Comparative Analysis of Pattern Mining Algorithms for Event Logs

Orkhan Gasimov, Risto Vaarandi and Mauno Pihelgas

Comparative Analysis of Pattern Mining Algorithms for Event Logs

Orkhan Gasimov
CYBERS
Tallinn, Estonia
orkhan@cybers.eu

Risto Vaarandi
Tallinn University of Technology
Tallinn, Estonia
risto.vaarandi@ttu.ee

Mauno Pihelgas
Tallinn University of Technology
Tallinn, Estonia
maunopihelgas@gmail.com

***Abstract*—During the last two decades, the mining of message patterns from textual event logs has become an important security monitoring and system management task. A number of algorithms have been developed for that purpose, and recently several comparative studies of these algorithms have been published. However, existing studies have several drawbacks like the lack of performance evaluation on real-life data sets and the use of suboptimal settings for evaluated algorithms. This paper addresses these issues and evaluates commonly used log mining algorithms on a number of security and system event logs.**

***Index Terms*—pattern mining for event logs, event log analysis**

I. Introduction

Nowadays, organizational networks and IT systems are growing constantly and generating large volumes of events. For example, in a recent paper we described a SIEM installation where a single IDS appliance produced more than 1,000 events per second [1]. Although a number of event log management solutions and logging protocols have been developed that support structured logging, producing event log messages in unstructured and semi-structured formats is still widespread. For instance, many devices and applications are generating messages in traditional BSD *syslog* format which specifies a few message fields like timestamp and hostname, while the message text itself is a free-form string without any structure [2]. For example, Fig. 1 displays such messages from a network switch.

```
Port 1:16 link down
Port 3:43 link down
Port 1:12 link UP at speed 10 Mbps and half-duplex
Port 2:13 link UP at speed 1 Gbps and full-duplex
Port 3:14 link UP at speed (down) and half-duplex
```

Fig. 1. Example *link down* and *link up* messages from a network switch (timestamps and hostnames have been omitted for brevity and privacy).

The messages from Fig. 1 can be summarized with the following message patterns:

*Port * link down*

*Port * link UP at speed * and **

Note that each wildcard * matches one or more words, and such words are called *variable words* in the rest of this paper. Also, the words that match constant (i.e., non-wildcard) parts of the message pattern are called *constant words*.

The detection of such message patterns is an important data mining problem, with identified patterns being useful for several purposes like the development of event correlation rules and automated anomaly detection [3]–[5]. During the last two decades, a number of algorithms have been developed for mining message patterns from event logs [4]–[12]. The algorithms assume that event logs are files with textual content and each line in the log file represents one message. During event log processing, each message is split into words by a predefined delimiter (usually whitespace characters). The algorithms then attempt to identify variable words as precisely as possible in order to produce meaningful message patterns. Note that in some papers (e.g., [5]) message patterns are called templates, but for the sake of consistency the term *message pattern* is used in the rest of this paper.

Recently, several comparative studies have been published for message pattern mining algorithms [13]–[16]. However, past studies have several drawbacks. First, some studies were limited to comparing the features of pattern mining algorithms and did not evaluate the performance of algorithms on event log data sets [15], [16]. Also, some studies with performance evaluations were using suboptimal settings for algorithms, and were conducted on preprocessed data sets which do not represent the original log files from production systems [13].

This paper addresses the above shortcomings and evaluates the performance of message pattern mining algorithms on real-life security and system logs. The paper first measures the pattern detection rate of algorithms (the ability to identify meaningful patterns in log files), continuing with measuring the CPU time and memory consumption of algorithms when mining patterns from medium-sized and large log files.

The remainder of this paper is organized as follows – section II discusses related work, section III presents the experiments for measuring the performance of evaluated algorithms, and section IV concludes the paper.

II. Related Work

IPLoM is one of the earliest message pattern mining algorithms that is based on a divisive clustering method [7]. Initially, IPLoM considers all messages in an event log as members of a single cluster. During its first step, IPLoM splits the initial cluster by assigning messages with the same number of words to the same cluster. During the second step, each cluster is divided further by identifying the word position with the least number of unique words, and splitting the cluster by these unique words. During the third step, clusters are split based on associations between word pairs. Finally, a message pattern is derived for each cluster.

