IPLoM consistently outperforms the other similar algorithms such as SLCT [14], Teiresias [23] and Loghound [22] when they were compared during the conducted experiments with seven various event log files.

On the other hand, the authors mentioned that the data files used during the experiments, at the initial stage, were manually labeled. Likewise, based on the partition results of IPLoM, it was discovered that in some situations, it was almost impossible for IPLoM to generate correct cluster descriptions. The authors concluded that it should not be considered as an issue for a human since people are able to manually identify right cluster descriptions. To sum up, even though the algorithm is used as an automatic log analysis algorithm, there is still a need for a human interaction.

In terms of the evaluation criteria used during the analysis, the authors of the study used metrics such as Recall, Precision and F-Measure in order to measure the clustering quality of IPLoM.

2.2.2 Spell

In their study [10], Du and Li state that in previous similar research papers, scientists were mainly investigating log analysis algorithms that were parsing raw log files in an offline mode. Nevertheless, the need to provide online monitoring and processing of log files is increasing day by day. Consequently, the given paper recommends the usage of an online log parsing algorithm, which is called Spell (Streaming Parser for Event Logs), that parses unstructured system event logs into structured log types with the help of a longest common subsequence technique.

In the research paper [10], it is said that let us assume there is a universe of alphabets, for instance, from a-z and from 0-9. Let us also assume that there are two different sets consisting of elements from this given alphabet. If a subsequence is both the subsequence of first and second sets of elements at the same time, then, this subsequence is called a common subsequence of set one and two. Furthermore, for a given input of sets of elements, LCS (longest common subsequence) is defined as the longest of such subsequences. For example, if there are sets of elements {1,3,5,7,9} and {1,5,7,10}, then an LCS will be {1, 5, 7} [10]. As a result, it was decided to utilize an LCS-based technique to extract message types from raw log messages. According to the authors of the paper, the output generated by the log printing statements is considered as a sequence. Static parts cover the most portions of these log printing statements, while the variable parts take only a small one. Therefore, if two log entries are generated by the identical log printing statement, where the only difference is in the parameter values, then, LCS of these sequences is most probable to be the constant in the code, which denotes the message type.

Consequently, the designed algorithm contains a data structure, which is called LCSObject, the main purpose of which is to store parsed LCS sequences and the associated metadata information. Each LCSObject consists of LCSseq, which is the notation used to denote LCS of multiple log message lines. In addition, all currently parsed LCSObjects are stored in a list that is called LCSMap. When Spell receives a new log message line, it is first compared with all LCSseq's contained in LCSObjects of LCSMap. Next, the algorithm decides to either insert this new log line to the existing LCSObject, or compute and generate a new LCSObject, which is then added into LCSMap.

According to the results of the conducted experiments, the authors claim that in the case of large real system logs, Spell in comparison with the offline alternatives such as IPLoM and CLP (clustering-based log parser) demonstrates its supremacy in terms of both effectiveness and efficiency.

For conducting experiments and comparing the performance of log mining algorithms [10], the authors used two real log datasets with various formats.

The first drawback of the study, however, is the overall number of datasets used to make comparisons of log parsing algorithms. Another drawback is that the authors of the paper measured the efficiency only in terms of runtime of log mining algorithms both in the form of a logarithmic scale and in the form of a normal decimal scale. In other words, the authors focused only on the time efficiency of these log parsing algorithms.

2.2.3 Drain

The study by He et al. [2] is similar to the previous one [10] since the authors of the paper start with the introduction that majority of the currently available log parsing algorithms are working based on the offline mode, and, considering the fact that the size and the volume of logs are increasing enormously, the model training process, where the offline log mining algorithms employ all obtained log data after collecting them, consumes a lot of time. As a result, the paper is also focused on the resolution of this issue by proposing an online log parsing algorithm called Drain, which is able to accurately and efficiently parse raw logs in a streaming and timely manner.

Overall, in order to parse raw log files, besides the raw log messages, Drain does not need the source code of applications or any other information. With the help of the fixed depth parse tree technique, Drain can generate log templates from raw log files and divide them into the log groups. When, for instance, Drain receives a new raw log message, the tool starts searching a log group, in other words, leaf node of the tree in accordance with the specially implemented rules that are encoded in the internal nodes of the tree. In case, Drain finds out a suitable log group, the given log message will be compared and matched with the log event, which already existed in the log group. If, however, Drain does not find any suitable log group, then, a new log group associated with the given log message will be generated. Additionally, one of the key features of Drain is the fact that log messages with different number of words are divided between different branches of the tree.

In their paper [2], the authors compared Drain with four similar log mining algorithms, such as LKE [9], IPLoM [5], SHISO [12], and Spell [10] in terms of the accuracy, efficiency, and effectiveness. Overall, Drain was compared to other similar log parsing algorithms based on the five real-world log datasets, which had more than 10 million raw log messages. According to the results, Drain outperformed other algorithms in terms of the accuracy on four datasets, and almost reached better accuracy on the remaining one. During the experiment, Drain also got improvements in its running time, in other words, Drain was running faster than the other algorithms.

The efficiency metric, however, was again limited only to the running time of the algorithm. In other words, the authors of the study measured only the time efficiency, but not the memory consumption.

2.2.4 AEL (Abstracting Execution Logs)

In their paper [13], Jiang et al. stated that even though the monitoring of computer and network systems via log message files is an integral part of providing better and secure services, the methods that are currently used for monitoring, for instance, the code instrumentation and profiling consume a lot of resources to produce correct results. Another drawback of such techniques is that specialists need to get the source code for a particular system or to contact to system experts.

As a result, in their study, the authors recommend the usage of execution logs to observe the way applications are working. Every instance of a certain execution event ends up in a different log line because log lines generally consist of static/constant and dynamic parts that varies each time. Consequently, since these execution logs were initially not designed for monitoring tasks, the authors of the study propose a technique, where log lines will be abstracted to a set of various execution events. In the paper, it is said that the proposed approach is similar to a rule-based approach. Nevertheless, for this technique to work, there is a little need of effort and the system knowledge. Instead of encoding rules to identify specific execution events, the method described in [13] utilizes a few common heuristics to differentiate static and dynamic parts in log messages. At the end, log messages, which have similar static portions are categorized together for the abstraction of log messages to execution events. To be more specific, the algorithm mainly consists of three steps, which are **Anonymize**, **Tokenize** and **Categorize**. During the first stage, Anonymize, the algorithm utilizes heuristics to identify tokens in log message lines, which are related to dynamic parts. After being identified, the tokens are replaced by *$v* symbol that is used to denote generic tokens. Two main heuristics that are used to recognize dynamic parts are [13]:

1. *Assignment pairs like "word=value"*
2. *Phrases like "is[are|was|were] value"*

Next, during the Tokenize step, based on the number of words and variable parts, the anonymized log messages are divided into the various groups, which are also referred as ***bins***. Finally, in the case of Categorize step, the algorithm makes comparison of log lines in every bin and abstracts these log messages to the respective execution events.

In the study, the authors, firstly, mention that this approach does not work based on some specific strict requirements regarding the format of log files. Moreover, according to [13], the authors conducted some experiments, where the target system was a large enterprise application, in order to compare this abstraction technique against other log mining algorithms, namely SLCT [14]. Based on the results of experiments, the authors claim that the abstraction technique outperforms SLCT while parsing raw log data.

The drawback of the paper is that first of all, instead of log files obtained from various domains, the case study was performed on a single application, which is also considered as a drawback by the authors. Secondly, according to the paper, this approach requires some human interaction, in other words, engineers need to go through some log lines, analyze them and define the anonymization rules. Finally, in terms of time or memory consumption of this method while parsing raw log files, nothing is mentioned in the corresponding paper since it only focuses on the accuracy of the results obtained by the log abstraction technique.

2.2.5 LogSig

In the paper [15], Tang, Li and Perng, first, describe the existing challenges in the analysis of log files such as a huge volume of log data generated day by day by the modern computing systems, the need for specialists, that use these log message files, to have domain knowledge in order to understand the behavior of their computer systems, etc. In addition, according to the study, another challenge related to the log analysis is that many of these log files are unstructured and in a raw textual format, which makes it difficult for automated log analysis while generating system events from these raw textual logs.

It is known that usually, these log messages are not very long, nevertheless, may contain a large vocabulary, in other words, consist of completely different words for each log line. The authors of the study claim that the fact that log files can have various words for each log line generally leads to the poor performance when traditional text clustering methods have been used to parse the log data. Furthermore, other log mining techniques have different restrictions and limitations and are suitable for specific systems log files.

In the study [15], the authors propose a technique called logSig, which is a message signature-based algorithm used to obtain system events by analyzing raw textual log data. According to the paper, overall, logSig algorithm involves three main steps in order to produce results. The first step is the division of the log message lines into several pairs of terms. The second one is to obtain *k* groups of log message lines, where *k* is specified by the user, by utilizing local search strategy, in which every group would have common pairs as many as possible. Finally, the algorithm generates message signatures by investigating common pairs found in the second step for every message group. In other words, this technique is using the common subsequence information of raw log messages in order to categorize them and describe the newly created events.

In the paper, the authors also conducted experiments by taking into account five real system log files such as Apache, FileZilla, Hadoop etc., to make a comparison of logSig against other alternative algorithms. According to the study, logSig outperforms other log mining algorithms in terms of overall performance.

One of the drawbacks of the paper, however, is that the authors measured the performance of algorithms mainly from the perspective of the quality of results obtained by the algorithm and the scalability. The quality of results was measured with the help of F-measure (F1 score), which is a traditional metric to evaluate how accurate the results generated by the algorithm are by combining the precision and recall. The scalability, on the other hand, was measured by considering only the average running time of log mining algorithms while parsing raw log files of different sizes.

2.2.6 LenMa

According to the study by Shima [16], the analysis of system log messages mainly consists of two parts. First part of the log analysis is defined as a message template generation, while the second one is the possibility to extract something interesting and useful from the messages that are categorized by the generated templates. Therefore, the generation of accurate and correct templates plays a significant role in achieving better and precise log analysis results.

In his paper, in order for specialists to generate better and accurate templates from log messages, the author suggests the usage of a technique, which is a classification methodology that utilizes the length of words that appear in each log message. In other words, in the study [16], it is mentioned that currently, many existing log mining methods while creating templates tend to characterize words of messages by analyzing, for instance, their character types, ratios etc. On the other hand, the proposed classification method focuses more on the length of each word that appears in a log message, which is then used for clustering the messages. Overall, upon receiving a new log message line, this technique performs five steps to produce results:

1. Generation of word length and word vectors from the new log message
2. Calculation of a similarity score between this new log message and every existing cluster with the same number of words
3. Creation of a new cluster with this new log message if the similarity score of the existing clusters is not larger than the threshold
4. Update of the most similar cluster with this new log message using the algorithms provided in [16] if step 3 is false
5. Return of the cluster

Additionally, considering the fact that the number of log messages generated by system components is increasing day by day and that the classification technique mentioned in the research study does not need to perform two-pass analysis to create template messages, the author of the paper suggests the usage of this methodology for online template generation as well.

Even though the recommended method can be useful for certain cases to parse raw log messages obtained from various systems, in terms of its performance, nothing is said in the research study [16]. In other words, the drawback of the paper is that there is no focus on specific aspects of the given methodology such as its time or memory consumption, or the overall performance of the algorithm.

2.2.7 LogCluster

According to the study [4] by R. Vaarandi and M. Pihelgas, in order to make the management of log files an easy process, many researchers have proposed in their papers the usage of various log mining algorithms, which are intended to generate event patterns from event logs. These event patterns can be useful in various areas such as the creation of event correlation rules, detection of network anomalies or system faults, monitoring and creation of reports about network traffic patterns, automated generation of intrusion detection system (IDS) alarm classifiers, etc.

The authors of the study state that previously proposed log parsing algorithms have been mainly based on the data clustering techniques, where there was an assumption that every event in the log file is described by a single line, and every line pattern shows a group of identical events. However, in their paper, the authors suggest a data clustering algorithm, which is called LogCluster, in order to produce event patterns from raw unstructured log files. LogCluster, firstly, starts by making a pass over the log file for generating a dictionary of all words together with their occurrence counts, so that frequent words can be detected (words that occur in at least ***S*** log file lines, where S is a support threshold provided by the user). Then, LogCluster makes another pass over the log file for creating the cluster candidates from the combinations of frequent words. As a final step, the candidates, that occur at least in ***S*** log file lines, are selected as clusters. Since the dictionary of words can consume a lot of memory for very large log files that contain millions of words, LogCluster can be configured to make an extra pass over the log file for creating a word sketch (a compact vector of ***N*** counters), which allows for filtering out many infrequent words and saving large amounts of memory.

In the study, the authors also conducted experiments with large event logs, where they evaluated the performance of LogCluster and compared it with other similar algorithms.

2.2.8 Other Similar Research Studies

According to the research study paper [8] by Zhu, He, Liu et al., the traditional manual way to analyze log files becomes impractical since the overall number of log messages generated by modern software systems is increasing day by day. Since detailed log files are used to monitor the behavior of computer systems and to identify any anomalies, these log data play a very crucial role for specialists in order to develop and maintain the workflow of computer systems.

In the paper, it is highlighted that for the analysis purposes, since log messages are usually in an unstructured form, the first fundamental step is the ability to parse or convert them into structured data. As a result, according to the research study, there have been many automated log parsing algorithms used for the last couple years, which have also been studied and analyzed both in the industry and the academia.

In order to better understand the way these log mining algorithms are working, the authors of the paper [8] provide a comprehensive evaluation study. In other words, they conduct experiments, where the performance of 13 log parsers is observed on an overall of 16 log datasets obtained from various systems such as supercomputers, mobile systems, operating systems, server applications, etc. The metrics that have been utilized in the study are the accuracy, robustness, and efficiency of log mining algorithms while parsing raw log data files.

