Efficient Event Log Mining with LogClusterC

Chen Zhuge and Risto Vaarandi

This paper has been accepted for publication at the 2017 IEEE International Conference on Big Data Security on Cloud, and the final version of the paper is included in *Proceedings of the 2017 IEEE International Conference on Big Data Security on Cloud* (DOI: 10.1109/BigDataSecurity.2017.26)

Efficient Event Log Mining with LogClusterC

Chen Zhuge and Risto Vaarandi
TUT Centre for Digital Forensics and Cyber Security
Tallinn University of Technology
Tallinn, Estonia
zhugeh@yahoo.com, risto.vaarandi@ttu.ee

***Abstract*—Nowadays, many organizations collect large volumes of event log data on a daily basis, and the analysis of collected data is a challenging task. For this purpose, data mining methods have been suggested in past research papers, and several data clustering algorithms have been developed for mining line patterns from event logs. In this paper, we introduce an open-source tool called LogClusterC which implements the LogCluster algorithm for discovering line patterns and outliers from event logs. According to our performance measurements, LogClusterC compares favorably to other publicly available log clustering tools.**

Keywords–event log clustering; mining line patterns from event logs; LogCluster algorithm; data clustering; data mining

I. INTRODUCTION

In modern data centers, significant amounts of event log data are generated on a daily basis [1]. Since the manual review and analysis of larger events logs are infeasible, data mining methods have been often suggested for this purpose [2-10], with data clustering algorithms being one of the most commonly proposed approaches [2, 3, 5-8]. Because event logs are often textual and each event is described by a single event log line, previously suggested data clustering algorithms have been designed for mining line patterns from such logs. Detected line patterns (e.g., *sshd: Failed password for * from * port * ssh2*) provide valuable insights into commonly occurring event types and can be used for various purposes, e.g., the development of log monitoring and event correlation rules. Since data clustering algorithms also allow for the identification of outlier data points, they are useful for highlighting unusual events.

In our recent paper, we have proposed the LogCluster data clustering algorithm and its Perl-based prototype implementation for mining line patterns and outliers from event logs [2]. Unfortunately, we were only able to conduct preliminary experiments for assessing the performance of the LogCluster algorithm, while the development of an efficient C-based implementation and more detailed experiments were identified as future work [2]. The current paper closes this research gap and introduces LogClusterC which is an open-source C-based implementation of the LogCluster algorithm. The remainder of the paper is organized as follows – section II discusses related work, section III introduces LogClusterC, section IV describes our experiments for evaluating its performance, and section V outlines future work.

II. RELATED WORK

The earliest event log clustering algorithm is SLCT that was developed in 2003 [3]. SLCT takes support threshold *s* for its input parameter and mines clusters that contain *s* or more lines. All lines in the same cluster match a common line pattern, and a cluster is identified by a set of words with their offsets. For example, if the cluster is identified by words *(Connection, 1)*, *(to, 2)*, and *(broken, 4)*, lines in this cluster correspond to the line pattern *Connection to * broken*. SLCT reports all detected clusters as line patterns to the end user. In recent research papers, several shortcomings of SLCT have been pointed out – it does not detect wildcard tails for line patterns, it is sensitive to shifts in word positions and delimiter noise, and lower support thresholds can lead to overfitting.

In order to address these issues, Reidemeister has suggested the clustering of line patterns from SLCT with a single-linkage clustering algorithm that uses a Levenshtein distance function, and deriving a common line pattern for each cluster [5]. IPLoM by Makanju is a hierarchical clustering algorithm that splits the event log iteratively [6]. The first step involves creating partitions for event log lines with the same number of words. During the second step, a word position with the smallest number of unique words is identified in each partition, and each partition is split by the word appearing in this position. The third step involves further partitioning based on associations between word pairs. After these steps, IPLoM derives a common line pattern for each partition.

Another event log clustering algorithm is HLAer by Ning et al. [7] which uses the OPTICS clustering method [11]. According to the authors, HLAer outperforms SLCT and IPLoM on several event logs. In their recent work, Kimura et al. [8] have suggested a line pattern mining algorithm for event logs which is based on word scoring and employs the DBSCAN clustering technique [12]. A leading event log management platform Splunk [13] implements a clustering algorithm for events in the Splunk database. The algorithm uses vector-based distance function for calculating similarity between two events, but unlike other methods discussed in this section, a common line pattern is not derived for events in the same cluster.