LogCluster is another message pattern mining algorithm that is based on a clustering method [4], [11]. LogCluster begins its work by identifying *frequent words* which appear in at least *s* event log lines, where *s* is the *support threshold* provided by the user. LogCluster then makes another pass over the event log, extracting frequent words from each event log line, and assigning the line to the cluster candidate that is identified by the sequence of extracted frequent words. During this process, a message pattern is maintained for each cluster candidate. Finally, cluster candidates which cover at least *s* event log lines are selected as clusters, and their message patterns are reported to the end user. Also, event log lines without a cluster are assigned to the *cluster of outliers* for identifying rare and potentially anomalous messages. Unlike IPLoM, LogCluster is able to generate a message pattern for similar messages even if they do not have the same number of words. For example, for *link up* messages from Fig. 1, LogCluster is able detect the pattern *Port* *{*1,1*} *link UP at speed* *{*1,2*} *and* *{*1,1*}, where the wildcard *{*1,1*} matches one variable word and the wildcard *{*1,2*} either one or two variable words.

The LenMa algorithm [10] assumes that the event log messages which correspond to the same message pattern contain the same number of words. For distinguishing constant words from variable words, the algorithm uses the following heuristic – the constant word has the same length for all messages that match the same pattern, while variable words are likely to have different lengths. For detecting the message patterns, LenMa builds a word length vector for each log message, joining log messages with the same number of words into the same cluster if the cosine similarity of their word length vectors is higher than user-given threshold. After a new message has been joined with the cluster, the message pattern of the cluster is updated.

The AEL algorithm [9] first applies several heuristics for detecting variable words in log messages. The algorithm then divides event log messages into bins, so that each bin contains the messages with the same number of words and variable words. Finally, one or more message patterns are derived for each bin.

Similarly to IPLoM, LenMa and AEL, the Drain algorithm [5] assumes that the messages corresponding to the same pattern must have the same number of words. Drain makes a single pass over the event log, using a parse tree with a fixed depth where message patterns are created and updated in leaf nodes. One of the main purposes of the parse tree is to limit the number of message patterns than need the inspection and update, so that for every event log message the patterns only in one leaf node would be processed.

The Spell algorithm [8] maintains longest common subsequences (LCSs) of words for event log messages while processing the event log. For each event log message, it is either merged with the most similar LCS object that involves an update of the object, or a new LCS object is created if the similarity with existing objects is below user-given threshold τ. With each LCS object, wildcards * are maintained which describe the location of variable words, so that each LCS object represents a message pattern.

LogSig [6] is an event log clustering algorithm which begins with converting each event log message into a set of word pairs from that message. Local search function is then used for finding an optimal partition of event log messages into clusters, so that messages in each cluster would share as much word pairs as possible. Finally, a message pattern is generated for messages in each cluster.

The work by Zhu et al. [13] was one of the earliest comparative studies of event log pattern mining algorithms, with its findings being presented in several later studies (e.g., [14], [15]). The work included an analytical comparison of evaluated algorithms and their accuracy assessment on 16 small data sets of 2,000 event log messages. Before the experiments, a number of preprocessing rules were applied to data sets for replacing variable parts in messages (such as IP addresses, numbers, etc.). The algorithms with the highest accuracy were then selected for further testing on three larger 1GB data sets. According to tests, Drain, IPLoM and AEL were the fastest algorithms, while other algorithms did not scale well to larger event logs. Data sets and tools used during the experiments were publicly released in LogPAI [17] and Loghub [18] repositories.

Copstein et al. [14] researched how well the patterns detected from security logs by different algorithms are reflecting a clear heuristic (e.g., whether a detected pattern reflects some network protocol). For establishing the heuristic for a pattern, constant words from the pattern were extracted and compared with regular expression based rules. Copstein et al. also re-executed accuracy related experiments from the work by Zhu et al. [13], using publicly available data sets and tools [17]. According to Copstein et al., significant differences of more than 10% were observed for the accuracy of several algorithms on some data sets [14]. While in most cases the number of such data sets was 1–3 out of 16, the largest number of differences was observed for LogSig, with experiments on five data sets producing a different result, and one experiment not completing in a reasonable amount of time. Given the deterministic nature of tested algorithms, the authors were unable to establish a clear reason for significant differences of over 10% in measured accuracy [14].

In a recent study [19], Pihelgas has pointed out several other shortcomings of the work by Zhu et al. [13]. First, the analytical comparison of evaluated algorithms provided several incorrect statements about the LogCluster algorithm. More importantly, LogCluster was evaluated with suboptimal input settings that lead to poor performance.

Other comparative studies of event log pattern mining algorithms include the works by El-Masri et al. [15] and Landauer et al. [16]. However, these studies were limited to comparing and discussing the features of event log mining algorithms, and they did not describe any experiments for evaluating the algorithms on event log data sets.