Even though the focus area of the research paper is directly related to the analysis of the performance of log parsing algorithms, the results obtained during the experiments are not thorough and/or fair enough. According to the study [19] by Mauno Pihelgas, the fairness of these experiments is questionable because some tools have been tested with badly chosen input parameter settings, resulting in poor performance of these algorithms. In addition, the authors of the paper [4] also claim that the input parameters of the algorithm, LogCluster, which was executed during the experiments, were badly chosen. Consequently, in order to achieve better and fair results, LogCluster had to be utilized during these experiments with its optimal input parameters since it would help to reveal the true capabilities of the algorithm. One more drawback of the study [8] is that the labeled datasets that were obtained from different systems and used for measuring the accuracy and during the experiments, are not available to be downloaded and utilized for other research studies. In addition, even though initially in the paper, it was claimed that the total number of distinct datasets utilized during the experiments was 16, while measuring the performance of algorithms, for instance, the authors utilized only 3 too specific datasets such as a supercomputer log, a Hadoop file server log, and Android log. This certainly does not demonstrate real-life scenarios, in which, it is common to encounter log types like Linux or Windows logs. Another drawback of the study is that the authors preprocessed the datasets by, for example, replacing IP addresses with special tags, which affected the produced outcomes in a great manner and corrupted the actual performance results of tested algorithms. In conclusion, according to the research study [31] by by Copstein et al., where the authors replicated the experiments conducted in the study [8] by using same implementation of tools and annotated datasets, it was found out that while parsing the Android logs, one of the tested log mining algorithms, LogSig, could not to produce the same outcomes as were described in the original study. Additionally, after completing the replication experiments, the authors concluded that in comparison with the original paper, there was about more than 10 percentage points of difference in the obtained results.

One of the similar research studies is the paper [17] by El-Masri et al., where the authors provide a systematic literature review regarding the existing automated log abstraction techniques. Firstly, the authors of the study start the paper by describing the challenges and problems related to the log analysis and log parsing. They also mention that in order to solve existing issues, software engineers can take an advantage of a wide range of ALATs (Automated Log Abstraction Techniques), which would help them to decrease the overall amount of log data to process. All of these methods have a different technique and different log-abstraction algorithms to accomplish various tasks such as information security related issues, performance optimization while parsing raw log files, anomaly detection, etc.

On the other hand, according to the study, there still exists a huge gap between the academia and the industry. For instance, majority of specialists are not well-informed regarding all of the existing ALATs, that have been developed in the academia. Naturally, there is always a chance for a certain engineer to research and identify a particular log mining tool and what kind of an algorithm has been utilized in that tool. However, the study highlights that in general, software engineers do not have enough time and resources to allocate to the study and research of such algorithms and understand the characteristics of each of them.

Consequently, the goal of the research study [17] is to reduce this existing gap by providing a systematic literature review (SLR), which would help to raise the awareness of many software engineers about the existing ALATs. Firstly, the authors of the paper categorize ALATs' specifications, that were found out during the SLR, into seven most desired quality aspects and generates a quality model to evaluate these log abstraction algorithms. Then, based on the created quality model, there is a demonstration of the comparison of 17 log mining algorithms. According to the authors of the study, this comparison of algorithms helps to clarify and reduce the research gaps and to make recommendations for specialists on which log parsing algorithms to utilize.

The drawback of the study is that the authors of it perform only a qualitative comparison of log mining algorithms while parsing raw log files. In other words, no performance experiments with certain log data files have been conducted.

Another similar research study, which makes the comparison of log mining algorithms with each other is the paper [18] by Landauer et al. The authors of the paper start by describing the importance of log files, the role they play and why they are crucial for specialists by emphasizing the fact that these files contain significant information about nearly all events that occur in systems such as databases, web servers, operating systems, firewalls, etc. Since the most of log messages are in a textual and human-readable format, and have, for instance, a time stamp that denotes the time this log message was generated, large organizations and companies can benefit from them for the forensic analysis or investigation cases of the long-term data.

According to the study [18], the log messages, which show system faults and that are used for the forensic analysis, provides system administrators the opportunity to analyze and trace back the root causes of certain problems. Likewise, with the help of log files, specialists can, for example, restore the system, which had problems, to a non-faulty state, recover important data, stop the loss of information, test locally the scenario that caused problems in the system, etc.

On the other hand, however, the authors of the paper claim that there is a major challenge with the forensic log analysis – log messages that are useful for the forensic investigations can only be detected in hindsight. Moreover, this task is very time and resource consuming since in order to successfully accomplish it, specialists need to have domain knowledge regarding the system in use. Consequently, according to the research study, in order to solve the above-mentioned issue, specialists are moving the analysis techniques from a purely forensic to a proactive. In other words, specialists are monitoring log messages in a real-time online fault detection mode.

In the study, it is also mentioned that the manual forensic analysis of log files by humans is becoming impossible day by day since the generated log messages have huge sizes and volumes. In terms of automatic log parsing algorithms, on the other hand, today, there exist various methods that automatically process the log message lines and extract interesting patterns from them. These methods are utilizing different algorithms to parse the log files and were mainly developed for the specific scenarios.

In the paper [18], the authors conduct a comprehensive survey, where they get information regarding various automatic log mining algorithms found in scientific literature, which are specifically designed for applications used in the field of cyber security and compares them to each other.

The drawback of the study, again, is that the authors of it make a comparison of log mining algorithms in a qualitative way. There have been no performance related experiments conducted in the paper.

Finally, in the research paper [31] by Copstein et al., the authors analyzed the current state-of-the-art and open-source log parsing and abstraction techniques in order to find out how they deal with the security-related log messages. In their study, the authors also conducted experiments to measure the pattern detection quality and performance of log mining algorithms, where they utilized the log files, which included messages from DNS, HTTP, FTP, Firewall, SNORT IDS, SSH, etc. On the other hand, the tested datasets contained only 2000 log message lines, which affected the runtimes of log parsing algorithms significantly during the performance tests by achieving very fast results. Therefore, the study does not provide insights what would be the performance of algorithms on larger log files that are a typical target for log mining algorithms.

Chapter 3. Experiments

In this chapter, the author, first, gives information about the experimental setup. In other words, the author provides details of the hardware and OS (Operating System) used to conduct the experiments, information about the log mining algorithms that were chosen for the comparative analysis, the detailed information about the datasets that were utilized for the experiments, and what kind of metrics and tools were used for evaluation purposes. Then, the author of this thesis paper describes the conducted experiments and the steps that were taken, and at the end, provides the details of the achieved results.

3.1 Experimental Setup

In order to conduct experiments, where log parsing algorithms would be compared to each other, the author of the paper used a dedicated physical Linux server. The main reason why the author preferred to use physical machine for experiments rather than a virtualized environment is to reduce a possibility of occurrence of issues and problems specific to such infrastructures. To be more specific, the utilized physical machine had more computing power and memory than a typical virtual machine, which enabled the author to run experiments without any technical issues.

Below, in the **Table 1**, the author provides technical characteristics of this physical server:

Operating System (**OS**)	Rocky Linux release 8.4 (Green Obsidian)
Physical memory (also known as random-access memory (**RAM**))	64 GB
Disk Type	Samsung SSD 860 EVO 250GB, RVT02B6Q, max UDMA/133
CPU model/make	Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz
CPU sockets / CPUs	2

Cores per CPU	6
Threads per CPU	12
Total CPU Cores	12
Total CPU Threads	24

Table 1. Technical Characteristics of Linux Server Used for Experiments

3.2 Log Mining Tools Used for Comparative Analysis

For comparative analysis, overall, the author chose seven log mining algorithms to be used to parse unstructured raw log messages. In the **Table 2**, the author of the thesis provides the list of these log parsing algorithms:

Log Parsing Tool 1	IPLoM (Iterative Partitioning Log Mining) [5]
Log Parsing Tool 2	Spell [10]
Log Parsing Tool 3	Drain [2]
Log Parsing Tool 4	AEL (Abstracting Execution Logs) [13]
Log Parsing Tool 5	LogSig [15]
Log Parsing Tool 6	LenMa [16]
Log Parsing Tool 7	LogCluster [4]

Table 2. Selected Log Mining Algorithms

Since, as of the moment of writing this thesis paper, there were no other publicly available implementations of most of the above-mentioned log mining algorithms, the author utilized the tool implementations from the ***github*** repository of University of Hong Kong [8] paper. In other words, besides the log mining tools ***Drain3*** and ***LogCluster***, which were obtained from their official repositories [24] (*release 0.9.7*) and [25] (*version 0.10*) respectively, other tools were downloaded/copied from [26] for ***IPLoM***, [27] for ***Spell***, [28] for ***AEL***, [29] for ***LogSig*** and [30] for ***LenMa***. It is also worth to note that since the new version of ***Drain***, which is ***Drain3***, was available to be downloaded from [24], instead of the old version of the tool, the author decided to conduct experiments with the newer one.

In terms of the reason behind the log parsing algorithms selection, the author decided to experiment with the best and worst algorithms from the paper [8] with the main focus on the ones released since 2015. To be more specific, according to the results of the experiments conducted in the research study by University of Hong Kong [8], the log parsing algorithms that produced the most accurate results are ***LenMa*** [16], ***Spell*** [10], ***AEL*** [13], ***IPLoM*** [5] and ***Drain*** [2]. While ***LogCluster*** [4] has the average level of accuracy for the produced outcome, ***LogSig*** [15] has the worst performance in terms of the quality of parsed log messages. Since recent studies have questioned the results in [8] for ***LogCluster*** and ***LogSig*** [19, 31], the author of the thesis decided to include these algorithms in the tests. As a result, the author performed experiments with the help of above-mentioned log mining algorithms in order to find out how algorithms would be different compared to each other by the pattern detection rate and efficiency.

3.3 Datasets Used for Experiments

As it was stated earlier in the thesis, after signing the NDA (non-disclosure agreement) with the corresponding parties, the author obtained log data from two various organizations.

One of these log data was obtained from TalTech (Tallinn University of Technology). The size of this log data is 11GB, and it consists of five log files from Linux Servers, which are in an unstructured raw syslog format. The other log data from TalTech is a log file with Suricata IDS alert messages (97MB).

On the other hand, from NATO CCDCOE, the author got one separate log file and two large data collections consisting of several log files. One of these collections has the log data from Suricata systems and involves 188 compressed log files in it. In terms of the size, Suricata log file collection is 3.7GB. Unlike the Suricata log data from TTU, the NATO CCDCOE log data is not only containing IDS alert messages, but also network traffic records for network flows and various common application layer protocols. The second collection involves compressed log files, which were obtained from Windows operating systems. This collection has 202 log files and is 3.4GB. Finally, as it was stated above, the author obtained an individual log file, which contains syslog messages gathered from XS19 exercise. To be more specific, this file has 27365365 log message lines from the central syslog server of XS19. These events were monitored and collected from 54 distinct hosts during the period of 10 days. In addition, the overall log file size is 4.6GB.

To sum up, the experiments for this thesis paper were conducted with the help of the log messages obtained from the above-mentioned datasets, which are TTU (11GB from Linux Servers and 97MB from Suricata IDS) and NATO CCDCOE (7.1 GB compressed files from Suricata and Windows systems, and 4.6GB from syslog server of XS19).

3.4 Metrics/Evaluation Criteria of Experiments

3.4.1 Performance Testing Setup

Previously in this thesis, it was mentioned that the objective of the author was to perform the comparative analysis of log mining algorithms and eventually, get fair and clear results. Likewise, it was specified that the main focus during these experiments would be on the pattern detection rate and the efficiency of log parsing algorithms. In other words, one of the criteria to evaluate results achieved at the end of experiments is the efficiency of log mining algorithms, where the efficiency, in this thesis paper, is defined as the processing speed of these algorithms, such as CPU time and memory consumption, while parsing raw unstructured log messages. In order to measure the efficiency of log parsing algorithms, the author used GNU Time utility [20]. The GNU 'time' command is used to run another program, and then, to display the information about the system resources, which were consumed by that program. During the experiments, the author used the following command along with a particular log mining tool to parse raw log messages and obtain system resource consumption information:

```
/usr/bin/time -f 'memory %M KB, kernelCPU %S, userCPU %U' <log
mining tool> <log file to parse>
```

In terms of the datasets, that were utilized to conduct performance testing experiments, the author used four data files with various sizes. The first dataset was derived from Suricata log file collection obtained from NATO CCDCOE and consisted of HTTP Apache log messages, thus, was called **http-apacheformat.log**. Compared to other utilized datasets, **http-apacheformat.log** was a relatively medium-sized one because it consisted of 95457 log message lines and had a size of 21MB. The second dataset, which was one of the large-sized datasets, was called **windows-text.log**. This dataset was generated from the second log file collection received from NATO CCDCOE, which had logs of Windows operating systems. In terms of the number of the log messages, **windows-text.log** had 3551025 lines and had a size of 3.2GB. The third dataset, which was tested during the experiments and was considered as a large-sized data file as well, was one of the five log files received from TalTech (Tallinn University of Technology). This dataset, called **logsrv.log.2**, had 17015421 syslog message lines and had a size of 2.5GB. Finally, the author utilized the last large-sized data file, which was **xs19-syslog.log** dataset. This dataset contained 27365365 syslog events from XS19 exercise and had a size of 4.6GB.

3.4.2 Quality Testing Setup

Naturally, obtaining results about which log mining algorithms are more efficient compared to other ones is not sufficient for the experiments. In other words, even if the performance of a particular log parsing algorithm is much better than others, if quality of produced outcome is very poor, then, this particular log mining algorithm cannot be considered as a valid option to parse unstructured raw log messages in a real-world environment. Consequently, in this thesis paper, the second metric used for the evaluation of the produced results is the output quality for the parsed log files. The quality is measured in terms of the pattern detection rate – if the log file is known to contain N patterns, and the output of the tool contains K patterns out of these N patterns ($K \leq N$), then the pattern detection rate is defined as $K / N * 100$ %.

In order to estimate the quality of generated outcomes, four specific small and medium-sized datasets were used during the experiments. All these four datasets were derived from TalTech Linux and Suricata log data. Three of these datasets, that were used for quality testing, were created by extracting *sshd*, *su* and *sudo* syslog log messages from Linux log files of TTU log data. Henceforth, these datasets are called *sshd*, *su* and *sudo*. The fourth dataset, on the other hand, is the Suricata log file from TalTech, which consists of Suricata IDS alert log messages, that are also in a syslog format. In summary, in terms of the number of log message lines, *sudo* dataset had 815, *su* contained 1496, *sshd* - 420104 log messages, while *suricata* consisted of 499805 log message lines and had 97MB of size.