Shortcomings of SLCT have also motivated the development of the LogCluster algorithm which is designed to find more meaningful patterns than SLCT [2]. LogCluster

takes the support threshold *s* for its input parameter, and detects *frequent words* (words that appear in *s* or more lines) and cluster candidates during separate data passes, selecting candidates with at least *s* lines as clusters. Unlike SLCT, LogCluster does not consider words with positional information.

For assigning an event log line to a cluster candidate, all frequent words from the line are arranged into a sequence in the order of appearance, and the line is assigned to the candidate which is identified by this sequence of frequent words. Also, each cluster candidate contains information about the position of infrequent words for all lines that have been assigned to this candidate. For example, if the words *User*, *logout*, and *for* are frequent, event log lines *User logout for bob* and *User logout for john doe* are assigned to the cluster candidate *User logout for *{1,2}*. This candidate is identified by the sequence *(User, logout, for)*, and the wildcard **{1,2}* matches one or two words. In addition, the LogCluster algorithm supports overfitting mitigation heuristics, uses word classes for detecting infrequent words with the same format, supports several advanced input preprocessing features, and employs sketches for reducing its memory footprint.

During our recent research, we created a publicly available Perl-based implementation of LogCluster [2], and for comparing its performance to SLCT in a fair way, we implemented SLCT in Perl. Our experiments indicated that despite several similarities between two algorithms, SLCT was 1.28-1.62 times faster than LogCluster [2]. However, we were unable to establish whether this performance gap is related to the use of Perl or is a genuine weakness of the LogCluster algorithm. The following sections describe our work that answers this research question.

III. OVERVIEW OF LOGCLUSTERC

LogClusterC is an open-source C-based implementation of the LogCluster algorithm that is publicly available [14] under the terms of GNU GPLv2. Similarly to the Perl-based implementation of the LogCluster algorithm (it is called LogClusterP in the rest of the paper), LogClusterC is a UNIX tool which is executed from command line and configured with command line options. Apart from some experimental features and Perl-specific command line options of LogClusterP, LogClusterC is compatible with LogClusterP.