III. Performance Evaluation

A. Evaluation Setup

As discussed in section II, although the paper by Zhu et al. [13] offers interesting insights into event log pattern mining algorithms, some of its results have been challenged in recent works [14], [19]. When studying the experiments described in [13], we identified another subtle issue. Namely, before measuring the performance of log mining algorithms, the event logs were modified by applying a number of preprocessing rules. For some specific scenarios event log preprocessing can be both beneficial and straightforward – for example, *syslog* messages commonly begin with timestamps which makes them easy to identify and discard, provided that timestamps are not needed during the message pattern mining process. However, the preprocessing rules applied in [13] were more complex and largely targeted free-form event message texts that do not have any structure.

Unfortunately, writing such preprocessing rules assumes the previous domain knowledge about event message texts that the users often do not have, especially when log files are large and contain many different types of messages. In fact, the very purpose of event log mining algorithms is to discover such previously unknown knowledge. For example, before seeing the pattern *Port * link down*, the user might not be aware that some events in the event log originate from network switches and contain port identifiers (see Fig. 1).

More importantly, such preprocessing rules might oversimplify the pattern detection task for evaluated algorithms. For example, if the user has the previous domain knowledge about messages from Fig. 1 and uses a preprocessing rule for replacing all port identifiers with the string tag *PORTID*, the *link down* messages from Fig. 1 will no longer have any variable words and are summarized by the following pattern: *Port PORTID link down*. However, since the identification of variable words is essential for the pattern detection, such preprocessing can hide more complex pattern detection scenarios from evaluated algorithms, providing an unrealistic view about their real-life performance during comparative testing.

Another drawback of the experiments from [13] is the lack of memory consumption measurements for evaluated algorithms. However, excessive memory consumption might prevent the use of an event log pattern mining algorithm on a regular hardware that is available to a human analyst [19].

For the reasons above, performance evaluation experiments described in this paper have been conducted according to several key principles. First, in order to avoid badly chosen input settings for evaluated algorithms, we contacted the authors of the algorithms and followed their recommendations on proper settings. Second, we did not apply any kind of preprocessing to event log messages (such as replacing IP addresses with wildcards * in message texts). Finally, our experiments also included memory consumption measurements for evaluated algorithms.

We selected the following seven algorithms for a detailed evaluation – Drain, IPLoM, AEL, LenMa, Spell, LogCluster and LogSig. Drain, IPLoM, AEL, LenMa and Spell were selected as the algorithms with the highest accuracy according to both Zhu et al. [13] and Copstein et al. [14]. Also, we selected LogCluster and LogSig, because the results reported for them have been questioned in recent works [14], [19].

During the experiments, we used the original implementations of LogCluster, Drain and LenMa [20]–[22]. Unfortunately, other algorithms did not have publicly available implementations by their authors, and therefore we used the implementations by Zhu et al. from LogPAI repository [17]. When contacting the authors of evaluated algorithms, we received responses from the authors of IPLoM, LenMa, Drain, AEL and LogCluster, while the authors of Spell and LogSig did not reply to our emails.

For evaluating the algorithms, we used *syslog* log data collected from 149 Linux servers and network devices in Tallinn University of Technology, and log data collected from 75 Linux and Windows servers during Crossed Swords 2019 and 2020 (XS19 and XS20) cyber security exercises in NATO CCDCOE. Most of the data sets used in this study were security specific, and apart from the work by Copstein et al. [14], previous comparative studies have not focused on security event logs. Also, since several data sets were collected during cyber security exercises when systems were under stress, it allowed us to assess the performance of the algorithms on log data with many unexpected messages. All experiments were conducted on a physical server with two Intel Xeon E5-2630Lv2 CPUs (12 cores in total), 64GB of memory, and Samsung 860 EVO SSD drive with 250GB of size. The server was running Rocky Linux 8 as an operating system.

B. Quality of Detected Message Patterns

For evaluating the quality of message patterns detected by different algorithms, we used four data sets of unstructured *syslog* messages described in Table I. The *sudo* and *su* data sets represented small event logs with authentication messages from *sudo* and *su* tools on Linux. The *sshd* and *suricata* data sets had over 400,000 events and represented event logs with medium size. The *sshd* data set contained Linux SSH daemon messages, while the *suricata* data set contained Suricata IDS alert messages generated by a network IDS on an organizational outer network perimeter.