After the creation of the above-mentioned two small-sized (*su* and *sudo*) and the two medium-sized (*sshd* and *suricata*) datasets, the author generated a file, where he manually identified and combined the log message patterns/clusters/templates that were expected to be found from *su, sudo, sshd* and *suricata* datasets while being parsed by log mining tools. The manually identified patterns, in total, consisted of 5 **sudo**, 12 **su** and 34 **sshd** patterns. Additionally, since *suricata* dataset contained IDS alerts that were all sharing the same format, the tools were expected to detect just 1 generic pattern that would represent a message template for all IDS alerts. The main goal in the creation of a file with manually identified patterns was to make a comparison of the results produced by a certain log parsing tool against the patterns that were manually generated by the author and calculate the pattern detection rate. According to the author of this thesis paper, the pattern detection rate is calculated based on the following rules:

1. If ***P*** is a manually created pattern, ***P*** is considered as found if ***P*** exists in the output of the log mining algorithm.
2. A certain log mining tool, like ***LogCluster***, can generate a pattern, where instead of wildcards, lists of possible values are shown. For instance, for the following pattern, "`(alice|bob) : unable to resolve host *`", the list of all possible variable values, such as (*alice|bob*) are provided. In the case of such patterns, the value lists are treated as wildcards.
3. For some manually created patterns, the log file contained messages, where for some wildcard in the manually created pattern, the same constant appeared in all messages. For example, suppose that for the manually created pattern "`* : unable to resolve host *`", the log file contains the following three messages: "`alice : unable to resolve host 10.1.1.1`", "`alice : unable to resolve host 10.1.1.2`" and "`alice : unable to resolve host 10.1.1.3`". Since for the first wildcard, the

constant "`alice`" is appearing in all log file messages, the log mining algorithms are unable to identify it as a wildcard. In the case of such scenarios, the author regarded the manually generated pattern ***P*** successfully found even if the pattern generated by the log mining algorithm was reporting a constant for such wildcard. For example, in the case of the above example scenario, the log mining algorithm reports the pattern "`alice : unable to resolve host *`", and the manually created pattern "`* : unable to resolve host *`" will be regarded as found.

One of the key points while evaluating the experiment results is that if redundant patterns are identified by the log mining tools, which do not match the manually generated ones, the pattern detection rate is not penalized. For instance, if the tool not only detects/generates all expected patterns, but also identifies 15-20 additional patterns, the overall matching score will not be penalized or decreased because of these extra patterns, instead, will still be 100%. The reason for this is that in the field of data and pattern mining, it is common for log mining algorithms to provide redundant patterns that represent the same knowledge as other patterns. Likewise, unlike a false positive or false negative in the classical machine learning, a redundant pattern is not a false pattern, but something that actually exists in the log file, which is just either too specific or too generic.

3.5 Experiment Results

In this sub-chapter of the thesis, the author discusses the outcomes achieved at the end of the quality and performance testing experiments.

During the experiments, the author of the thesis tested the chosen log mining tools by changing only required input parameters since his objective was to check the capability of these log parsing tools in terms of quality of generated results and their efficiency when there would not be any human interaction with the tool such as the modification of other settings by enabling, for instance, the pre-processing with the help of regular expressions that can be utilized to replace IP addresses with special tags, and so on.

Some of the tested tools (e.g., ***Drain*** and ***LogCluster***) support advanced input pre-processing functionality, for example, pre-processing with the help of regular expressions for replacing IP addresses with special tags, etc. Even though some tools were supporting such pre-processing functionality, and had it even enabled by default, since this functionality was not supported by other tested tools, the author disabled it for the sake of fair comparisons. Another reason of disabling pre-processing feature was that the pre-processing of the log files assumes previous knowledge on the content of the log messages, which contradicts the fact that in the beginning of log parsing process, the human analyst has usually no information regarding the content of log data.

In terms of the experiment results, even though the author was running tools with various input parameters, for reporting in this thesis, he picked only three or four combinations of input options that were making log mining tools produce results with best quality and achieve highest efficiency. Additionally, while conducting performance testing experiments, each input value of every tool was tested three times on each dataset in order to find out the average of results. Likewise, the maximum amount of time that the author waited after the execution of a certain log parsing tool was 36 hours. If a particular log mining tool was not able to produce results within the given timeframe, the running process would be stopped, and it would be concluded that the tool did not manage to generate/parse results. Finally, even though the modern multi-CPU and multi-core systems offer a lot of multithreading opportunities, and the system that the author used for conducting the experiments was a relatively powerful multi-CPU system, overall, it was decided not to utilize the multithreading extensively. The main reason was that while parsing the log message lines, most algorithms face complexities of safely and efficiently syncing/sharing clustering data between the threads, which leads to non-deterministic clustering outcomes.

Regarding the quality test experiments, it is worth to note that since the *su*, *sudo* and *sshd* datasets contained some rare events/log messages as well, it is common for log mining tools not to be able to detect/identify those rarely occurring patterns. In other words, the fact that there were some log message types that appeared too little, the pattern detection rate of log mining algorithms was expected to be perfect. In order for log parsing tools to identify rarely occurring log messages as patterns, the author of this thesis paper contacted the authors of those tools to get some recommendations regarding the possible options and parameters that could have improved the quality of produced results. Out of seven log mining tools used during the experiments, the authors of five of them replied back, which were the authors of ***IPLoM***, ***AEL***, ***LenMa***, ***LogCluster***, and ***Drain3***.

Another key point here is that most log parsing algorithms, for example ***IPLoM***, ***Drain3***, ***LenMa*** and ***AEL***, which were tested during the experiments, are designed with an assumption that across multiple log message lines, an identified pattern should have the same number of words for the corresponding log messages (see section 2 for more details). However, due to this kind of algorithm design, while testing some log mining tools to parse *suricata* dataset, produced results had very poor pattern detection rates. For instance, if there are two log message lines such as "*SNMP get with public Classification: recon {UDP} 10.1.1.1:45661 -> 10.1.1.2:161*" and "*SQL injection Classification: recon {TCP} 10.1.1.1:45661 -> 10.1.1.2:3333*", instead of identifying them as one generic pattern, most tools, due to the difference in the number of words in log messages (10 and 8 words respectively, since substring "*SNMP get with public*" contains 4 words and substring "*SQL injection*" - 2 words), would generate two distinct patterns. As a result, except for the ***LogCluster***, which achieved 100% of the pattern detection rate while parsing Suricata IDS alert messages, other log mining tools, regardless of the various input parameters, performed very poorly with pattern detection rate of 0%.

3.5.1 Quality Testing

IPLoM

While testing ***IPLoM*** with *su*, *sshd*, *sudo* and *suricata* datasets, the author was mainly changing two input parameters like ***CT***, which stands for cluster goodness threshold, and ***lowerBound*** to achieve outcomes with better quality. The summary of the quality testing results along with the optimal input parameters of ***IPLoM*** can be found in the **Table 3** below (the figures in parentheses represent the total number of detected patterns).

After testing several input parameter values to parse *sudo* logs, the author ended up with three different options that were producing the best results. One of them is when the author set ***CT*** to be equal to 0.3 and ***lowerBound*** to be 0.3 as well. The second combination of input parameter settings is when ***CT*** was equal to 0.3 and ***lowerBound*** was 0.1. And finally, the last tested parameters were ***CT*** to be 0.5 and ***lowerBound*** to be 0.3. These three combinations of input parameters produced, in total, eight templates that demonstrated 80% of the pattern detection rate.

In order to parse *su* log messages, the author, first, set ***CT*** to be equal to 0.6 and ***lowerBound*** to be 0.1. In total, these values generated 14 templates, where most logs messages were covered with a lot of specific patterns rather than generic ones. In terms of the pattern detection rate, the tested values showed approximately 75% (74.7%) of match with the manually identified clusters. Secondly, input values, where ***CT*** was 0.5 and ***lowerBound*** was 0.2, were tested. The number of overall produced templates was 11, in which, compared to the first threshold values, relatively fewer specific patterns were covering the log messages, likewise, this time, a bit higher pattern detection rate, such as 83% of match, was achieved. Finally, the author executed the tool with input parameters, where both ***CT*** and ***lowerBound*** were equal to 0.3. These input values identified same number of templates as the second threshold, which was 11. The difference, however, was that these templates were mostly generic ones. In other words, in comparison with the previous threshold values, this time, most logs messages were covered by generic patterns, consequently, reaching higher pattern detection rate, which was 91.3%.

The first combination of input values, that were tested while parsing *sshd* logs, is ***CT*** to be 0.6 and ***lowerBound*** to be 0.1. Overall, the tool identified 52 templates, most of which were too specific patterns, thus, resulting in approximately 78.3% of pattern detection rate. As the second combination, the author set ***CT*** to be equal to 0.5 and ***lowerBound*** to be 0.2. This time, the tool managed to identify 49 templates and achieved better pattern detection rate with about 84% of match. Finally, the author tried the combination of ***CT*** to be 0.3 and ***lowerBound*** to be 0.3 as well. In comparison with the second combination of input values, almost same outcomes were achieved – 47 templates with generic patterns and nearly 84.1% of pattern detection rate.

As a recommendation for ***IPLoM*** to parse/identify rarely occurring events, the author of the tool advised the usage of **File Support Threshold** and **Partition Support Threshold** parameters. According to the author of the ***IPLoM***, by setting both of these thresholds to 0, it would be possible that clusters with rare events were found. However, even though the author of this thesis was mainly testing ***CT*** and ***lowerBound*** input parameters, the above-recommended input options, by default, were set to be 0. In other words, the experiments conducted with ***IPLoM*** were already configured to identify rarely appearing log messages.

Finally, regardless of the tested combinations of input values, ***IPLoM*** achieved 0% of pattern detection rate while parsing *suricata* dataset.

	CT = 0.1, lowerBound = 0.1	**CT = 0.3, lowerBound = 0.3**	**CT = 0.3, lowerBound = 0.1**	**CT = 0.5, lowerBound = 0.3**	**CT = 0.5, lowerBound = 0.2**	**CT = 0.6, lowerBound = 0.1**
***sudo* logs**		80% (8)	80% (8)	80% (8)		
***su* logs**		91.3% (11)			83% (11)	74.7% (14)
***sshd* logs**		84.1% (47)			84% (49)	78.3% (52)
***suricata* logs**	0% (17)	0% (272)				0% (270)

Table 3. Quality Test Results and Optimal Input Parameters of IPLoM

Spell

The main input parameter the author was modifying while running ***Spell*** on datasets *suricata, sshd, sudo* and *su*, is called ***tau***, which basically provides a threshold for merging two messages into a pattern, where ***tau*** reflects the number of common tokens (words) in messages. The summary information regarding the quality testing outcomes of Spell is provided in **Table 4**.

After finishing the testing of various input values for parameter ***tau*** to parse *sudo* logs, the author selected three distinct values, where outcomes with relatively higher quality were generated. The first value, that ***tau*** was set, equals to 0.85. This value led to the identification of 26 templates, which were mainly specific ones rather than generic. In terms of the pattern detection rate, with this value, the tool achieved 60% of the match. For the second value, the author set ***tau*** to be equal to 0.7. Even though the pattern detection rate in this case was identical to the one achieved in the previous threshold value, the number of produced templates decreased to 10, where most part of log messages were covered by the generic patterns. Nevertheless, the best quality results while parsing *sudo* logs were achieved when ***tau*** was set to 0.5. This time, the tool managed to get 80% of the pattern detection rate by generating 7 generic templates to cover majority of the log message lines.

The input parameter values, that produced much higher quality results while parsing *su* log messages, were same as the ones used to parse *sudo* log messages. For the ***tau*** 0.85, the overall number of identified templates was 30, where mainly specific patterns were noticeable. Likewise, with the ***tau*** equals to 0.85, the tool achieved approximately 50% (49.8%) of the pattern detection rate. When the author changed the value of the input parameter and set ***tau*** to be equal to 0.7, even though the tool generated 14 templates with some specific patterns in it, in terms of the quality, it managed to reach 83% of the pattern detection rate. As it was in the case of *sudo* log messages, when ***tau*** was set to 0.5, the tool produced the best outcomes in terms of the quality. In total, 12 templates were identified, where most log messages were covered by the generic patterns, moreover, the pattern detection rate was much higher than the previous two results – it was equal to 91.3%.

Similar to the input argument values used to mine *su* and *sudo* logs, the author of this thesis paper found out that ***tau*** 0.85, ***tau*** 0.7 and ***tau*** 0.5 were the optimal parameters in the identification process of templates with much higher pattern detection rate, in other words, with the best quality and accuracy. While testing ***tau*** 0.5, the tool achieved its best outcome by identifying 42 templates and obtaining 72.5% of the pattern detection rate. The quality of the produced results was getting worse when the ***tau*** was set to 0.7 and 0.85. In the case of ***tau*** 0.7, the tool generated 5111 templates, that were too specific for each pattern. In other words, Spell identified almost each log message with variables as a separate pattern. In terms of the quality, 49.3% of the pattern detection rate was achieved. The worst-case scenario was reached when ***tau*** was set to 0.85 since the tool crashed and produced only partial results by generating 19152 patterns. As a result, there was not an efficient way to find out the pattern detection rate achieved by the given value of the ***tau***.

In the case of *suricata* dataset, ***Spell*** achieved 0% of the pattern detection rate in all tested input values.

	tau = 0.55	**tau = 0.5**	**tau = 0.7**	**tau = 0.85**	**tau = 0.9**
***sudo* logs**		80% (7)	60% (10)	60% (26)	
***su* logs**		91.3% (12)	83% (14)	49.8% (30)	
***sshd* logs**		72.5% (42)	49.3% (5111)	<**tool crashed**> (19152)	
***suricata* logs**	0% (111)		0% (206)		0% (6886)

Table 4. Quality Test Results and Optimal Input Parameters of Spell

Drain3

One of the key input arguments that were tested while running the tool is ***st,*** which denotes the similarity threshold, where if the percentage of similar tokens for a log message is below this number, a new log cluster will be created. The second key parameter is ***depth***, which defines the max depth levels of log clusters. Optimal input values along with the summary of the quality testing experiments for ***Drain3*** are described in the **Table 5**.

The first pair of input parameters, that author tried while testing the ***Drain3*** on *sudo* log messages dataset, are ***st*** equals to 0.4, and ***depth*** equals to 4. The tool managed to identify overall 14 clusters with mainly specific patterns in it. Moreover, the pattern detection rate of the tool was estimated and found to be 60%. Even though the next two combinations of input values, which were used during the experiment, were completely different from each other, the achieved outcomes were identical. In other words, when the author set ***st*** to be 0.3 and ***depth*** to be 6, likewise, when he set ***st*** to be equal to 0.6 and ***depth*** to 5, the tool generated 14 clusters, most of which were specific patterns, and obtained 60% of the pattern detection rate.