Since the LogCluster algorithm has several design similarities to SLCT (such as frequent word based candidate generation), LogClusterC borrows some source code and data structures from SLCT like sketches, fast Shift-Add-Xor string hashing functions [15], and move-to-front hash tables [16]. Nevertheless, since LogCluster has a more complex cluster candidate generation procedure and has several features for addressing the shortcomings of SLCT (such as heuristics for mitigating overfitting), the code base of LogClusterC is significantly larger.

```
logclusterc --support=30 --input=suricata.log \
--lfilter='suricata\[[0-9]+\]: (.+)'  --template='$1' \
--wweight=0.5 --outliers=outliers.log



# detected clusters (reported as line patterns)

ET CINS Active Threat Intelligence Poor Reputation IP
group *{1,1} [Classification: Misc Attack] [Priority: 2]
{TCP} *{1,1} -> *{1,1}
Support : 999

GPL SNMP public access udp [Classification:
Attempted Information Leak] [Priority: 2] {UDP}
(10.24.253.130:39734|10.131.49.54:8013|10.48.31.19:8013)
-> *{1,1}
Support : 144

ET DOS Possible NTP DDoS Inbound Frequent Un-Authed
MON_LIST Requests IMPL 0x03 [Classification: Attempted
Denial of Service] [Priority: 2] {UDP} *{1,1} -> *{1,1}
Support : 132

GPL DNS named version attempt [Classification: Attempted
Information Leak] [Priority: 2] {UDP} *{1,1} -> *{1,1}
Support : 75

ET COMPROMISED Known Compromised or Hostile Host Traffic
group *{1,1} [Classification: Misc Attack] [Priority: 2]
{TCP} *{1,1} -> *{1,1}
Support : 50

ETPRO EXPLOIT Possible Asus WRT LAN Backdoor Command
Execution [Classification: Attempted Administrator
Privilege Gain] [Priority: 1] {UDP} *{1,1} -> *{1,1}
Support : 34

ETPRO DOS Possible RPC Portmapper Scanning
[Classification: Attempted Denial of Service]
[Priority: 2] {UDP} *{1,1} -> *{1,1}
Support : 34



# sample outlier events from outliers.log

Jun 21 03:53:39 mysensor suricata[30655]: [1:2019137:2]
ET WEB_SPECIFIC_APPS Possible WP CuckooTap Arbitrary File
Download [Classification: Web Application Attack]
[Priority: 1] {TCP} 10.65.9.18:59353 -> 192.168.14.10:80

Jun 21 03:54:00 mysensor suricata[30655]: [1:2020221:4]
ET WEB_SPECIFIC_APPS WP Generic revslider Arbitrary File
Download [Classification: Web Application Attack]
[Priority: 1] {TCP} 10.65.9.18:34488 -> 192.168.14.10:80

Jun 21 03:54:07 mysensor suricata[30655]: [1:2016078:2]
ET WEB_SPECIFIC_APPS Amateur Photographer Image Gallery
file parameter Local File Inclusion Attempt
[Classification: Web Application Attack] [Priority: 1]
{TCP} 10.65.9.18:45538 -> 192.168.14.10:80

Jun 21 03:54:44 mysensor suricata[30655]: [1:2015494:2]
ET WEB_SPECIFIC_APPS Wordpress Plugin PICA Photo Gallery
imgname parameter Local File Inclusion Attempt
[Classification: Web Application Attack] [Priority: 1]
{TCP} 10.65.9.18:46615 -> 192.168.14.10:80

Jun 21 03:54:50 mysensor suricata[30655]: [1:2015499:2]
ET WEB_SPECIFIC_APPS Wordpress Plugin Newsletter data
parameter Local File Inclusion vulnerability
[Classification: Web Application Attack] [Priority: 1]
{TCP} 10.65.9.18:55170 -> 192.168.14.10:80

Jun 21 03:54:56 mysensor suricata[30655]: [1:2014948:4]
ET WEB_SPECIFIC_APPS WordPress Simple Download Button
Shortcode Plugin Arbitrary File Disclosure Vulnerability
[Classification: Web Application Attack] [Priority: 1]
{TCP} 10.65.9.18:35783 -> 192.168.14.10:80

Jun 21 03:55:01 mysensor suricata[30655]: [1:2014899:6]
ET WEB_SPECIFIC_APPS Wordpress Plugin Tinymce Thumbnail
Gallery href parameter Remote File Disclosure Attempt
[Classification: Web Application Attack] [Priority: 1]
{TCP} 10.65.9.18:44546 -> 192.168.14.10:80
```

Figure 1. Example use of LogClusterC for mining an IDS alarm log (for the sake of privacy, all sensitive fields like IP addresses are anonymized).

```
logclusterc --support=100 --input=switch.log \
--lfilter=' (%[^[:space:]]+: .+)' --template='$1'



# detected clusters (reported as line patterns)

%LINEPROTO-5-UPDOWN: Line protocol on Interface *{1,1}
changed state to up
Support : 1,494

%LINEPROTO-5-UPDOWN: Line protocol on Interface *{1,1}
changed state to down
Support : 1,408

%LINK-3-UPDOWN: Interface *{1,1} changed state to up
Support : 1,261

%LINK-3-UPDOWN: Interface *{1,1} changed state to down
Support : 1,174

%DOT1X-5-SUCCESS: Authentication successful for client
*{1,1} on Interface *{1,1} AuditSessionID *{1,1}
Support : 717

%ETHPORT-5-IF_TX_FLOW_CONTROL: Interface *{1,1} operational
Transmit Flow Control state changed to on
Support : 128

%ETHPORT-5-IF_DUPLEX: Interface *{1,1} operational duplex
mode changed to Full
Support : 128

%ETHPORT-5-IF_RX_FLOW_CONTROL: Interface *{1,1} operational
Receive Flow Control state changed to on
Support : 122



# SEC rule for detecting conditions where a switch interface
# has been down for longer than 10 seconds, and alerting the
# network administrator via e-mail

type=PairWithWindow
ptype=RegExp
pattern=^[[:alpha:]]{3} [\d ]\d \d\d:\d\d:\d\d ([\w.-]+) \
.*: %LINK-3-UPDOWN: Interface ([\w\/]+), changed state to down
desc=Interface $2 on switch $1 has been down for 10 seconds
action=pipe '%s' mail -s 'Interface failure' root@example.com
ptype2=RegExp
pattern2=^[[:alpha:]]{3} [\d ]\d \d\d:\d\d:\d\d $1 \
.*: %LINK-3-UPDOWN: Interface $2, changed state to up
desc2=Interface %2 on switch %1 has come up within 10 seconds
action2=logonly
window=10
```