TABLE I
Data Sets for Evaluating the Pattern Detection Rate

Data set	# of events	# of message patterns
sudo	815	5
su	1,496	12
sshd	420,104	34
suricata	499,805	1

For each data set from Table I, message patterns were first identified by the human expert (see the last column in Table I). Note that in the case of the *suricata* data set only one pattern was identified, since all IDS alerts had the same format.

Also, the *sshd* data set was especially challenging for the event log pattern mining algorithms, because manually identified patterns had very different occurrence times – 10 patterns were very frequent and appeared at least 1,000 times in the data set, while 15 patterns were infrequent and appeared less than 20 times. Therefore, when consulting the authors of log mining algorithms, we also asked for specific recommendations on the detection of infrequent event patterns. For example, the author of IPLoM advised to set the file and partition support thresholds to 0.

As another example, for AEL and LogCluster the usage of *iterative mining* was suggested – the algorithms should be first executed with parameters that facilitate the detection of frequent patterns, assigning events not matching these patterns to the cluster of outliers. The cluster of outliers should then serve as an input for the next mining step in order to detect less frequent patterns. This technique is well-known and is not only illustrated in research papers for LogCluster and other algorithms [4], [12], but is also supported by industrial event log pattern mining tools like *pdbtool* from the widely used *syslog-ng* framework [23]. Unfortunately, the AEL implementation from LogPAI repository did not support the extraction of outlier events to a separate file, and we were thus not able to use this technique for AEL.

For assessing the algorithm accuracy, Zhu et al. defined the accuracy as the ratio of correctly parsed log messages to the total number of messages, and the message was regarded as correctly parsed if and only if its event pattern corresponded to the same group of log messages as the ground truth does [13]. However, this definition focuses on how event log messages are divided into groups, and does not consider the nature of the message patterns that are generated for message groups. For illustrating this issue, consider an example event log of three messages from Fig. 2.

```
# an example event log with three messages
Disconnected from 10.11.1.7 port 40387
Disconnected from 10.16.9.4 port 31903
Disconnected from 10.18.5.4 port 10226

# a pattern identified by a human expert
Disconnected from * port *

# a pattern identified by an algorithm
* from * * *
```

Fig. 2. Example event log and message patterns.

All three messages in the event log correspond to one message pattern and the algorithm has correctly assigned all messages to one group, thus yielding the accuracy of 1. However, the pattern that is generated by the algorithm does not represent the event log messages properly, and is quite different from the pattern created by the human expert.

Another issue of the accuracy metric is its ambiguity, because several event log clustering algorithms can assign a message to more than one cluster (i.e., clusters are allowed to overlap) [11], [12]. For example, with the LogCluster implementation this clustering mode can be activated with the *–aggrsup* command line switch. It remains unclear how to calculate the accuracy in such cases.

For addressing these issues, we used the *pattern detection rate* metric for assessing the quality of detected patterns. If P is the set of patterns identified by the human expert, then pattern detection rate for algorithm A is defined as $|F|/|P|$, where F is the set of all patterns from P identified by A. For example, if the algorithm A discovers a pattern *Disconnected from* $\langle * \rangle$ *port* $\langle * \rangle$ for the messages from Fig. 2, where $\langle * \rangle$ denotes a wildcard that matches a variable word, the pattern *Disconnected from * port ** specified by the human expert has been correctly identified by A.

For measuring the pattern detection rate, we tested all algorithms according to author recommendations with many different settings. Since we were not able to contact the authors of Spell and LogSig, we had to evaluate these tools without any specific guidance, trying a wide range of parameter values. Table II presents the best pattern detection rates observed for each algorithm (the numbers of detected patterns are provided in parentheses).

TABLE II
PATTERN DETECTION RATE OF EVALUATED ALGORITHMS

	sudo	su	sshd	suricata
IPLoM	0.8 (4)	0.917 (11)	0.5 (17)	0 (0)
Spell	0.8 (4)	0.917 (11)	0.382 (13)	0 (0)
Drain	0.6 (3)	0.917 (11)	0.706 (24)	0 (0)
AEL	0.8 (4)	0.917 (11)	0.559 (19)	0 (0)
LogSig	0.6 (3)	0.667 (8)	0.353 (12)	0 (0)
LenMa	0.6 (3)	0.833 (10)	0.706 (24)	0 (0)
LogCluster	1 (5)	0.833 (10)	0.794 (27)	1 (1)

According to Table II, all algorithms apart from LogCluster failed to discover the IDS alert pattern from the *suricata* data set. For understanding the reason for this phenomenon, see the example messages from the *suricata* data set in Fig. 3.

```
[1:2101411:13] GPL SNMP public access udp
[Classification: Attempted Information Leak]
[Priority: 2] {UDP} 10.19.17.8:59071 -> 192.168.24.27:161

[1:2025883:3] ET EXPLOIT MVPower DVR Shell UCE
[Classification: Attempted Administrator Privilege Gain]
[Priority: 1] {TCP} 10.35.18.16:39952 -> 192.168.12.15:80
```

Fig. 3. Example messages from the *suricata* data set.