The two combinations of distinct input values, that the author experimented in order to parse *su* log messages, produced nearly the same outputs with the same pattern detection rate. These values are ***st*** 0.3 and ***depth*** 6, and ***st*** 0.6 and ***depth*** 5. By using these pairs of options, ***Drain3*** identified 17 clusters, in the case of ***st*** 0.3 and ***depth*** 6, and 16 clusters, in the case of ***st*** 0.6 and ***depth*** 5. Even though the pattern detection rate that was obtained in these experiments were equal to 83%, the generated templates consisted of more specific patterns rather than generic ones. The last input parameters that were ***st*** 0.4 and ***depth*** 4, on the other hand, not only enabled the tool to identify more generic clusters, but also achieved much higher pattern detection rate. In total, the number of generated templates was 14, and the corresponding pattern detection rate was 91.3%.

While parsing *sshd* log messages, the author managed to achieve the best quality outcomes when he set the input values ***st*** to be 0.4 and ***depth*** to be 4. In terms of the identified clusters, ***Drain3*** produced 40 of them, where the number of generic patterns were prevailing the number of specific ones. Furthermore, the pattern detection rate obtained with this group of input parameters was nearly 90% (89.9%). The second combination of input values that produced relatively better quality results were ***st*** equals to 0.3 and ***depth*** to be 6. This time, the tool generated 42 clusters, where almost half of it were specific patterns. Nevertheless, there was 84.1% of the pattern detection rate achieved. In comparison with two previous combinations of input parameters, the last pair of them produced relatively poor results. When the author set the value of ***st*** to 0.6 and ***depth*** to 5, ***Drain3*** identified 50 clusters in total. While comparing with previous thresholds, this time, generated results contained slightly more specific patterns and had a pattern detection rate of 78.3%.

When it comes to parsing rare events, according to the author of Drain3, the tool does not provide specific options for rarely occurring log messages. Even though it is expected for Drain to generate the templates when there are 2 or more log messages in the same group, the author of Drain says that the error of the tool mainly comes from the fact that some events contain the same number of tokens, and their first ***k*** tokens are the same so that multiple events may be assigned into the same group.

The pattern detection rate achieved by the end of parsing *suricata* dataset by ***Drain3*** was equal to 0%.

	st = 0.3, depth = 6	**st = 0.4, depth = 4**	**st = 0.6, depth = 5**
***sudo* logs**	60% (14)	60% (14)	60% (14)
***su* logs**	83% (17)	91.3% (14)	83% (16)
***sshd* logs**	84.1% (42)	89.9% (40)	78.3% (50)
***suricata* logs**	0% (108)	0% (69)	0% (156)

Table 5. Quality Test Results and Optimal Input Parameters of Drain3

AEL

The main two input parameters, which were tested with various values while experimenting ***AEL*** over *suricata*, *sshd*, *sudo* and *su* datasets, were ***mineventcount*** and ***mergepercent***. The author provides the summary of the quality testing outcomes of ***AEL,*** likewise, the input values in the **Table 6**.

Overall, there were three combinations of input parameters tested to parse *sudo* logs, two of which generated higher quality outcomes, while the third one produced output with a poor pattern detection rate. First, the author set ***mineventcount*** to be equal to 2 and ***mergepercent*** to 0.3. This combination of inputs identified, in total, 12 templates, where a bit more specific patterns were noticeable. In terms of the pattern detection rate, however, the given combination of values achieved 80% of the matching with the manually generated patterns. Results got improved when the author left the ***mineventcount*** unchanged and modified just ***mergepercent*** to be 0.6. Even though the obtained pattern detection rate was same as in the case of the previous threshold values, 80%, the number of produced templates decreased to 8, in which, mainly generic patterns were dominating. As it was stated above, the third combination of tested input values, which are ***mineventcount*** to be equal to 10 and ***mergepercent*** to 0.6, led to the worst outcomes, where 23 mainly specific patterns were identified, thus, obtaining 40% of the pattern detection rate.

Based on the observations obtained while parsing *sudo* logs with ***AEL***, the author decided to test just the two combinations of optimal input values from previous experiments to parse *su* log messages. Therefore, the first input parameters are ***mineventcount*** to be 2 and ***mergepercent*** to 0.3. In general, ***AEL*** identified 19 templates, which consisted of mainly specific patterns. On the other hand, these produced results managed to achieve a pattern detection rate of 91.3%. The second combination of input values, that are ***mineventcount*** to be 2 and ***mergepercent*** to 0.6, however, identified fewer patterns, which, in total, were 11, where most log messages were covered by generic patterns rather than the specific ones. Nevertheless, this combination of options also obtained 91.3% of the pattern detection rate.

For parsing *sshd* log messages, overall, the author managed to find three optimal combinations of input values, which were producing results with pattern detection rates above 70%. The first pair of values, that generated, in comparison with other threshold values, relatively low-quality outcomes, were ***mineventcount*** to be 10 and ***mergepercent*** to 0.3. With these values, ***AEL*** identified 76 templates, which contained too many specific patterns. As a result, the pattern detection rate obtained in this case was approximately 72.5%. In contrast, the pattern detection rate of generated results got better, when the author changed the value of ***mergepercent*** to 0.8 and left ***mineventcount*** same. This time, the tool identified 34 templates, where log message lines were covered mainly by generic patterns. Consequently, in the case of the second combination of input parameters, the pattern detection rate increased to 84.1%. However, among the three optimal combinations of values, the third one reached the highest quality and pattern detection rate in the generated outcomes. The author tested the tool by, again, leaving ***mineventcount*** unchanged and modifying only ***mergepercent*** to 0.6. ***AEL*** created 37 templates, where the number of generic patterns were prevailing the number of specific ones, and obtained the pattern detection rate of 89.9% with the manually generated templates.

In terms of the identifying rarely occurring log templates, the author of ***AEL*** suggested to run ***AEL*** twice, first, by generating frequent patterns and a file with rarely occurring events, then, running the tool on the file with rare logs with lower threshold values. The problem in this case was that since there were no other implementations available, the author of this thesis paper was using the University of Hong Kong paper [8] implementation of ***AEL***, which does not produce outliers file with rare events so that to run ***AEL*** on them with lower threshold values.

When the author tested ***AEL*** to parse *suricata* dataset, the pattern detection rate obtained at the end was 0%.

	mineventcount = 2, mergepercent = 0.3	**mineventcount = 2, mergepercent = 0.6**	**mineventcount = 10, mergepercent = 0.3**	**mineventcount = 10, mergepercent = 0.6**	**mineventcount = 10, mergepercent = 0.8**
***sudo* logs**	80% (12)	80% (8)		40% (23)	
***su* logs**	91.3% (19)	91.3% (11)			
***sshd* logs**			72.5% (76)	89.9% (37)	84.1% (34)
***suricata* logs**			0% (172)	0% (73)	

Table 6. Quality Test Results and Optimal Input Parameters of AEL

LogSig

When ***LogSig*** was used in the experiments to parse *sudo*, *su*, *sshd* and *suricata* datasets, the author was mainly testing an input parameter called ***groupNum***, which is used to specify the number of groups that log messages had to be portioned, in other words, it tells the tool to partition logs into ***k*** groups. The detailed information about the quality of produced results and the input values used to generate those outcomes are provided in the **Table 7**.

Starting from the ***groupNum*** values 20, 40 and above, when ***LogSig*** was used to parse *sudo* log message lines, results were getting worse and worse such that the pattern detection rates were declining, and the number of specific patterns were increasing with each threshold value. Consequently, the first input value of ***groupNum***, which is considered to be optimal and produce relatively better results, was 15. The tool identified 9 templates, which contained comparatively more specific patterns, nevertheless, achieved 40% of the pattern detection rate. The second input parameter value was ***groupNum*** to be equal to 10. In this case, the number of generated templates was 7, where compared to the previous threshold value, there were fewer specific patterns. In terms of the pattern detection rate, however, it was same – 40%. The best outcomes were produced, when the author set ***groupNum*** to be equal to 5 since the tool identified 4 generic templates, thus, achieving 60% of the pattern detection rate.

When ***LogSig*** was configured to parse *su* log message lines, results with relatively higher pattern detection rates were produced after the author set ***groupNum*** to be equal to 20. The tool managed to identify 20 templates, where generic patterns were dominating. Therefore, there was a 66.4% of the matching with the manually identified patterns. Even though the next input value, which was ***groupNum*** to be 25, led to the generation of outcomes with almost the similar pattern detection rate, 60%, the identified templates, that in total were 23, contained mostly specific patterns compared to the results obtained when ***groupNum*** was 20. For the final input value, the author set ***groupNum*** to be 15, where the tool identified 15 templates, which had relatively poor quality since the pattern detection rate was 58.1%.

Despite the fact that the author tried various input values while testing ***LogSig*** on *sshd* log messages, overall, the tool did not manage to produce outcomes with more than 40% of the pattern detection rate. Three out of four input values for ***groupNum***, which were 30, 45 and 50, achieved 34.8% of the pattern detection rate. The number of identified templates were 27, 38 and 44 respectively, where, in comparison with each other, ***groupNum*** 30 produced more generic patterns rather than specific ones. The last input value that the author tested, ***groupNum*** to be 40, obtained even lower pattern detection rate, which was 31.9%. This time, in total, the tool identified 38 templates, where the number of specific patterns were much more than the generic patterns.

When being tested on *suricata* dataset, based on the results obtained by ***LogSig***, the pattern detection rate was concluded to be 0%.

	groupNum = 5	**groupNum = 10**	**groupNum = 15**	**groupNum = 20**	**groupNum = 25**	**groupNum = 30**	**groupNum = 40**	**groupNum = 45**	**groupNum = 50**
***sudo* logs**	60% (4)	40% (7)	40% (9)						

***su* logs**			58.1% (15)	66.4% (20)	60% (23)				
***sshd* logs**						34.8% (27)	31.9% (38)	34.8% (38)	34.8% (44)
***suricata* logs**						0% (21)			0% (30)

Table 7. Quality Test Results and Optimal Input Parameters of LogSig

LenMa

The only input parameter, which the author was modifying to test ***LenMa*** with *su, sudo, sshd* and *suricata* datasets, was ***threshold***. In the **Table 8**, the author provided the optimal input values used during the experiments along with the pattern detection rates of the generated outputs of ***LenMa***.

After testing several input options, the author stopped on three threshold values such as 0.5, 0.7 and 0.9, which produced results with relatively higher pattern detection rates. For instance, when the ***threshold*** value was set to 0.9 while parsing *sudo* logs, the tool identified in total 20 templates, where mainly specific patterns were exceeding. In terms of the pattern detection rate, with the given ***threshold*** value, the tool managed to achieve the rate of 40%. On the other hand, by setting the ***threshold*** value to 0.5 and 0.7, the author managed to produce outcomes with 60% of the pattern detection rate. Overall, these values identified 14 templates, most of which were generic patterns.

In contrast, while parsing *su* log messages, the above-mentioned threshold values generated templates, the pattern detection rates of which had large differences with each other. For example, the lowest pattern detection rate was obtained when the author set ***threshold*** to be equal to 0.9. The tool produced 37 templates, which mainly consisted of the specific patterns. Consequently, it got a pattern detection rate of 58.1%. Secondly, when ***threshold*** value was set to 0.7, ***LenMa*** achieved 66.4% of the pattern detection rate by producing 28 patterns that consisted of slightly fewer specific templates. Nevertheless, the outcomes with best pattern detection rates were generated when ***threshold*** was set to be equal to 0.5. This time, the tool, in total, produced 19 templates, and by having mainly generic patterns, reached 83% of the pattern detection rate.

In the final part of the experiments with ***LenMa***, when it was used to parse *sshd* logs, the pattern detection rates of the produced outcomes were high and close to each other when ***threshold*** values were set to 0.5 and 0.7. In the case of the first value, 0.5, the tool identified 40 templates, most of which were generic patterns, thus, leading to the pattern detection rate of 89.9%. The second ***threshold*** value, 0.7, generated almost similar results – 47 templates with mainly generic ones, and reached 84.1% of the pattern detection rate. On the other hand, results decreased drastically when the ***threshold*** was set to be 0.9. This time, ***LenMa*** led to the creation of 63 templates, where most logs messages were covered by the specific patterns and got a pattern detection rate of 49.3%.

According to the author of ***LenMa***, with the current implementation of the algorithm, it is not possible to identify rarely occurring events. In other words, since ***LenMa*** is based on the statistical analysis results of log messages, it is quite difficult to extract rarely happening log messages.

While parsing the final dataset – *suricata* dataset, the achieved pattern detection rate by ***LenMa*** was 0%.

	threshold = 0.5	**threshold = 0.7**	**threshold = 0.9**
***sudo* logs**	60% (14)	60% (14)	40% (20)
***su* logs**	83% (19)	66.4% (28)	58.1% (37)
***sshd* logs**	89.9% (40)	84.1% (47)	49.3% (63)
***suricata* logs**	0% (197)	0% (203)	0% (241)

Table 8. Quality Test Results and Optimal Input Parameters of LenMa

LogCluster

In terms of the input parameters, overall, the author was changing the values of three input options while using ***LogCluster*** in the experiments. During the experiments with the datasets *sudo*, *su* and *sshd*, the author utilized two input options such as ***support*** and ***wweight***, where ***wweight*** denotes the word weight threshold. On the other hand, he was using ***rsupport*** parameter, which stands for the relative support, while testing the tool on *suricata* dataset. When it comes to parse rarely occurring log messages in a file, according to the author of ***LogCluster***, it is possible to identify rarely occurring events by running the tool on the generated outliers file with lower threshold values. In other words, first, the tool will be mining the main file and create cluster of first outliers. Then, ***LogCluster*** can be used to mine the cluster of first outliers and create cluster of second outliers. Based on the need, this process might continue until all possible rarely occurring patterns are found. This process is called **Iterative Clustering**, and this pattern mining technique has also been described in a paper that discusses various usage techniques of ***LogCluster*** [21]. As a result, the author of this thesis decided to run ***LogCluster*** on the outliers of *sudo*, *su* and *sshd* logs to find out rarely appearing messages. The summary of the quality testing experiments with ***LogCluster*** can be found in the **Tables 9** and **10**.