Figure 2. Example use of LogClusterC for discovering common event types from Cisco network switch logs, and a SEC event correlation rule example which has been derived from detected patterns.

Unlike SLCT, LogClusterC stores cluster candidates into prefix tree if the algorithm is executed with the support aggregation heuristic, in order to reduce the computational cost of the heuristic (in contrast, SLCT uses less efficient move-to-front hash table). Other differences with SLCT code base include support for the unique features of the LogCluster algorithm such as word classes and word weight functions. Finally, LogClusterC also provides several command line options for advanced output formatting and debugging which are not supported by SLCT.

Fig. 1 displays an example application of LogClusterC for the Suricata IDS alarm log. In this example, support threshold was set to 30 with the *--support* option, while the *--lfilter* and *--template* options were employed for removing the *syslog* timestamp, IDS sensor name, and *syslog* tag from each IDS alarm. With the *--wweight* option, cluster joining heuristic was enabled that is based on the concept of a word weight (see [2] for more details). The second line pattern in Fig. 1 represents a joint cluster created by this heuristic, and corresponds to SNMP probing alarms for UDP peers 10.24.253.130:39734, 10.131.49.54:8013, and 10.48.31.19:8013. LogClusterC was also configured to detect outlier events with the *--outliers* option, and Fig. 1 displays some example outliers which correspond to a sophisticated attack from host 10.65.9.18. The attack was conducted against an institutional web server and triggered a number of unusual IDS alarms. As Fig. 1 illustrates, LogClusterC can help the security analyst to quickly identify common (often botnet-related) attack and network probing patterns, and also discover rare and more elaborate individual attacks.

The use of LogClusterC is not limited to security logs and the analysis of security incidents, but it can be applied to any event log type for various other purposes. Fig. 2 depicts a LogClusterC usage example for finding frequent event types from Cisco network switch logs, in order to employ detected knowledge for building event correlation rules for network fault management. In this example, the *--lfilter* and *--template* options were used for dropping the prefix from each log message which consists of timestamp and network switch name, focusing on message text during the mining process. Each detected line pattern represents a common event type and can be easily converted into a regular expression or other format that is supported by a dedicated event correlation tool. For example, Fig. 2 presents a SEC (Simple Event Correlator) [17] rule which uses regular expressions derived from the third and fourth line pattern detected by LogClusterC (the event correlation rule alerts the network administrator if an interface goes down on a switch and does not come up within 10 seconds).

The following section will discuss experiments for evaluating the performance of LogClusterC that were conducted on supercomputer logs.

IV. PERFORMANCE OF LOGCLUSTERC

In order to evaluate the performance of LogClusterC and compare it to other log clustering algorithms, we selected seven publicly available log files from the USENIX Computer Failure Data Repository [18] that are described in Table I. All experiments were conducted on a notebook running Ubuntu 16.04 Linux with Intel Core i5-3230M 2.6GHz processor, 8GB of memory, and 180GB SSD disk.