As can be seen from Fig. 3, the IDS alert texts and classification descriptions had different lengths in terms of words. Consequently, IDS alert messages had different lengths (e.g., two messages from Fig. 3 contain 16 and 18 words). As discussed in section II, IPLoM, LenMa, AEL and Drain assume that messages corresponding to the same pattern must have the same number of words. Due to this algorithmic limitation, IDS alert messages were never grouped together, and deriving a common message pattern for all IDS alerts was thus not possible. This limitation was also acknowledged by the authors of several algorithms (we were not able to consult with the authors of Spell and LogSig to establish why the pattern was not found). However, scenarios like illustrated by the *suricata* data set are quite common for security and system logs (e.g., see the network switch messages from Fig. 1).

Our second finding was that using algorithms with well chosen input settings can greatly improve their performance. For example, Zhu et al. reported a modest average accuracy of 0.665 for LogCluster [13], but as pointed out in [19], the input settings of LogCluster were suboptimal. In contrast, as illustrated by Table II, LogCluster can feature a high pattern detection rate when used in appropriate way. In particular, some past studies (e.g., [13], [15]) have mistakenly claimed that LogCluster is not able to find infrequent patterns from event logs. However, as already discussed in this section, iterative mining allows for the detection of both frequent and infrequent patterns. This is highlighted by results from Table II – in the case of the *sshd* data set almost half of the patterns identified by the human expert were infrequent, and LogCluster had the best pattern detection rate for this data set.

In [13], Zhu et al. reported Drain to have the best average accuracy that was about 9 percentage points higher than the second best result from IPLoM, and 20 percentage points higher than the result from LogCluster. However, we did not observe the superiority of Drain over other algorithms – it had the best pattern detection rate of 0.917 only on the *su* data set, but in this particular case three other algorithms (IPLoM, Spell and AEL) produced the same result. For other data sets, Drain's pattern detection rate remained below 0.71 and it was outperformed by other algorithms.

When leaving aside the *suricata* data set that four algorithms were not able to handle due to their design limitations, we did not identify any particular algorithm that would perform noticeably better than competitors on all other data sets (*sudo*, *su* and *sshd*). For example, for the *sudo* data set IPLoM, Spell, AEL and LogCluster were four best algorithms, detecting at least 4 patterns out of 5. For the *su* data set, all algorithms except LogSig identified 10–11 patterns out of 12. For the *sshd* data set, three best algorithms were LenMa, LogCluster and Drain with 24–27 detected patterns out of 34. In other words, every algorithm apart from LogSig achieved a good pattern detection rate on at least two data sets. However, the modest performance of LogSig might be explained by the fact that we did not manage to consult with its authors.

When not considering LogSig, our results illustrate that with the use of well chosen input settings and proper data mining techniques, all algorithms can achieve a good pattern detection rate. Furthermore, there is no "best" algorithm for message pattern detection, and the choice of the algorithm should depend on its suitability for the given scenario (e.g., algorithms with a builtin outlier detection mechanism might be more suitable for anomaly detection tasks).

C. Computational Cost

For assessing the computational cost of algorithms, data sets described in Table III were used. The *webserver* data set represented a medium-sized event log and contained HTTP request events in Apache web server format (events were collected during the XS20 cyber security exercise). The remaining three data sets represented large event logs, with the *windows* data set containing about 3.5 million events received from 21 Windows nodes during the XS20 exercise. The *ttu-syslog* data set had about 17 million *syslog* events from 149 Linux servers and network devices in Tallinn University of Technology, while the *xs19-syslog* data set had over 27 million *syslog* events received from 54 Linux servers during the XS19 cyber security exercise.

TABLE III
DATA SETS FOR EVALUATING THE COMPUTATIONAL COST

Data set	# of events	Size
webserver	95,457	21MB
windows	3,551,025	3.2GB
ttu-syslog	17,015,421	2.5GB
xs19-syslog	27,365,365	4.6GB

When evaluating the algorithms on data sets from Table III, the CPU time and memory consumption of algorithms were measured, testing each algorithm with a number of input setting combinations. If the execution of a test took longer than 36 hours (1.5 days), the test was interrupted and regarded as incomplete.