In the case of *sudo* log messages, the first input value, that ***support*** threshold was set to, was 20. In total, ***LogCluster*** identified 11 clusters and 88 outliers, where specific patterns were dominating among the generated clusters. The produced results obtained 40% of the pattern detection rate in total. Almost similar outcomes were achieved when ***support*** was set to be equal to 40. The number of identified clusters decreased to 8, while outliers remained same, 88, and the pattern detection rate of generated clusters was 40% as well. The next two ***support*** threshold values, which were 60 and 80, produced results with relatively higher pattern detection rates compared to the previous two threshold values. In the case of 80, 5 clusters and 35 outliers were identified, while in the case of 60, 8 clusters and 22 outliers were generated. In both cases, the produced clusters were mainly generic patterns, consequently, leading to the higher pattern detection rate of 60%.

Same ***support*** threshold values were also tested while parsing *su* log messages with ***LogCluster***. This time, results with relatively lower pattern detection rates were produced in the case of ***support*** to be equal to 60 and 80. The number of identified clusters and outliers were same in both cases – 9 clusters and 210 outliers. In terms of the pattern detection rate, these two values enabled the tool to reach 41.5% of the pattern detection rate. The quality of generated outcomes got improved when the author set ***support*** to be 20. The tool identified 25 clusters along with 16 outliers, where there were many specific patterns among generated templates. Nevertheless, 58.1% of the pattern detection rate was managed to be achieved by ***LogCluster***. Finally, the author tested the ***support*** to be 40. This time, the tool identified 19 clusters, where mainly generic patterns were dominating, and 18 outliers. Likewise, the best pattern detection rate was obtained with this ***support*** value – 66.4%.

While parsing *sshd* log messages, two combinations of input parameters were used: ***support*** threshold only and ***support*** threshold along with ***wweight*** parameter. The results with worst pattern detection rate were produced when ***support*** was set to 60. The tool identified 52, mainly specific, clusters and 335 outliers, moreover, the obtained pattern detection rate was 20.3%. The second ***support*** value was 40, where ***LogCluster*** identified 53 clusters and 407 outliers. Nevertheless, this time, the tool achieved a bit higher pattern detection rate - 29%. Finally, in comparison with the previous threshold values, the ***support*** value to be 20 produced outcomes with relatively higher pattern detection rate – 34.8%. The downside of this ***support*** value was that overall, 79 clusters, in which specific patterns were mainly noticeable, and 249 outliers were identified. As a result, in order to improve the quality of the produced outcomes, the author decided to test ***support*** value of 20 further with ***wweight*** input parameter. First, he set ***wweight*** to be equal to 0.5. In this case, the tool identified 43 clusters, mostly generic patterns, and 249 outliers. Additionally, this time, much higher pattern detection rate was obtained – 43.5%. The pattern detection rate of results increased further to 49.3% when the author set ***wweight*** to be 0.8. Overall, ***LogCluster*** identified 39 generic clusters and 249 outliers.

In order to parse rarely occurring events of *sudo* dataset, the author executed the tool with ***support*** to be 5 on the outliers of ***support*** threshold 60 and 80 since they previously produced comparatively higher quality outcomes. While running the tool with ***support*** to be 5 on outliers of ***support*** 80, the tool identified five new patterns and increased the overall pattern detection rate from 60% to 80%. However, in the case of outliers of ***support*** 60, there was 40% of an increase, in other words, the pattern detection rate of 100% was achieved.

To parse rarely occurring events of *su* log messages, the author executed the tool on the outliers of ***support*** 40 with lower threshold value of 2. In this case, the tool managed to identify five new clusters and increased the overall pattern detection rate by 16.6%, which in total, became 83%.

In the case of *sshd* logs, the author first tested ***support*** value of 5 on the outliers of ***support*** 20 and ***wweight*** 0.8. ***LogCluster*** produced additional 20 clusters and 41 outliers, which led to an increase in the pattern detection rate by 20.3%, thus, making it 69.6%. Later, the author tested even lower threshold value, which was 2, on the newly generated 41 outliers. This time, the tool identified 12 additional clusters, which had a pattern detection rate of 78.3% with the manually generated patterns.

As it was previously mentioned in this thesis paper, only ***LogCluster*** managed to produce results with higher quality while parsing Suricata IDS alert messages. First, the author tested the tool by setting ***rsupport*** to be 10, in which, 0% of the pattern detection rate was achieved. Later, however, when ***rsupport*** was set to be 99, the tool identified one generic cluster to cover all Suricata IDS alert messages, thus, managed to obtain a pattern detection rate of 100%.

	rsupport = 10	**rsupport = 99**	**support = 20**	**support = 40**	**support = 60**	**support = 80**	**support = 20, wweight = 0.5**	**support = 20, wweight = 0.8**
***sudo* logs**			40% (11)	40% (8)	60% (8)	60% (5)		
***su* logs**			58.1% (25)	66.4% (19)	41.5% (9)	41.5% (9)		
***sshd* logs**			34.8% (79)	29% (53)	20.3% (52)		43.5% (43)	49.3% (39)
***suricata* logs**	0% (9)	100% (1)						

Table 9. Quality Test Results and Optimal Input Parameters of LogCluster

	sudo outliers (**support** 80)	*sudo* outliers (**support** 60)	*su* outliers (**support** 40)	*sshd* outliers (**support** 20 and **wweight** 0.8)	*sshd* outliers (**support** 5 and **wweight** 0.8)
support 2			83% (5)		78.3% (12)
support 5	80% (5)	100% (4)		69.6% (20)	

Table 10. Quality Test Results of Iterative Clustering of LogCluster

3.5.2 Summary of Quality Testing

Below in the **Table 11**, the author provides the summary of best pattern detection rates obtained by the log mining algorithms while being tested to parse the quality testing datasets.

	IPLoM	**Spell**	**Drain3**	**AEL**	**LogSig**	**Lenma**	**LogCluster**
***sudo* logs**	80%	80%	60%	80%	60%	60%	100%
***su* logs**	91.3%	91.3%	91.3%	91.3%	66.4%	83%	83%
***sshd* logs**	84.1%	72.5%	89.9%	89.9%	34.8%	89.9%	78.3%
***suricata* logs**	0%	0%	0%	0%	0%	0%	100%
Average Rates	63.85%	60.95%	60.3%	65.3%	40.3%	58.23%	90.33%
Average Rates without suricata logs	85.1%	81.3%	80.4%	87.1%	53.7%	77.6%	87.1%

Table 11. Best Pattern Detection Rates of Each Log Parsing Algorithm During Quality Testing

According to the table above, it can be seen that while parsing the *suricata* dataset, most algorithms achieved the pattern detection rate of 0%. This illustrates the fact that it is important to design algorithms to detect the patterns, where the number of words in corresponding log messages is not necessarily a constant. Likewise, this finding demonstrates that a number of current log mining algorithms such as Drain, IPLoM, Lenma and AEL are lacking in this area, and thus, their design cannot be regarded suitable for all log types.

It is also worth to mention that during the quality testing experiments, the author discovered that the selection of parameters can have a significant effect on the outcome and using the right parameter values can noticeably improve the result. For instance, with some values, the algorithms can have lower pattern detection rates, while with other values, better results will be achieved. Consequently, in order to be able to use certain log parsing algorithm successfully, engineers must have a good understanding of input parameters, which was the case in this thesis paper, where the author of the thesis consulted with the authors of the tools to find out optimal input values of algorithms.

Finally, based on the quality testing experiment results, apart from LogSig, all other tested algorithms were having a quite good performance on the first three datasets, *su*, *sudo* and *suricata*. For example, the difference between the average pattern detection rates of Lenma for the first three datasets, that is 77.6%, and AEL or LogCluster, that is about 87%, is nearly 10%. In other words, besides the quality testing results of LogSig, the difference between worst and best tools is less than 10%. However, according to the paper from University of Hong Kong [8], there were much larger differences in the accuracy of produced results of algorithms, for example, for LogCluster 66.5% and for Drain 86.5%, which is the range covering 20%. This can be explained by the fact that not all algorithms were tested with good input settings. Likewise, the input data was preprocessed by Hong Kong researchers. To be more specific, IP addresses were replaced by special tokens, which, in general, distorts the original log data and might also influence the quality of generated results.

Summary of the main findings obtained as a result of the quality testing experiments:

- It is important to design log mining algorithms to detect patterns, where the number of words in corresponding log messages is not necessarily constant. Hence, log parsing algorithms like Drain, IPLoM, Lenma and AEL, due to their designs, are not suitable for all log types.

- Selection of input values can have significant effect on quality of produced outcome and usage of right parameter values can noticeably improve result.

3.5.3 Performance Testing

IPLoM

The summary of the performance testing results along with the tested input parameters of ***IPLoM*** can be found in the **Table 12**.

When the author executed ***IPLoM*** to parse **http-apacheformat.log** dataset, mainly, four different combinations of input parameters were tested. First, the tool was tested by setting both ***CT*** and ***lowerBound*** to 0.1. As a result, during 27 seconds period of time, 284 templates were identified, where the log mining tool consumed 286200 KB of memory and 30.84 of total CPU time (kernelCPU 2.05 and userCPU 28.79). The next combination of tested input values, which were ***CT*** and ***lowerBound*** to be equal to 0.3, produced 439 templates within 27 seconds. In terms of the memory and CPU time consumption, the tool consumed 286732 KB of memory and 30.43 of total CPU time (kernelCPU 2.02 and userCPU 28.41). Then, when ***CT*** was equal to 0.5 and ***lowerBound*** was 0.2, ***IPLoM*** parsed the logs within 27 seconds and generated 494 templates, where it consumed 286756 KB of memory and 30.8 of CPU time (kernelCPU 1.99 and userCPU 28.81). The tool managed to achieve almost the same results when it was tested with ***CT*** to be 1 and ***lowerBound*** to be 1. This time, even though the parsing time was similar – 27 seconds, the number of identified templates was 611. Likewise, there was not too much of a difference regarding the memory and CPU time consumption, which were 290864 KB of memory and 30.94 of CPU time (kernelCPU 2.02 and userCPU 28.92).

IPLoM did not manage to achieve any results while being tested on **windows-text.log** dataset. Additionally, while analyzing the source code of the utilized implementation of ***IPLoM*** [26], it was found out that the tool requires ***maxEventLen*** input option, which was set to 200 by default. This parameter denotes the length of the longest log/event, which is used in step 1 of original algorithm [5] to split logs into partitions according to their length. By running the tool with the default value of ***maxEventLen***, the author received "*list index out of range*" error since **windows-text.log** dataset contained long log message lines such as more than 18000 characters. As a result, the author changed the value of this parameter to be 30000 during the experiments. It worth to note, however, that in order to be able to set the correct value for ***maxEventLen***, security engineers need to have a prior knowledge about the parsed log file, which contradicts to the assumption of this thesis paper that log mining tools were being tested without any human interaction with the tool such as the modification of configurations, etc. In terms of the tested parameters, when the author tested the combination of input values such as ***CT*** to be 0.5 and ***lowerBound*** to be 0.2, the tool did not finish parsing the logs in a reasonable amount of time (approximately 36 hours). Regarding the other combinations like ***CT*** and ***lowerBound*** to be 0.3, and ***CT*** and ***lowerBound*** to be 0.1, even though ***IPLoM*** could not finalize the parsing process, it still identified some number of templates, 2901 and 1991 templates respectively.

When ***IPLoM*** was tested on **logsrv.log.2** and **xs19-syslog.log** datasets with combinations of input values of ***CT***=0.3 and ***lowerBound***=0.3, ***CT*** = 0.5 and ***lowerBound*** = 0.2, and ***CT*** = 1 and ***lowerBound*** = 1, no results were achieved.

	CT = 0.1, lowerBound = 0.1	**CT = 0.3, lowerBound = 0.3**	**CT = 0.5, lowerBound = 0.2**	**CT = 1, lowerBound = 1**
http-apacheformat.log	memory 286200 KB, CPU Time 30.84 (kernelCPU 2.05 + userCPU 28.79)	memory 286732 KB, CPU Time 30.43 (kernelCPU 2.02 + userCPU 28.41)	memory 286756 KB, CPU Time 30.8 (kernelCPU 1.99 + userCPU 28.81)	memory 290864 KB, CPU Time 30.94 (kernelCPU 2.02 + userCPU 28.92)
windows-text.log	<no results>	<no results>	<no results>	
logsrv.log.2		<no results>	<no results>	<no results>
xs19-syslog.log		<no results>	<no results>	<no results>

Table 12. Performance Test Results of IPLoM

Spell

During the experiments, when the author tested ***Spell*** on the performance testing datasets, the tool, except for one case, did not manage to complete parsing process in a reasonable time, which was previously decided to be 36 hours (the summary of the performance testing is described in the **Table 13)**.

In general, the tested input values were ***tau = 1***, ***tau = 0.5*** and ***tau = 0.1***. Among these three distinct parameters, ***Spell*** was able to complete the parsing of only **http-apacheformat.log** dataset when ***tau*** was set to be equal to 1. In total, it took 53 minutes and 7 seconds for the tool to complete the parsing of the logs, where it identified 17006 templates. In terms of the memory and CPU time consumption, ***Spell*** consumed the memory of 255896 KB and total CPU time of 3181.6 (kernelCPU 9.40 and userCPU 3172.20). On other cases, where the tool was tested on **windows-text.log, logsrv.log.2** and **xs19-syslog.log** datasets, no results were achieved.

	tau = 0.1	**tau = 0.5**	**tau = 1**
http-apacheformat.log	<no results>	<no results>	memory 255896 KB, CPU Time 30.94 (kernelCPU 9.40 + userCPU 3172.20)
windows-text.log	<no results>	<no results>	<no results>
logsrv.log.2	<no results>	<no results>	<no results>
xs19-syslog.log	<no results>	<no results>	<no results>

Table 13. Performance Test Results of Spell

Drain3

While analyzing the source code of ***Drain3***, which was downloaded from the official repository [24], the author found out that the given implementation of the tool loads the entire log file into the memory before ***Drain3*** starts its work to parse log messages. This means that the memory usage of ***Drain3*** reflects the size of the log file, not the memory usage of algorithms' data structures. Therefore, the author tested ***Drain3*** on two different scenarios, where in the first case, the tool will be loading the entire log file into the memory and then start the parsing process, and in the second case, ***Drain3*** will be reading log messages from the datasets line by line and perform the mining. The results of the performance testing experiments of ***Drain3*** are described in the **Table 14** and **Table 15**.