TABLE I. LOG FILES USED DURING EXPERIMENTS

Log file	Description	Size (MB)	Size (# of lines)
Cray_A	Cray XT logs, data set 6	20.86	379,457
Cray_B	Cray XT logs, data set 4	52.12	958,075
Cray_C	Cray XT logs, data set 1	172.72	3,170,514
BGL	HPC4 BlueGene/L supercomputing system logs	708.76	4,747,963
LBR	HPC4 Liberty supercomputing system logs	30,235.34	265,569,231
TDB	HPC4 Thunderbird supercomputing system logs	30,386.44	211,212,192
SPT	HPC4 Spirit supercomputing system logs	38,236.88	272,298,969

TABLE II. PERFORMANCE COMPARISON FOR LOGCLUSTERC (LCC), SLCT, AND LOGCLUSTERP (LCP)

Row #	Log file	Support threshold	LCC runtime in seconds	SLCT runtime in seconds	LCP runtime in seconds	LCC memory usage in kilobytes	SLCT memory usage in kilobytes	LCP memory usage in kilobytes	# of clusters (candidates) detected by LCC/LCP	# of clusters (candidates) detected by SLCT
1	Cray_A	1,897	1.25	1.30	10.83	27,432	26,792	226,068	0 (63,876)	0 (53,615)
2	Cray_A	379	1.28	1.33	11.31	34,012	37,064	281,332	27 (81,044)	29 (79,280)
3	Cray_A	200	1.35	1.41	12.37	63,396	65,664	543,768	13 (151,707)	16 (150,299)
4	Cray_B	4,790	2.82	2.91	26.14	8,816	5,384	29,304	32 (5,700)	32 (910)
5	Cray_B	958	3.38	3.06	32.08	162,512	13,504	1,451,208	10 (382,322)	10 (15,667)
6	Cray_B	200	3.31	3.47	33.79	193,156	180,676	1,684,240	323 (465,187)	323 (412,317)
7	Cray_C	15,852	9.25	9.48	86.09	9,900	8,028	39,164	26 (8,622)	26 (1,011)
8	Cray_C	3,170	10.79	10.85	104.12	437,320	371,532	3,953,568	44 (1,046,947)	44 (836,461)
9	Cray_C	200	11.07	11.84	112.18	612,836	636,884	5,499,112	2,044 (1,501,118)	2,045 (1,500,167)
10	BGL	23,739	47.21	48.12	200.03	518,932	527,668	1,153,120	55 (4,160)	58 (1,890)
11	BGL	4,747	46.68	50.42	207.14	517,404	527,720	1,153,796	162 (34,844)	162 (17,213)
12	BGL	200	48.13	51.77	224.26	517,488	527,720	3,026,716	813 (543,035)	814 (538,370)
13	LBR	1,327,846	1,492.73	1,577.71	11,046.43	1,176,496	1,211,888	2,788,732	36 (93,010)	36 (54,573)
14	LBR	265,569	1,498.97	1,568.70	11,156.85	1,178,340	1,212,008	2,789,600	274 (165,197)	274 (111,536)
15	LBR	200	2,767.77	2,975.33	23,221.76	2,024,736	2,020,552	6,933,324	5,738 (7,406 / 7,505)	5,738 (7,386)
16	TDB	1,056,060	1,618.85	1,669.96	9,252.64	1,927,992	1,975,816	4,555,400	6 (197,847)	9 (85,862)
17	TDB	211,212	1,643.49	1,665.39	9,774.04	1,928,104	1,977,588	5,751,584	22 (1,126,218)	28 (913,546)
18	TDB	200	2,770.25	2,942.97	19,945.70	2,341,996	2,350,680	7,135,324	1,683 (3,292 / 3,272)	1,728 (3,314)
19	SPT	1,361,494	1,792.52	1,884.23	13,740.50	1,045,092	1,055,016	2,503,904	39 (110,722)	39 (50,506)
20	SPT	272,298	1,816.48	1,906.76	13,955.47	1,043,228	1,054,824	2,507,108	122 (407,835)	123 (325,568)
21	SPT	200	3,657.80	3,849.74	31,970.67	2,090,548	2,076,108	7,620,424	164,776 (237,505 / 237,420)	164,776 (237,083)

From clustering methods discussed in section II, we were unable to test IPLoM, HLAer, and algorithms by Reidemeister and Kimura et al., since their implementations are not publicly available. Therefore, C-based implementation of SLCT (version 0.05) [19] and LogClusterC (version 0.05) were evaluated. Both SLCT and LogClusterC were compiled with *gcc*, using the *–O2* option (optimize for speed). We also included LogClusterP (version 0.08) in all tests, in order to compare its performance with LogClusterC.