During the experiments we discovered that IPLoM, AEL, Spell and LogSig were not able to process the *windows*, *ttu-syslog* and *xs19-syslog* data sets within 36 hours (LogSig run out of memory and crashed during all tests). Although LenMa managed to process the *windows* data set, consuming 9,806–12,005 seconds (about 2.7–3.3 hours) of CPU time and about 8GB of memory, it was not able to complete the processing of *ttu-syslog* and *xs19-syslog* data sets within 36 hours. Since only Drain and LogCluster managed to complete all tests, we are providing more detailed resource consumption data only for these two algorithms in the remainder of this section.

When evaluating LogCluster, we employed three relative support threshold values 1%, 0.5% and 0.1% that are commonly used [11], [19]. Relative support threshold is calculated from the number of events in the event log, for example, if the event log contains 20,000 events, relative support threshold 0.1% means support threshold 20. Drain was evaluated with three combinations of values for the *st* (similarity threshold for searching log groups) and *depth* (parse tree depth) parameters: *st*=0.4 and *depth*=4 (default settings for Drain), *st*=0.3 and *depth*=6, *st*=0.6 and *depth*=5. These three combinations were found to produce the best results during the experiments described in section IIIB.

Unlike the LogCluster implementation, Drain is not a UNIX tool but rather a library that requires the development of a custom Python script for event log mining. In the Drain repository, an example script called *drain_bigfile_demo.py* was provided for the purposes of mining large event logs [24]. This script loads the entire event log file into memory before the pattern mining which is apparently done for speeding up the mining. However, this can noticeably increase the memory footprint of the algorithm. Therefore, we also evaluated Drain in memory-efficient mode that involves keeping the event log on disk only. For assessing the memory saving techniques of LogCluster, we also evaluated LogCluster with sketching enabled (creating a sketch involves an additional pass over the event log and thus requires extra CPU time [11]).

The experiment results have been given in Table IV, with each cell providing the consumed CPU time and maximum memory footprint of the algorithm. The CPU time and memory consumption readings in parentheses represent the case where the given algorithm was executed in a memory-efficient mode. Since both algorithms are single-threaded, consumed CPU time closely reflects the run time of the algorithms.

TABLE IV
RESOURCE CONSUMPTION OF EVALUATED ALGORITHMS

	webserver	**windows**	**ttu-syslog**	**xs19-syslog**
Drain: st=0.3, depth=6	3s 48MB (3s 20MB)	234s 3,602MB (235s 36MB)	3,669s 3,665MB (3,851s 22MB)	838s 6,557MB (840s 31MB)
Drain: st=0.4, depth=4	3s 48MB (3s 21MB)	430s 3,603MB (416s 36MB)	20,224s 3,668MB (20,323s 24MB)	9,283s 6,552MB (9,513s 25MB)
Drain: st=0.6, depth=5	6s 48MB (6s 21MB)	437s 3,605MB (435s 37MB)	83,606s 3,694MB (82,882s 49MB)	4,953s 6,581MB (4,907s 55MB)
LogCluster: rsupport=1%	4s 33MB (7s 19MB)	739s 844MB (1,286s 19MB)	667s 1,911MB (1,330s 32MB)	953s 5,571MB (1,832s 28MB)
LogCluster: rsupport=0.5%	4s 33MB (8s 21MB)	761s 925MB (1,312s 19MB)	682s 1,911MB (1,352s 63MB)	969s 5,571MB (1,864s 33MB)
LogCluster: rsupport=0.1%	4s 38MB (8s 33MB)	758s 2,735MB (1,355s 23MB)	699s 1,912MB (1,369s 161MB)	993s 5,571MB (1,873s 74MB)

As Table IV indicates, when algorithms were executed in memory-efficient mode, the memory footprint of both algorithms decreased dramatically for large event logs. Also, because event logs were stored on a fast local SSD disk, memory-efficient mode did not impact the CPU time consumption of Drain significantly.

Our second observation was that Drain and LogCluster had a modest CPU time consumption and comparable memory footprint for the *webserver* and *windows* data sets which contain about 95 thousand and 3.5 million events respectively. For example, for the *windows* data set the CPU time consumption of the algorithms was 234–761 seconds (about 4–13 minutes; LogCluster needed somewhat more CPU time because unlike Drain, it made two passes over the event log).