The first pair of input values of ***Drain3*** that the author set to parse **http-apacheformat.log** dataset was ***st*** to be equal to 0.3 and ***depth*** to be 6. When the tool was executed based on the first scenario, overall, it took 2.41 seconds to finalize the parsing of logs and identify 204 clusters, moreover, the memory and CPU time consumptions ended up being 49076 KB and 2.65 (kernelCPU 0.05 and userCPU 2.60) respectively. On the other hand, when the author ran ***Drain3*** to parse log messages according to the second scenario, 204 clusters were produced within 2.51 seconds, in which, the tool consumed 20984 KB of memory along with the 2.65 of total CPU time (kernelCPU 0.02 and userCPU 2.63). Next, input values of the tool were changed to ***st*** to be 0.4 and ***depth*** to be 4. This time, in the first case, the tool completed parsing and identification of 296 clusters in 2.85 seconds. In addition, ***Drain3*** consumed 48836 KB of memory and 3.07 of CPU time (kernelCPU 0.08 and userCPU 2.99). Except for the memory consumption, achieved results were almost similar in the second case since within 2.89 seconds, the tool produced 296 clusters, where it also consumed 3.03 of CPU time (kernelCPU 0.02 and userCPU 3.01) and 21096 KB of memory. Finally, when the author set the value of ***st*** to be 0.6 and changed ***depth*** to 5, ***Drain3*** completed parsing of the log messages in 5.57 seconds and generated 785 clusters when it was executed in accordance with the first scenario. In terms of the CPU time and memory consumptions, the tool consumed 49512 KB of memory and 5.82 of total CPU time (kernelCPU 0.05 and userCPU 5.77). When the tool was executed based on the second case, overall, it took 5.49 seconds to finish the mining of logs and produce 785 clusters, in which, the CPU time and memory consumptions of ***Drain3*** were 21728 KB and 5.64 (kernelCPU 0.03 and userCPU 5.61) respectively.

Even though the total CPU time consumptions of ***Drain3*** while parsing **windows-text.log** dataset in accordance with scenarios 1 and 2 were close to each other, the overall memory consumptions were completely different. For instance, in the first scenario, when the input values were set to ***st*** = 0.3 and ***depth*** = 6, the tool identified 1327 clusters in 3 minutes and 46 seconds and utilized 3688412 KB of memory and 233.93 of total CPU time (kernelCPU 3.70 and userCPU 230.23). On the other hand, when the tool was executed by the second scenario, 1327 clusters were produced in 3 minutes and 55 seconds, in which, the memory and CPU time consumptions were 36364 KB and 235.17 (kernelCPU 1.70 and userCPU 233.47) respectively. Secondly, when the second combination of input values, ***st*** to be 0.4 and ***depth*** to be 4, were tested under the first case, ***Drain3*** managed to finalize the parsing process within 7 minutes and 3 seconds and generate 1335 clusters. Furthermore, the associated memory and CPU time consumptions of the tool were 3689744 KB and 429.95 (kernelCPU 5.70 and userCPU 424.25) respectively. When being tested according to the second scenario, it took for the tool to identify 1335 clusters in 6 minutes and 56 seconds, where ***Drain3*** also used 415.54 of total CPU time (kernelCPU 2.60 and userCPU 412.94) and 36544 KB of memory. In approximately 7 minutes and 10 seconds, the tool produced 1653 clusters when it was executed under the first scenario with input values of ***st*** to be equal to 0.6 and ***depth*** to be 5. Likewise, the total CPU time and memory consumptions utilized during the parsing process were 436.98 (kernelCPU 4.83 and userCPU 432.15) and 3691880 KB respectively. In the case of second scenario, on the other hand, it took 7 minutes and 15 seconds for ***Drain3*** to complete the parsing of logs and produce 1653 clusters. This time, ***Drain3*** utilized 434.65 of total CPU time (kernelCPU 2.49 and userCPU 432.16) along with the memory of 38208 KB.

When the author executed ***Drain3*** under the first scenario to parse **logsrv.log.2** dataset, with input values of ***st*** to be 0.3 and ***depth*** to equal to 6, the tool was running for about an hour 1 minute and 12 seconds in order to generate 1719 clusters. In addition, along with 3753124 KB of memory consumption, the tool utilized 3668.88 of total CPU time (kernelCPU 12.49 and userCPU 3656.39). ***Drain3*** finalized parsing of logs nearly in 5 hours 37 minutes and 51 seconds and produced 4016 clusters when input values were changed to be ***st*** = 0.4 and ***depth*** = 4. This time, the corresponding memory and CPU time consumptions ended up being 3755736 KB and 20224.06 (kernelCPU 5.76 and userCPU 20218.30) respectively. In the case of final combination of input values, ***st*** = 0.6 and ***depth*** = 5, in total, 40542 clusters were identified within 23 hours and 17 minutes, where ***Drain3*** used 83606.47 of total CPU time (kernelCPU 128.11 and userCPU

83478.36) and 3782560 KB of memory. On the other hand, when ***Drain3*** was tested under the second scenario, in the case of input values of ***st*** to be 0.3 and ***depth*** to be 6, 1719 clusters were generated in an hour 4 minutes and 22 seconds, in which, the total CPU time of 3850.86 (kernelCPU 2.69 and userCPU 3848.17) and memory of 22656 KB were consumed. Then, with the second combination of input values of ***st*** to be 0.4 and ***depth*** to be 4, ***Drain3*** utilized 20323.14 of total CPU time (kernelCPU 194.74 and userCPU 20128.40) and 24272 KB of memory, likewise, generated 4016 clusters within approximately 5 hours 39 minutes and 40 seconds. At the end, when the author set the input values ***st*** to be equal to 0.6 and ***depth*** to be equal to 5, it took about 23 hours 5 minutes and 3 seconds to produce 40542 clusters. In terms of the overall CPU time and memory usages, the obtained results were 82881.63 (kernelCPU 101.82 and userCPU 82779.81) and 50304 KB respectively.

For the final dataset **xs19-syslog.log**, when ***Drain3*** was executed under the first scenario, with input values of ***st*** to be 0.3 and ***depth*** to be 6, in total, 8347 clusters were identified within 13 minutes and 46 seconds. The tool consumed 6714168 KB of memory and 837.95 of total CPU time (kernelCPU 6.98 and userCPU 830.97). When same combination of input values was tested under the second scenario, same number of clusters were produced in 14 minutes and 2 seconds, where the memory and CPU time consumptions of ***Drain3*** were 31372 KB and 840.34 (kernelCPU 2.73 and userCPU 837.61) respectively. With parameter values of ***st*** to be equal to 0.4 and ***depth*** to be 4, in the case of scenario 1, it took 2 hours 34 minutes and 57 seconds for ***Drain3*** to generate 4962 clusters. Likewise, the memory and CPU time consumptions ended up being 6709604 KB and 9282.77 (kernelCPU 10.72 and userCPU 9272.05) respectively. In the case of scenario 2, however, 4962 clusters were generated in 2 hours 39 minutes and 4 seconds, in which, the tool utilized 9512.93 of CPU time (kernelCPU 6.66 and userCPU 9506.27) and 25952 KB of memory. Finally, when the author tested ***Drain3*** with input values of ***st*** to be 0.6 and ***depth*** to be 5 under the first scenario, by consuming 6739196 KB of memory and 4953.48 of CPU time (kernelCPU 13.11 and userCPU 4940.37), the tool produced 34328 clusters in 1 hour 22 minutes and 33 seconds. For the scenario 2, ***Drain3*** used 4906.79 of CPU time (kernelCPU 7.50 and userCPU 4899.29) and 56220 KB of memory, where it identified same number of clusters, 34328, within 1 hour and 22 minutes.

	st = 0.3, ***depth*** = 6	***st*** = 0.4, ***depth*** = 4	***st*** = 0.6, ***depth*** = 5
http-apacheformat.log	memory 49076 KB, CPU Time 2.65 (kernelCPU 0.05 + userCPU 2.60)	memory 48836 KB, CPU Time 3.07 (kernelCPU 0.08 + userCPU 2.99)	memory 49512 KB, CPU Time 5.82 (kernelCPU 0.05 + userCPU 5.77)
windows-text.log	memory 3688412 KB, CPU Time 233.93 (kernelCPU 3.70 + userCPU 230.23)	memory 3689744 KB, CPU Time 429.95 (kernelCPU 5.70 + userCPU 424.25)	memory 3691880 KB, CPU Time 436.98 (kernelCPU 4.83 + userCPU 432.15)
logsrv.log.2	memory 3753124 KB, CPU Time 3656.39 (kernelCPU 12.49 + userCPU 3656.39)	memory 3755736 KB, CPU Time 20224.06 (kernelCPU 5.76 + userCPU 20218.30)	memory 3782560 KB, CPU Time 83606.47 (kernelCPU 128.11 + userCPU 83478.36)
xs19-syslog.log	memory 6714168 KB, CPU Time 837.95 (kernelCPU 6.98 + userCPU 830.97)	memory 6709604 KB, CPU Time 9282.77 (kernelCPU 10.72 + userCPU 9272.05)	memory 6739196 KB, CPU Time 4953.48 (kernelCPU 13.11 + userCPU 4940.37)

Table 14. Performance Test Results of Drain3 when Entire Log File was Loaded into Memory (Scenario 1)

	st = 0.3, ***depth*** = 6	***st*** = 0.4, ***depth*** = 4	***st*** = 0.6, ***depth*** = 5
http-apacheformat.log	memory 20984 KB, CPU Time 2.65 (kernelCPU 0.02 + userCPU 2.63)	memory 21096 KB, CPU Time 3.03 (kernelCPU 0.02 + userCPU 3.01)	memory 21728 KB, CPU Time 5.64 (kernelCPU 0.03 + userCPU 5.61)
windows-text.log	memory 36364 KB,	memory 36544 KB,	memory 38208 KB,

	CPU Time 235.17 (kernelCPU 1.70 + userCPU 233.47)	CPU Time 415.54 (kernelCPU 2.60 + userCPU 412.94)	CPU Time 434.65 (kernelCPU 2.49 + userCPU 432.16)
logsrv.log.2	memory 22656 KB, CPU Time 3850.86 (kernelCPU 2.69 + userCPU 3848.17)	memory 24272 KB, CPU Time 20323.14 (kernelCPU 194.74 + userCPU 20128.40)	memory 50304 KB, CPU Time 82881.63 (kernelCPU 101.82 + userCPU 82779.81)
xs19-syslog.log	memory 31372 KB, CPU Time 840.34 (kernelCPU 2.73 + userCPU 837.61)	memory 25952 KB, CPU Time 9512.93 (kernelCPU 6.66 + userCPU 9506.27)	memory 56220 KB, CPU Time 4906.79 (kernelCPU 7.50 + userCPU 4899.29)

Table 15. Performance Test Results of Drain3 when Log Messages were Read Line by Line (Scenario 2)

AEL

The author provides the summary of the performance testing outcomes of ***AEL,*** likewise, the tested input values in **Table 16**.

When ***AEL*** was executed to parse **http-apacheformat.log** dataset, overall, four distinct combination of input values were tested. The first combination of input values was ***mineventcount*** to be 2 and ***mergepercent*** to be 0.3. The tool managed to complete the parsing of the logs in 22 seconds and identified 638 templates. Additionally, the total memory and CPU time consumptions were 199852 KB and 25.37 (kernelCPU 1.86 and userCPU 23.51) respectively. Results were slightly different when the value of ***mineventcount*** remained same, but ***mergepercent*** was changed to 1. This time, 30 templates were produced within 20 seconds, where ***AEL*** consumed 218040 KB of memory and 23.61 of total CPU time (kernelCPU 1.86 and userCPU 21.75). When the author executed ***AEL*** with the third combination of input values, ***mineventcount*** to be 10 and ***mergepercent*** to be 0.3, it took overall 22 seconds for the tool to finalize the parsing of log messages and generate 655 templates. Likewise, the memory and CPU time consumptions of the tool were 199880 KB and 25.69 (kernelCPU 1.81 and userCPU 23.88) respectively. Finally, when the author changed the value of ***mergepercent*** to 0.8 and left ***mineventcount*** unchanged, ***AEL*** finished the parsing of logs in 21 seconds, in which it identified 79 templates. In terms of the memory and CPU time consumptions, the results were 217132 KB of memory and 24.15 of CPU time (kernelCPU 1.78 and userCPU 22.37).

Regarding the parsing of **windows-text.log**, **logsrv.log.2** and **xs19-syslog.log** datasets, none of the tested combinations of input values helped ***AEL*** to complete the mining of logs within the reasonable amount of time, which was 36 hours.

	mineventcount = 2, mergepercent = 0.3	**mineventcount = 2, mergepercent = 1**	**mineventcount = 10, mergepercent = 0.3**	**mineventcount = 10, mergepercent = 0.8**
http-apacheformat.log	memory 199852 KB, CPU Time 25.37 (kernelCPU 1.86 + userCPU 23.51)	memory 218040 KB, CPU Time 23.61 (kernelCPU 1.86 + userCPU 21.75)	memory 199880 KB, CPU Time 25.69 (kernelCPU 1.81 + userCPU 23.88)	memory 217132 KB, CPU Time 24.15 (kernelCPU 1.78 + userCPU 22.37)
windows-text.log	<no results>		<no results>	<no results>
logsrv.log.2		<no results>	<no results>	<no results>
xs19-syslog.log	<no results>		<no results>	<no results>

Table 16. Performance Test Results of AEL

LogSig

When ***LogSig*** was tested to parse the logs, only in the case of **http-apacheformat.log** dataset, the tool managed to finalize the parsing process. For other three large-sized datasets, **windows-text.log, logsrv.log.2** and **xs19-syslog.log**, after certain period of time from the start of the execution, the log parsing tool crashed every time on each tested input value. The detailed information about the performance testing results and tested input parameters are provided in the **Table 17**.

In general, the author tried three or four input values for testing ***LogSig***. First, while parsing **http-apacheformat.log** dataset, he set ***groupNum*** parameter to be equal to 30, where ***LogSig*** identified 31 templates in about 16 minutes and 10 seconds. For this case, the tool consumed 669892 KB of memory and 970.96 of total CPU time (kernelCPU 2.39 and userCPU 968.57). The secondly tested input value was ***groupNum*** to be 20, in which, within roughly 9 minutes and 37 seconds, the tool finished the parsing process and generated 21 templates. In total, the memory consumption was 664644 KB, and total CPU time was 579.58 (kernelCPU 2.47 and userCPU 577.11). When ***groupNum*** was set to be 10, it took 4 minutes and 28 seconds for ***LogSig*** to complete the parsing and produce 11 identified templates, where the tool consumed 660064 KB of memory and 271.38 of total CPU time (kernelCPU 2.58 and userCPU 268.80). Finally, when the author set the value of ***groupNum*** to be 5, overall, 6 templates were identified within 3 minutes and 11 seconds. This time, the memory and total CPU time consumptions of ***LogSig*** were 649852 KB and 194.37 (kernelCPU 2.56 and userCPU 191.81) respectively.