The experiment results are provided in Table II. When clustering each log file from Table I, we measured the runtime, CPU time and memory consumption, and the number of detected clusters and cluster candidates. Since consumed CPU times were closely matching runtimes, we have omitted CPU times from Table II. In the table, LCC and LCP denote LogClusterC and LogClusterP respectively. Also, two rightmost columns present the number of detected clusters, with the number of cluster candidates given in parentheses.

When evaluating LogClusterC, SLCT, and LogClusterP, we clustered each log file from Table I with relative support thresholds 0.5% and 0.1% (i.e., setting the support threshold to 0.5% and 0.1% of the number of event log lines). We also clustered log files with the support threshold of 200 that corresponds to relative thresholds 0.05271-0.00007% for log files in Table I. The employment of such a low threshold allowed for imposing heavier workloads on tested algorithms, since it leads to the generation of a larger number of frequent words and cluster candidates. When executing all implementations with this input parameter setting, the clustering of LBR, TDB, and SPT log files required significant amounts of memory, and implementations run out of RAM in several cases. For this reason, we enabled sketch based memory optimization techniques for LogClusterC, SLCT, and LogClusterP when LBR, TDB, and SPT log files were mined with support threshold 200 (experiments described by rows 15, 18, and 21 in Table II). Although implementations use the sketches in the same manner for filtering out cluster candidates with insufficient support [2, 3], the number of remaining candidates may easily differ for LogClusterC and LogClusterP as data from rows 15, 18, and 21 also illustrate.

Table II reveals several interesting phenomena that we observed when clustering log files from Table I. Firstly, we expected LogClusterC to consume more memory than SLCT, since during the experiments LogClusterC detected on average 3.05 times more candidates than SLCT. Also, the LogCluster algorithm has a more complex candidate generation procedure and requires more bytes for storing a candidate than SLCT. However, apart from the experiment depicted by row 5, the memory footprint of LogClusterC remained comparable to SLCT and was even slightly smaller in a number of cases. As the main reason, our investigation revealed the smaller memory consumption of LogClusterC during frequent word detection. Unlike SLCT, the LogCluster algorithm does not need to maintain several occurrence counters for the same word that appears in different positions. For example, during our experiments LogClusterC created on average 15.36% less counters which outweighed its larger memory footprint during candidate generation. The only exception is the experiment in row 5, where LogClusterC detected 24.40 times more candidates and therefore required 12.03 times more memory than SLCT.

Table II also indicates that apart from the experiment in row 5, LogClusterC was slightly faster than SLCT, requiring on average 3.65% less runtime. These results were unexpected, since the candidate generation procedure is more expensive for the LogCluster algorithm, and it detected more candidates during the experiments. When investigating this phenomenon, we found that SLCT needs to spend additional CPU time for encoding word position information into words. Also, word position information adds four bytes to internal representation of each word, creating additional work for the word hashing function. Although word position encoding and word hashing are implemented with fast bitwise and integer operations in SLCT implementation, the extra work will nevertheless have to be spent on *every* word during *all* data passes. On larger data sets, this effort can easily consume more CPU time than the more complex candidate generation procedure of the LogCluster algorithm. When we created an experimental SLCT version with word position encoding disabled, it outperformed LogClusterC in terms of runtime.

The experiment results and their analysis indicates that C-based implementations of LogCluster and SLCT algorithms have comparable runtime and memory usage, and the LogCluster algorithm does not have significant performance weaknesses when compared to SLCT. However, as discussed in [2], LogCluster is able to discover more refined and meaningful line patterns which is a clear advantage over SLCT.

Finally, according to Table II, LogClusterC was on average 7.82 times faster and consumed on average 4.92 times less memory than LogClusterP. We also compared the performance of LogClusterC and LogClusterP for log files from Table I with overfitting mitigation heuristics enabled, and found that the performance gap between two implementations widened further. Therefore, LogClusterC offers significant performance benefits over LogClusterP, especially when larger log files need to be mined in a fast and memory-efficient way.

V. Conclusion and Future Work

In this paper, we have introduced the LogClusterC event log mining tool and described a number of experiments for evaluating its performance against other publicly available log clustering tools. The experiments have revealed that LogClusterC compares favorably to other algorithms and tools, and is able to efficiently mine large event logs.