However, as the number of events in data sets increased, Drain started to need noticeably more CPU time than LogCluster. The *ttu-syslog* and *xs19-syslog* data sets contained about 17 and 27 million events respectively, and for these data sets we observed a comparable CPU time consumption only during one experiment for the *xs19-syslog* data set (see the last cell of the first data row in Table IV). In other cases for this data set, Drain needed 4,953 seconds (about 1.4 hours) and 9,283 seconds (about 2.6 hours) of CPU time, while for LogCluster the CPU time consumption was 953–993 seconds (about 16–17 minutes). For the *ttu-syslog* data set, LogCluster required 667–699 seconds (about 11–12 minutes) of CPU time, while Drain needed 3,669 seconds (about 1 hour), 20,224 seconds (about 5.6 hours), and 83,606 seconds (about 23.2 hours).

Also, Drain was much more sensitive to different input settings – for example, for the *ttu-syslog* data set the CPU time consumption ranged from 1 hour to almost 1 day. On the other hand, the largest CPU time difference observed for LogCluster was less than 1 minute for all data sets. The CPU time consumption of LogCluster remained modest for much lower support thresholds than 0.1% – when we tested a very low relative support threshold 0.001%, LogCluster required 865–1,166 seconds (about 14–19 minutes) of CPU time for processing the *windows*, *ttu-syslog* and *xs19-syslog* data sets.

According to our observations, performance of Drain was negatively impacted not only by the number of events in the event log, but also by heterogeneity of log data. While *webserver* and *windows* data sets featured a moderate number of event types, *ttu-syslog* and *xs19-syslog* data sets contained many events of different kind with a wide variety of message texts. For these data sets, Drain discovered thousands of patterns during the event log mining process, with a large fraction of these patterns being redundant (e.g., many too specific patterns that should have been detected as one pattern). Although Drain arranges detected patterns into a parse tree for reducing its computational cost, for larger heterogenous data sets the parse tree is apparently not always efficient, leading to a high CPU time consumption.

As discussed previously in this section, AEL, IPLoM, LenMa, Spell and LogSig did not scale to large event logs. Contrary to our results, Zhu et al. [13] reported IPLoM and AEL to have a good performance on three 1GB event logs that were generated from the following data sets available in LogHub repository [18] – HDFS (1.47GB), BGL (709MB) and Android (3.38GB, with 184MB of data being publicly available). According to Zhu et al., 1GB event logs were generated by truncating the data sets to 1GB [13]. However, since BGL and Android data sets were smaller than 1GB, we were able to repeat this process only for the HDFS data set.

Because we hypothesized that the performance results from [13] might be a consequence of event log preprocessing, we repeated the experiments for the unmodified event logs. Since for the BGL and Android data set we were not able to generate the 1GB event logs, we used the original 709MB (BGL) and 184MB (Android) data sets instead. When we executed the algorithms with the same settings as documented in LogPAI repository, we found that IPLoM was not able to process the Android data set and crashed (we experienced the same result after using a number of different settings for IPLoM).

As for AEL, in [13] the run times of 20–30 minutes were reported for HDFS and Android data sets, while for BGL the run time was about 200 minutes. In contrast, for the Android data set we observed the run time of 4 hours and 10 minutes, although we used a much smaller data set (184MB vs 1GB). For the BGL data set (about 30% smaller than the data set used by Zhu et al.), we observed the run time of 2 days, 10 hours and 48 minutes. Finally, for the HDFS data set the run time was 5 days, 12 hours and 2 minutes.

Note that the implementations of evaluated algorithms were single-threaded and we tested them on a physical server with the same OS type (Linux), CPU type (Intel Xeon), and the same amount of memory (64GB) as reported in [13]. Therefore, aforementioned significant run time gaps can not be explained solely by hardware differences. They also reflect the fact that preprocessing of event log data can have a profound effect on the algorithm performance, providing an unrealistic view about the actual performance on the original event log.

IV. Conclusion and Future Work

In this paper, we have conducted a comparative study of event log pattern mining algorithms, evaluating the pattern detection rate and resource consumption of seven algorithms. Our main findings are summarized below.

First, we found that several commonly used algorithms (Drain, IPLoM, AEL and LenMa) share the same design limitation which prevents the detection of event patterns for messages with different number of words. As discussed in this work, such event patterns are commonly found in security and system logs.

Second, for proper evaluation of event log mining algorithms, a good understanding of their input settings and usage techniques is required. The algorithms evaluated in this work featured good pattern detection rates when used with settings recommended by their authors. Also, we did not find any particular algorithm being superior to others in terms of pattern detection rate.

Third, the evaluation of event log mining algorithms should not be conducted on preprocessed event log data sets. As shown in this work, such preprocessing can significantly influence the performance of the algorithms, providing an overly optimistic picture about the algorithm performance on real-life event logs.