When being tested on **windows-text.log** dataset, regardless of the tested input values such as ***groupNum***=30, ***groupNum***=20 or ***groupNum***=10, ***LogSig*** crashed every time. The primary reason for this was that in each case, the memory consumption of the tool reached the maximum available memory on the machine. In the case of ***groupNum*** to be 30, the tool crashed after consuming 64738668 KB of memory and 1028.39 of total CPU time (kernelCPU 103.26 and userCPU 925.13). On the other hand, when ***groupNum*** was set to be 20, the tool crashed after 64730832 KB of memory and 1039.19 of CPU time (kernelCPU 117.44 and userCPU 921.75) consumptions. Finally, when the author set ***groupNum*** to be 10, after using memory of 64728916 KB and CPU time of 1086.95 (kernelCPU 158.84 and userCPU 928.11), ***LogSig*** crashed.

As it was stated above, after certain period of time from the start of the execution, ***LogSig*** crashed on each tested input value while parsing **logsrv.log.2** dataset as well. The reason was same as in the case of **windows-text.log** dataset, in other words, the tool crashed due to consuming the maximum available memory on the experiment server. For instance, when ***groupNum*** was set to 30, ***LogSig*** crashed after consuming 64766840 KB of memory and 93342.24 of total CPU time (kernelCPU 413.60 and userCPU 92928.64). When ***groupNum*** was set to 20, the tool crashed when it utilized the memory of 64763224 KB and CPU time of 92811.54 (kernelCPU 482.13 and userCPU 92329.41). Finally, when the author changed the value of ***groupNum*** to 10, the overall memory and CPU time consumptions were 64804468 KB and 93472.53 (kernelCPU 570.63 and userCPU 92901.90) respectively, in which, ***LogSig*** crashed one more time.

Similarly, as in the case of other large-sized datasets, while parsing **xs19-syslog.log**, ***LogSig*** crashed in each tested input value. For instance, with ***groupNum*** to be equal to 30, after consuming the memory of 64792384 KB and CPU time of 4767.77 (kernelCPU 293.88 and userCPU 4473.89), the tool crashed. For ***groupNum*** value of 20, ***LogSig*** crashed after utilizing 64804288 KB of memory and 4691.51 of total CPU time (kernelCPU 272.06 and userCPU 4419.45). The last tested input value was ***groupNum*** to be equal to 10, in which, after the consumption of 64785508 KB of memory and 4662.98 of total CPU time (kernelCPU 229.19 and userCPU 4433.79), ***LogSig*** crashed.

	groupNum = 5	**groupNum = 10**	**groupNum = 20**	**groupNum = 30**
http-apacheformat.log	memory 649852 KB, CPU Time 194.37 (kernelCPU 2.56 + userCPU 191.81)	memory 660064 KB, CPU Time 271.38 (kernelCPU 2.58 + userCPU 268.80)	memory 664644 KB, CPU Time 579.58 (kernelCPU 2.47 + userCPU 577.11)	memory 669892 KB, CPU Time 970.96 (kernelCPU 2.39 + userCPU 968.57)
windows-text.log		**<tool crashed>** memory 64728916 KB, CPU Time 1086.95 (kernelCPU 158.84 + userCPU 928.11)	**<tool crashed>** memory 64730832 KB, CPU Time 1039.19 (kernelCPU 117.44 + userCPU 921.75)	**<tool crashed>** memory 64738668 KB, CPU Time 1028.39 (kernelCPU 103.26 + userCPU 925.13)
logsrv.log.2		**<tool crashed>** memory 64804468 KB, CPU Time 93472.53 (kernelCPU 570.63 + userCPU 92901.90)	**<tool crashed>** memory 64763224 KB, CPU Time 92811.54 (kernelCPU 482.13 + userCPU 92329.41)	**<tool crashed>** memory 64766840 KB, CPU Time 93342.24 (kernelCPU 413.60 + userCPU 92928.64)

xs19-syslog.log		**<tool crashed>** memory 64785508 KB, CPU Time 4662.98 (kernelCPU 229.19 + userCPU 4433.79)	**<tool crashed>** memory 64804288 KB, CPU Time 4691.51 (kernelCPU 272.06 + userCPU 4419.45)	**<tool crashed>** memory 64792384 KB, CPU Time 4767.77 (kernelCPU 293.88 + userCPU 4473.89)

Table 17. Performance Test Results of LogSig

LenMa

In the **Table 18**, along with the performance testing results of ***LenMa***, the author also provided the tested input values during the experiments.

In order to parse **http-apacheformat.log** dataset with ***LenMa***, the first tested value of input parameter ***threshold*** was set to 0.1, in which, it took 2 minutes and 20 seconds to complete parsing of logs and produce 426 templates. Furthermore, the tool consumed 174308 KB of memory and 145.99 of total CPU time (kernelCPU 3.24 and userCPU 142.75). Results were almost identical when ***threshold*** value was set to 0.5. This time, within 2 minutes and 49 seconds, ***LenMa*** finalized mining of logs and identified 434 templates. In terms of the memory and CPU time consumptions, the tool used 173896 KB of memory and 174.99 of CPU time (kernelCPU 3.25 and userCPU 171.74). Comparatively worse results were achieved when ***threshold*** was equal to 1, where, in total, 4639 templates were generated for 5 hours 51 minutes and 4 seconds. Additionally, even though, the utilized memory was similar to previous outcomes, which was 173912 KB of memory, the overall CPU time consumption was 20980.8 (kernelCPU 15.05 and userCPU 20965.75).

In the case of **windows-text.log** dataset, ***LenMa*** managed to produce results within reasonable time when ***threshold*** value was set to 0.1 and 0.5. For ***threshold*** to be equal to 1, no outcomes were obtained. Overall, it took for the tool 2 hours and 44 minutes to parse the logs when ***threshold*** was 0.1 and identify 910 templates, where the memory and total CPU time consumptions were 8296304 KB and 9806.97 (kernelCPU 16.88 and userCPU 9790.09) respectively. On the other hand, with the ***threshold*** value of 0.5, ***LenMa*** finalized parsing in 3 hours 54 minutes and

18 seconds by producing 1051 templates. This time, the total consumed memory was 8296064 KB, and CPU time was 14005.13 (kernelCPU 16.78 and userCPU 13988.35).

Regardless of the tested ***threshold*** values, ***LenMa*** did not manage to parse the logs in reasonable timeframe (nearly 36 hours) while parsing **logsrv.log.2** and **xs19-syslog.log** datasets.

	threshold = 0.1	**threshold = 0.5**	**threshold = 1**
http-apacheformat.log	memory 174308 KB, CPU Time 145.99 (kernelCPU 3.24 + userCPU 142.75)	memory 173896 KB, CPU Time 174.99 (kernelCPU 3.25 + userCPU 171.74)	memory 173912 KB, CPU Time 20980.8 (kernelCPU 15.05 + userCPU 20965.75)
windows-text.log	memory 8296304 KB, CPU Time 9806.97 (kernelCPU 16.88 + userCPU 9790.09)	memory 8296064 KB, CPU Time 14005.13 (kernelCPU 16.78 + userCPU 13988.35)	<no results>
logsrv.log.2	<no results>	<no results>	<no results>
xs19-syslog.log	<no results>	<no results>	<no results>

Table 18. Performance Test Results of LenMa

LogCluster

While testing ***LogCluster*** on the performance testing datasets, two different scenarios were utilized. The first scenario was to conduct experiments with ***--wsize***=100000 option, which would make an extra pass over the entire logfile for memory saving purposes. While this option would allow to save significant amounts of memory, since the experiment server had more than enough memory for successful completion of all experiments, the second scenario was realized as well, where the author omitted ***--wsize*** option while running the tool, which also helped to save a lot of CPU time. The summary of the performance testing experiments with ***LogCluster*** can be found in the **Tables 19** and **20**.

In the case of **http-apacheformat.log** dataset, first, the author set ***rsupport*** threshold value to be equal to 0.1. With ***wsize*** option being enabled, the tool identified 152 clusters within 7 seconds, where it consumed 33741 KB of memory along with 7.7 of total CPU time (kernelCPU 0.05 and userCPU 7.65). On the other hand, when ***wsize*** parameter was missing, ***LogCluster*** identified 152 clusters in 4 seconds by consuming 39016 KB of memory and 3.99 of CPU time (kernelCPU 0.02 and userCPU 3.97). Next, the ***rsupport*** value was changed to 0.5, where ***LogCluster*** produced 35 clusters in 8 seconds with ***wsize*** parameter. Overall, the memory and CPU time consumptions ended up being 21609 KB and 7.6 (kernelCPU 0.04 and userCPU 7.56) respectively. However, without ***wsize*** option, 35 clusters were produced within 3 seconds, in which, the memory and CPU time consumptions of ***LogCluster*** were 34152 KB and 3.94 (kernelCPU 0.03 and userCPU 3.91) respectively. Finally, when the author set the value of ***rsupport*** to be 1, in the case of enabled ***wsize*** parameter, in total, it took for the tool around 7 seconds to complete the parsing and identify 16 clusters, where ***LogCluster*** consumed 19213 KB of memory and 7.49 of total CPU time (kernelCPU 0.03 and userCPU 7.46). On the other hand, parsing and identification of 16 clusters were completed in 4 seconds with 34084 KB of memory and 3.82 of CPU time (kernelCPU 0.04 and userCPU 3.78) consumptions, when ***wsize*** was omitted.

The memory and total CPU time consumptions of ***LogCluster*** were 2626989 KB and 1283.24 (kernelCPU 5.64 and userCPU 1277.6) respectively when the author used ***wsize*** option and set ***rsupport*** to be 0.1 while parsing **windows-text.log** dataset. Likewise, the tool managed to complete the parsing process in 21 minutes and 29 seconds, in which it identified 98 clusters. However, when ***wsize*** was omitted, there were significant improvements in the performance of the tool, in other words, it generated 99 clusters in 12 minutes and 40 seconds by consuming 2800664 KB of memory and 758.42 of CPU time (kernelCPU 5.90 and userCPU 752.52). The second tested value of ***rsupport*** was 0.5. This time, in the case of ***wsize*** being utilized, the tool generated 43 clusters in 21 minutes and 19 seconds, where it consumed 764704 KB of memory and 1279.37 of CPU time (kernelCPU 5.44 and userCPU 1273.93). Again, the overall time taken to parse the logs and identify 43 clusters was much lower, 12 minutes and 43 seconds, when ***wsize*** was not used. Furthermore, the memory and CPU time consumptions of the tool were 947672 KB and 761.28 (kernelCPU 3.31 and userCPU 757.97) respectively. As a final input value, the author set ***rsupport*** to be 1, where along with ***wsize*** option being enabled, it took 21 minutes and 12 seconds for

LogCluster to produce 21 clusters. In terms of the memory and CPU time consumptions, the results were 675456 KB and 1262.13 (kernelCPU 4.50 and userCPU 1257.63) respectively. On the other hand, when ***wsize*** was disabled, ***LogClusters*** consumed 863828 KB of memory and 739.22 of CPU time (kernelCPU 4.08 and userCPU 735.14), likewise, completed the parsing process in 12 minutes and 27 seconds by identifying 21 clusters.

With ***wsize*** being enabled and ***rsupport*** to be 0.1, in total, it took 22 minutes and 40 seconds for ***LogCluster*** to complete the parsing of **logsrv.log.2** dataset, as a result of which, the tool generated 196 clusters. Moreover, the tool consumed 165116 KB of memory and 1368.88 of CPU time (kernelCPU 4.12 and userCPU 1364.76). When the author changed the value of ***rsupport*** to 0.5, the log parsing tool managed to finalize the mining of logs in 22 minutes and 19 seconds, where 20 clusters were identified. In terms of the memory and CPU time consumptions, the achieved results were 64193 KB and 1351.79 (kernelCPU 4.89 and userCPU 1346.9) respectively. For the final input value of ***rsupport*** to be 1, ***LogCluster*** produced 3 clusters within 22 minutes and 18 seconds, where it consumed 32676 KB of memory and 1330.16 of total CPU time (kernelCPU 4.43 and userCPU 1325.73). In contrast, when ***wsize*** was omitted, for ***rsupport*** to be 0.1, it took for the tool 11 minutes and 40 seconds to identify 196 clusters, in which, it utilized 1957376 KB of memory and 698.65 of total CPU time (kernelCPU 5.53 and userCPU 693.12). For ***rsupport*** value of 0.5, ***LogCluster*** finished the parsing of logs in 11 minutes and 24 seconds and produced 20 clusters. Additionally, 1956760 KB of memory and 681.92 of CPU time (kernelCPU 6.39 and userCPU 675.53) were used during the parsing. The tool identified 3 clusters within 11 minutes and 8 seconds when ***rsupport*** was set to 1, where the memory and CPU time consumptions were 1956788 KB and 666.51 (kernelCPU 4.88 and userCPU 661.63) respectively.