As for the future work, we plan to investigate opportunities to modify the LogCluster algorithm for stream mining scenarios. Another promising research direction is the visualization of clusters detected by LogClusterC. We also plan to study methods for automatic selection of the support threshold input parameter. Finally, our future research includes the use of the LogCluster algorithm in distributed computing environments.

Acknowledgment

The authors are grateful to Mr. Kaido Raiend, Mr. Ain Rasva, Dr. Rain Ottis and Prof. Olaf Maennel for supporting this research. This work was also supported by Estonian IT Academy (StudyITin.ee) and SEB Estonia.

References

[1] Risto Vaarandi and Mauno Pihelgas, "Using Security Logs for Collecting and Reporting Technical Security Metrics," in *Proceedings of the 2014 IEEE Military Communications Conference*, pp. 294-299.

[2] Risto Vaarandi and Mauno Pihelgas, "LogCluster – A Data Clustering and Pattern Mining Algorithm for Event Logs," in *Proceedings of the 2015 International Conference on Network and Service Management*, pp. 1-7.

[3] Risto Vaarandi, "A Data Clustering Algorithm for Mining Patterns From Event Logs," in *Proceedings of the 2003 IEEE Workshop on IP Operations and Management*, pp. 119-126.

[4] Kenji Yamanishi and Yuko Maruyama, "Dynamic Syslog Mining for Network Failure Monitoring," in *Proceedings of the 2005 International Conference on Knowledge Discovery and Data Mining*, pp. 499-508.

[5] Thomas Reidemeister, "Fault Diagnosis in Enterprise Software Systems Using Discrete Monitoring Data," PhD Thesis, University of Waterloo, 2012.

[6] Adetokunbo Makanju, "Exploring Event Log Analysis With Minimum Apriori Information," PhD Thesis, University of Dalhousie, 2012.

[7] Xia Ning, Geoff Jiang, Haifeng Chen and Kenji Yoshihira, "HLAer: a System for Heterogeneous Log Analysis," in *Proceedings of the 2014 SDM Workshop on Heterogeneous Learning*.

[8] Tatsuaki Kimura, Keisuke Ishibashi, Tatsuya Mori, Hiroshi Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Akio Watanabe, Akihiro Shimoda and Kohei Shiomoto, "Spatio-temporal Factorization of Log Data for Understanding Network Events," in *Proceedings of the 2014 IEEE International Conference on Computer Communications*, pp. 610-618.

[9] Sheng Ma and Joseph L. Hellerstein, "Mining Partially Periodic Event Patterns with Unknown Periods," in *Proceedings of the 17th International Conference on Data Engineering*, pp. 205-214, 2001.

[10] Wei Xu, Ling Huang, Armando Fox, David Patterson and Michael Jordan, "Mining Console Logs for Large-Scale System Problem Detection," in *Proceedings of the 3rd Workshop on Tackling Computer Systems Problems with Machine Learning Techniques*, 2008.

[11] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel and Jörg Sander, "OPTICS: Ordering Points To Identify the Clustering Structure," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pp. 49-60.

[12] Martin Esler, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," in *Proceedings of the 1996 International Conference on Knowledge Discovery and Data Mining*, pp. 226-231.

[13] https://www.splunk.com

[14] https://zhugehq.github.io/logclusterc/

[15] M. V. Ramakrishna and Justin Zobel, "Performance in Practice of String Hashing Functions," in *Proceedings of the 5th International Conference on Database Systems for Advanced Applications*, pp. 215-224, 1997.

[16] Justin Zobel, Steffen Heinz and Hugh E. Williams, "In-memory Hash Tables for Accumulating Text Vocabularies," Information Processing Letters, Vol. 80(6), pp. 271-277, 2001.

[17] Risto Vaarandi, Bernhards Blumbergs and Emin Çalışkan, "Simple Event Correlator – Best Practices for Creating Scalable Configurations," in *Proceedings of the 2015 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support*, pp. 96-100.

[18] https://www.usenix.org/cfdr

[19] https://ristov.github.io/slct/