Finally, we found that only Drain and LogCluster scaled to larger event logs. However, Drain was not well suited for processing large event logs with a high degree of heterogeneity. Also, Drain was highly sensitive to input settings, with its CPU time consumption fluctuating from 1 hour to almost 1 day in the case of one data set. In contrast, LogCluster was much less sensitive to changes in input settings, featuring modest CPU time consumption for all scenarios.

One open issue in the field of event log pattern mining is the lack of publicly available recent data sets from commonly used applications and operating systems. For example, in Loghub repository [18] several data sets are more than 15 years old (e.g., the BGL data set discussed in section IIIC). In particular, many organizations are not willing to publish their security event logs due to their sensitive nature. As for future work, we plan to contribute to a recent initiative by NATO CCDCOE that involves publishing the data sets collected during cyber security exercises [25].

Acknowledgment

The authors express their gratitude to Prof. Rain Ottis for supporting this work.

References

[1] R. Vaarandi and S. Mäses, "How to Build a SOC on a Budget," 2022 IEEE International Conference on Cyber Security and Resilience, pp. 171–177.

[2] https://www.ietf.org/rfc/rfc3164.txt

[3] R, Vaarandi, B. Blumbergs and M. Kont, "An Unsupervised Framework for Detecting Anomalous Messages from Syslog Log Files," 2018 IEEE/IFIP Network Operations and Management Symposium, pp. 1–6.

[4] R. Vaarandi, M. Kont and M. Pihelgas, "Event Log Analysis with the LogCluster Tool," 2016 IEEE Military Communications Conference, pp. 982–987.

[5] 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, pp. 33–40.

[6] L. Tang, T. Li and C.-S. Perng, "LogSig: Generating System Events from Raw Textual Logs," 2011 ACM International Conference on Information and Knowledge Management, pp. 785–794.

[7] A. A. O. Makanju, A. N. Zincir-Heywood and E. E. Milios, "Clustering event logs using iterative partitioning," 2009 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1255–1264.

[8] M. Du and F. Li, "Spell: Streaming Parsing of System Event Logs," 2016 IEEE International Conference on Data Mining, pp. 859–864.

[9] Z. M. Jiang, A. E. Hassan, P. Flora and G. Hamann, "Abstracting Execution Logs to Execution Events for Enterprise Applications," 2008 International Conference on Quality Software, pp. 181–186.

[10] K. Shima, "Length Matters: Clustering System Log Messages using Length of Words," https://arxiv.org/abs/1611.03213, 2016.

[11] R. Vaarandi and M. Pihelgas, "LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs," 2015 International Conference on Network and Service Management, pp. 1–7.

[12] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," 2003 IEEE Workshop on IP Operations and Management, pp. 119–126.

[13] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng and M. R. Lyu, "Tools and benchmarks for automated log parsing," 2019 International Conference on Software Engineering: Software Engineering in Practice, pp. 121–130.

[14] R. Copstein, J. Schwartzentruber, N. Zincir-Heywood and M. Heywood, "Log Abstraction for Information Security: Heuristics and Reproducibility," 2021 International Conference on Availability, Reliability and Security, pp. 1–10.

[15] D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj and A. Bouziane, "A systematic literature review on automated log abstraction techniques," Information and Software Technology, Vol. 122, 2020.

[16] M. Landauer, F. Skopik, M. Wurzenberger and A. Rauber, "System log clustering approaches for cyber security applications: A survey," Computers and Security, Vol. 92, 2020.

[17] LogPAI (Log Analytics Powered by AI), https://github.com/logpai

[18] Loghub, https://doi.org/10.5281/zenodo.3227177

[19] M. Pihelgas, "Automating Defences against Cyber Operations in Computer Networks," PhD thesis, Tallinn University of Technology, 2021.

[20] LogCluster, https://ristov.github.io/logcluster/

[21] Drain, https://github.com/logpai/Drain3

[22] LenMa, https://github.com/logpai/logparser/tree/master/logparser/LenMa

[23] Syslog-ng home page, https://www.syslog-ng.com/

[24] Drain bigfile demo script, https://github.com/logpai/Drain3/blob/master/examples/drain_bigfile_demo.py

[25] E. Halisdemir, H. Karacan, M. Pihelgas, T. Lepik and S. Cho, "Data Quality Problem in AI-Based Network Intrusion Detection Systems Studies and a Solution Proposal," 2022 International Conference on Cyber Conflict, pp. 367–383.