For the final dataset, **xs19-syslog.log**, with ***rsupport*** value of 0.1 and ***wsize*** parameter being enabled during the execution, by consuming 76088 KB of memory and 1872.55 of CPU time (kernelCPU 7.38 and userCPU 1865.17), ***LogCluster*** identified 295 clusters in 31 minutes and 18 seconds. When ***wsize*** was omitted, the tool produced same number of clusters in 16 minutes and 35 seconds, where the memory and CPU time consumptions were 5704668 KB and 992.71 (kernelCPU 8.99 and userCPU 983.72) respectively. When ***LogCluster*** was tested with ***rsupport*** value of 0.5, in the case of ***wsize*** parameter being enabled, the tool produced 31 clusters within 31 minutes and 10 seconds, likewise, it utilized 1863.9 of CPU time (kernelCPU 7.32 and userCPU 1856.58) and 33428 KB of memory. On the other hand, with ***wsize*** being disabled, it took 16 minutes and 11 seconds for the tool to generate 31 clusters, in which, it used the memory of 5704612 KB and CPU time of 969.04 (kernelCPU 8.92 and userCPU 960.12). The last tested value of ***rsupport*** was equal to 1. ***LogCluster*** consumed 29036 KB of memory and 1831.58 of total CPU time (kernelCPU 7.53 and userCPU 1824.05) to identify 17 clusters in 30 minutes and 36 seconds when it was executed with ***wsize*** option being enabled. When the author disabled the given option, same number of clusters, 17, were produced within 15 minutes and 56 seconds, where the tool utilized 953.28 of CPU time (kernelCPU 9.23 and userCPU 944.05) and the memory of 5704588 KB.

	rsupport=0.1	**rsupport=0.5**	**rsupport=1**
http-apacheformat.log	memory 33741 KB, CPU Time 7.7 (kernelCPU 0.05 + userCPU 7.65)	memory 21609 KB, CPU Time 7.6 (kernelCPU 0.04 + userCPU 7.56)	memory 19213 KB, CPU Time 7.49 (kernelCPU 0.03 + userCPU 7.46)
windows-text.log	memory 2626989 KB, CPU Time 1283.24 (kernelCPU 5.64 + userCPU 1277.6)	memory 764704 KB, CPU Time 1279.37 (kernelCPU 5.44 + userCPU 1273.93)	memory 675456 KB, CPU Time 1262.13 (kernelCPU 4.50 + userCPU 1257.63)
logsrv.log.2	memory 165116 KB, CPU Time 1368.88 (kernelCPU 4.12 + userCPU 1364.76)	memory 64193 KB, CPU Time 1351.79 (kernelCPU 4.89 + userCPU 1346.9)	memory 32676 KB, CPU Time 1330.16 (kernelCPU 4.43 + userCPU 1325.73)
xs19-syslog.log	memory 76088 KB, CPU Time 1872.55 (kernelCPU 7.38 + userCPU 1865.17)	memory 33428 KB, CPU Time 1863.9 (kernelCPU 7.32 + userCPU 1856.58)	memory 29036 KB, CPU Time 1831.58 (kernelCPU 7.53 + userCPU 1824.05)

Table 19. Performance Test Results of LogCluster with ***wsize*** parameter

	rsupport=0.1	**rsupport=0.5**	**rsupport=1**
http-apacheformat.log	memory 39016 KB, CPU Time 3.99 (kernelCPU 0.02 + userCPU 3.97)	memory 34152 KB, CPU Time 3.94 (kernelCPU 0.03 + userCPU 3.91)	memory 34084 KB, CPU Time 3.82 (kernelCPU 0.04 + userCPU 3.78)
windows-text.log	memory 2800664 KB, CPU Time 758.42 (kernelCPU 5.90 + userCPU 752.52)	memory 947672 KB, CPU Time 761.28 (kernelCPU 3.31 + userCPU 757.97)	memory 863828 KB, CPU Time 739.22 (kernelCPU 4.08 + userCPU 735.14)
logsrv.log.2	memory 1957376 KB, CPU Time 698.65 (kernelCPU 5.53 + userCPU 693.12)	memory 1956760 KB, CPU Time 681.92 (kernelCPU 6.39 + userCPU 675.53)	memory 1956788 KB, CPU Time 666.51 (kernelCPU 4.88 + userCPU 661.63)
xs19-syslog.log	memory 5704668 KB, CPU Time 992.71 (kernelCPU 8.99 + userCPU 983.72)	memory 5704612 KB, CPU Time 969.04 (kernelCPU 8.92 + userCPU 960.12)	memory 5704588 KB, CPU Time 953.28 (kernelCPU 9.23 + userCPU 944.05)

Table 20. Performance Test Results of LogCluster without **wsize** parameter

3.5.4 Summary of Performance Testing

In this subchapter, the author tested the log mining algorithms with various input values on different datasets in order to find out the efficiency of the given algorithms. Depending on the size of the dataset, some algorithms managed to complete the parsing successfully, while others failed to parse the log messages within a reasonable amount of time, which was 36 hours. Additionally, there were cases when a particular tool crashed while mining the logs.

According to the results of performance testing experiments, one important finding is that many algorithms required too much CPU time when processing larger datasets, however, they still were not able to complete the parsing within 36 hours. LogSig, AEL, Spell and IPLoM can be examples of such algorithms. On the other hand, in the paper from University of Honk Kong [8], while measuring the efficiency of log mining algorithms, where the efficiency was defined as the running time of an algorithm required to finalize the parsing process, the authors claimed that IPLoM had a better efficiency, which scaled linearly with the log file size. To be more specific, a log file, which has a size of 1 GB, can be parsed by IPLoM within 10 minutes. Likewise, except for BGL data, AEL was also reported to achieve higher efficiency during the experiments.

The second important finding is that many tools consumed too much memory and even crashed during the parsing process. Even though in the research paper [8], it was stated that their experiments were conducted on a server with 62GB of memory, which is almost similar to the one utilized in this thesis, the authors did not perform any memory usage analysis of log mining algorithms. However, with a similarly equipped server, it is highly probable that certain log parsing algorithms will crash, if the experiments are conducted with real-life larger datasets.

Finally, in terms of total CPU time, some algorithms were obviously faster than the others while parsing the log files. For instance, the two fastest algorithms were Drain3 and LogCluster. In contrast, for other algorithms, either it took too long to complete the parsing, or the algorithm did not manage to complete the mining of log messages at all.

When comparing Drain3 and LogCluster side by side, it can be observed that Drain3 is much more sensitive to input parameter settings on large log files. For example, for **xs19-syslog.log** dataset, the CPU time consumption ranged from 837.95 to 9282.77 seconds, and for **logsrv.2.log** dataset from 3656.93 to 82606.47 seconds (see **Table 14**). In contrast, LogCluster was much more stable in terms of consumed CPU time, spending from 953.29 to 992.71 seconds on **xs19-syslog.log** dataset, and from 666.51 to 698.65 seconds on **logsrv.2.log** dataset (see **Table 20**).

Summary of the main findings obtained as a result of the performance testing experiments:

- Many algorithms required too much CPU time when processing larger datasets, however, they still were not able to complete the parsing within 36 hours.
- Many algorithms consumed too much memory and even crashed during parsing process.

- Two log mining algorithms, specifically Drain3 and LogCluster, outperformed other tested algorithms in terms of total CPU time, in other words, completed parsing of logs faster than others.

Chapter 4. Conclusion

Over the last couple years, many researchers have been working to find solutions for the challenges related to log mining process and the associated log parsing algorithms. In addition, there have been many studies, the main focus of which were the comparison of these log mining/parsing algorithms with each other to find out the most effective ones that were also producing much more accurate outputs. However, after analyzing the past academic research documents related to the automatic log parsing algorithms and their comparisons, the author of the thesis found out that these studies have serious drawbacks in their analysis such as:

- Too little attention was paid to efficiency, in which processing speed of log mining algorithms would be compared to each other to find out how much CPU time and memory resources they consume.
- Many of these research studies did not measure the quality of produced output at all.
- The research studies, which measured the quality and performance of log parsing algorithms, missed some important aspects in their papers. For instance, the study [8] did not measure the memory consumption of algorithms, which plays a crucial role in real-life log parsing scenarios.
- The studies, which measured the quality and performance of log mining tools, utilized bad input parameter settings, which renders conducted experiments unfair.
- The datasets used during the quality and performance testing experiments, were preprocessed, which is not a realistic log file analysis scenario and is also distorting the performance results.

Eventually, the concentration area of this thesis was the comparative analysis of log parsing algorithms and measurement of their efficiency and pattern detection rates. In other words, the primary contribution of this thesis was conducting fair and clear experiments in order to address the above-mentioned drawbacks.

As it was stated earlier in the thesis, one of the key factors, which makes this thesis different than other studies, is that while executing the log mining algorithms to parse the log messages, the author of the thesis consulted with the authors of the tested log parsing algorithms to find out their optimal input values that would enable the algorithms to identify rarely occurring log lines, thus, generate output with much higher quality in a more efficient way. Compared to other similar research studies, another main distinction in this thesis paper is the analysis of the memory consumption of log parsing algorithms during the performance testing experiments.

Overall, the author conducted experiments consisting of two scenarios, where he compared the log mining algorithms with each other.

In the case of first scenario, he measured and compared the pattern detection rates of log mining algorithms while parsing the quality testing datasets. While, in general, the algorithms achieved relatively high-quality results, for most algorithms, the incapability to parse the rarely occurring log messages, likewise, the issues related to the design and implementation of the algorithm affected the quality of produced outputs by decreasing the total pattern detection rates. The main findings that the author obtained from the quality testing experiments are:

- It is important to design log mining algorithms to detect patterns, where the number of words in corresponding log messages is not necessarily constant. Thus, log parsing algorithms like Drain, IPLoM, Lenma and AEL, due to their designs, are not suitable for all log types.
- Selection of input values can have significant effect on quality of produced outcome and usage of right parameter values can noticeably improve result.

For the second scenario, the author executed the log parsing algorithms to parse the performance testing datasets, where he measured the memory and total CPU time consumptions of a particular algorithm. Depending on the tested input values and the size of the performance testing dataset, while some algorithms successfully finalized the parsing process in a reasonable amount of time, 36 hours, others either failed to parse or even crashed during the process. Additionally, the efficiency of tested log mining algorithms varied a lot since some of them consumed a lot of memory and CPU time resources to complete the mining of logs, while others accomplished the task in a much more efficient way by consuming less resources.

In terms of the most important findings obtained as a result of the performance testing experiments, the author provided the following points:

- Many algorithms required too much CPU time when processing larger datasets, however, they still were not able to complete the parsing within 36 hours.
- Many algorithms consumed too much memory and even crashed during parsing process.
- Two log mining algorithms, specifically Drain3 and LogCluster, outperformed other tested algorithms in terms of total CPU time, in other words, completed parsing of logs faster than others. However, in comparison with the LogCluster, Drain3 is much more sensitive to input parameter settings on larger log files since it consumed significantly more CPU time in a number of cases.

References

[1] Harjunkoski I., Isaksson A. J., Sand G., “The impact of digitalization on the future control and operations”, Computer & Chemical Engineering, 114, p. 122 - 129, 2018.

[2] P. He, J. Zhu, Z. Zheng and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," 2017 IEEE International Conference on Web Services (ICWS), pp. 33-40, 2017.

[3] Tang, Liang & Li, Tao & Perng, Chang-Shing, “LogSig: Generating system events from raw textual logs”, pp. 785-794, 2011.

[4] R. Vaarandi and M. Pihelgas, "LogCluster - A data clustering and pattern mining algorithm for event logs," 2015 11th International Conference on Network and Service Management (CNSM), pp. 1-7, 2015.

[5] Adetokunbo A.O. 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 (KDD '09), New York, NY, USA, pp. 1255–1264, 2009.

[6] J. Stearley, "Towards informatic analysis of syslogs," 2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935), pp. 309-318, 2004.

[7] Vaarandi, Risto, “Mining event logs with SLCT and LogHound”, pp. 1071 – 1074, 2008.

[8] Jieming Zhu, Shilin He, Jinyang Liu, Pinjia He, Qi Xie, Zibin Zheng, and Michael R. Lyu., “Tools and benchmarks for automated log parsing.”, In Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19), IEEE Press, pp. 121–130, 2019.

[9] Q. Fu, J. Lou, Y. Wang and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," *2009 Ninth IEEE International Conference on Data Mining*, pp. 149-158, 2009.

[10] M. Du and F. Li, "Spell: Streaming Parsing of System Event Logs," *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pp. 859-864, 2016.

[11] Jiawei Han, Jian Pei, and Yiwen Yin, “Mining frequent patterns without candidate generation”, SIGMOD Rec. 29, 2 (June 2000), pp. 1–12, 2000.

[12] M. Mizutani, "Incremental Mining of System Log Format," *2013 IEEE International Conference on Services Computing*, pp. 595-602, 2013.

[13] Z. M. Jiang, A. E. Hassan, P. Flora and G Hamann, “Abstracting Execution Logs to Execution Events for Enterprise Applications”, September 2008.

[14] Vaarandi, Risto. “A data clustering algorithm for mining patterns from event logs.”, *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003),* 119-126, 2003.

[15] Liang Tang, Tao Li, and Chang-Shing Perng, “LogSig: generating system events from raw textual logs.”, In Proceedings of the 20th ACM international conference on Information and knowledge management, Association for Computing Machinery, New York, NY, USA, 785–794, 2011.

[16] Shima, Keiichi, “Length Matters: Clustering System Log Messages using Length of Words”, 2016.

[17] Diana El-Masri, Fabio Petrillo, Yann-Gaël Guéhéneuc, Abdelwahab Hamou-Lhadj, Anas Bouziane, “A systematic literature review on automated log abstraction techniques”, Information and Software Technology, Volume 122, 2020.

[18] Landauer, Max & Skopik, Florian & Wurzenberger, Markus & Rauber, Andreas, “System Log Clustering Approaches for Cyber Security Applications: A Survey.”, Computers & Security, 2020.

[19] Pihelgas, Mauno, “Automating Defences against Cyber Operations in Computer Networks.”, Tallinn University of Technology (TalTech), 2021.

[20] GNU Time utility, Available Online: https://www.gnu.org/software/time/

[21] Risto Vaarandi, Markus Kont, and Mauno Pihelgas, “Event log analysis with the LogCluster tool”, MILCOM 2016 - 2016 IEEE Military Communications Conference, pp. 982 – 987, 2016.

[22] Vaarandi, Risto. “A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs.”, *INTELLCOMM*, 2004.

[23] J. Stearley, "Towards informatic analysis of syslogs," 2004 IEEE International Conference on Cluster Computing, pp. 309-318, 2004.

[24] Drain3, Available Online: https://github.com/IBM/Drain3

[25] LogCluster, Available Online: https://github.com/ristov/logcluster

[26] IPLoM, Available Online: https://github.com/logpai/logparser/tree/master/logparser/IPLoM

[27] Spell, Available Online: https://github.com/logpai/logparser/tree/master/logparser/Spell

[28] AEL, Available Online: https://github.com/logpai/logparser/tree/master/logparser/AEL

[29] LogSig, Available Online: https://github.com/logpai/logparser/tree/master/logparser/LogSig

[30] LenMa, Available Online: https://github.com/logpai/logparser/tree/master/logparser/LenMa

[31] Rafael Copstein, Jeff Schwartzentruber, Nur Zincir-Heywood, and Malcolm Heywood, "Log Abstraction for Information Security: Heuristics and Reproducibility", the 16th International Conference on Availability, Article 93, p. 1 – 10, 2021.