Event Log Analysis with the LogCluster Tool

Risto Vaarandi, Markus Kont and Mauno Pihelgas

Event Log Analysis with the LogCluster Tool

Risto Vaarandi
TUT Centre for Digital Forensics and Cyber Security
Tallinn University of Technology
Tallinn, Estonia
firstname.lastname@ttu.ee

Markus Kont and Mauno Pihelgas
Technology Branch
NATO CCD COE
Tallinn, Estonia
firstname.lastname@ccdcoe.org

***Abstract*—Today, event logging is a widely accepted concept with a number of event formatting standards and event collection protocols. Event logs contain valuable information not only about system faults and performance issues, but also about security incidents. Unfortunately, since modern data centers and computer networks are known to produce large volumes of log data, the manual review of collected data is beyond human capabilities. For automating this task, a number of data mining algorithms and tools have been suggested in recent research papers. In this paper, we will describe the application of the LogCluster tool for mining event patterns and anomalous events from security and system logs.**

Keywords*—security log analysis; event log clustering; pattern mining from event logs; data mining

I. Introduction

Nowadays, event logging is supported by most applications, services, network devices, and other IT system components. Well-known standards exist for event logging (such as BSD syslog [1] and IETF syslog [2]) and widely used solutions have been developed for event log collection (such as rsyslog [3], syslog-ng [4], and Elastic Stack [5]). Event logs contain valuable information about security incidents, but since large volumes of log data are generated in modern data centers and computer networks [6], the manual review of event logs is infeasible. In order to aid the human analyst, a number of data mining algorithms and tools have been proposed [7–22]. Many suggested approaches are semi-automated, allowing for interactive discovery of event patterns from event logs. This knowledge can be used for various purposes like handling security incidents and developing event correlation rules [23]. During the last decade, data clustering algorithms have been often suggested for mining line patterns from textual event logs. Proposed algorithms assume that each line in the event log is a complete representation of some event. The algorithms divide the lines into clusters, so that lines from the same cluster are similar and matching the same line pattern. Instead of printing lines in each cluster, the algorithms output a line pattern for each cluster to the end user. Also, lines that do not fit into any of the detected clusters are arranged into a special cluster of outliers and reported individually. Due to their nature, clustering algorithms are able to identify not only event patterns that reflect regularities, but also unusual outlier events that deserve closer attention from security personnel.

In this paper, we describe the LogCluster tool for mining textual event logs and present example scenarios of detecting security incidents and anomalous events. Full details of the clustering algorithm implemented by the tool have been given in our recent paper [7]. The remainder of this paper is organized as follows – section II reviews related work, section III describes the LogCluster tool and focuses on its newly developed functionality along with several use cases, while section IV concludes the paper and provides the download and licensing information for the LogCluster tool.

This work has been supported by Estonian IT Academy (StudyITin.ee) and SEB Estonia.

II. Related Work

One of the earliest event log clustering algorithms is SLCT [8] which has been applied in various domains like IDS alarm log processing [9, 10], detection of recurrent fault conditions [11, 12], and visualization of event log data [19, 20]. SLCT takes *support threshold s* as a user-given input parameter, and starts the clustering process by identifying *frequent words* that appear in *s* or more event log lines. The words are considered with positional information, e.g., if the fifth word of the event log line is *kernel*, it is treated as a tuple *(kernel, 5)*. After identifying frequent words, another pass is made over input data for assigning lines to cluster candidates. For each line, all frequent words are extracted, and the candidate for this line is identified by the set of extracted words. After the data pass, *frequent candidates* that contain *s* or more lines are selected as clusters. The number of lines in a cluster (or a candidate) is called the *support* of the cluster (or the candidate). For example, consider the event log with four lines:

User bob login from 10.0.0.1

User alice login from 10.0.0.1

User jim login from 10.0.0.2

User Srv Admin login from 10.0.0.3

If *s=3*, the words *(User, 1)*, *(login, 3)*, and *(from, 4)* are detected as frequent. Also, two candidates are identified – the candidate *{(User, 1), (login, 3), (from, 4)}* with support 3 that contains first three lines, and the candidate *{(User, 1)}* with support 1 that contains the last line. The first candidate is selected as a cluster and is reported as the line pattern *User * login from* (since the cluster has no word associated with position 2, a wildcard is printed for this position). Finally, the last line is reported as an outlier.

Unfortunately, SLCT is known to suffer from some shortcomings [9, 12, 13]. Firstly, it does not detect wildcard suffixes for line patterns as illustrated by the previous example.

Secondly, SLCT is sensitive to word delimiter noise and shifts in word positions. For instance, in the above example the last event log line is not assigned to the cluster represented by the pattern *User * login from*. Finally, when mining is conducted with lower support thresholds, SLCT is prone to overfitting – clusters with meaningful line patterns could be needlessly split, so that resulting clusters have too specific line patterns. For example, if *s=2* for the above event log example, only the pattern *User * login from 10.0.0.1* is detected which does not represent the general case.

Recently, we have developed a clustering algorithm called LogCluster that addresses the shortcomings of SLCT [7]. Similarly to SLCT, the user must supply the support threshold *s* to LogCluster which is used for finding frequent words during the first pass over the event log. However, positional information is not encoded into words. In order to identify a cluster candidate for each event log line during the second data pass, LogCluster extracts all frequent words from the line and arranges them into a tuple. Also, summary information about infrequent words in all assigned lines is maintained with each candidate. Candidates containing *s* or more lines are selected as clusters and reported as line patterns, while lines without a cluster are regarded outliers and reported during an optional data pass. For instance, if *s=3* for the example event log above, all lines are assigned to the cluster identified by the tuple *(User, login, from)*, and the line pattern *User *{1,2} login from *{1,1}* is reported for this cluster together with its support of 4.

Reidemeister et al developed a methodology for diagnosing recurrent faults in software systems which employs a modified version of SLCT for software logs [11, 12]. In order to handle delimiter noise and shifts in word positions, results from SLCT are clustered further with a single-linkage clustering algorithm that uses a Levenshtein distance function. Detected knowledge is then harnessed for building decision tree classifiers.

Makanju developed a divisive clustering algorithm IPLoM that starts with the event log as a single cluster and splits it into partitions during three steps [13]. Splitting is based on various criteria, such as the number of words in event log lines and associations between word pairs. After splitting, a line pattern is derived for each partition. Unlike SLCT, IPLoM is able to identify wildcard suffixes for line patterns.

Apart from clustering algorithms, frequent itemset mining methods have been often employed for event log mining. LogHound is a generalization of the Apriori algorithm that can discover line patterns from textual event logs [9]. Other frequent itemset mining approaches have been mainly used for the detection of temporal associations between event types [14–18] and for mining NetFlow traffic patterns [21, 22].

III. The LogCluster Tool

The LogCluster tool is an open-source Perl-based UNIX command line utility. It is able to mine meaningful patterns from large event logs, and our recent study provides detailed performance data for the 0.01 version [7]. In this section, we will review the features of the latest version and discuss several use cases. Event logs presented in this section originate from several large and mid-size private and military organizations, with all sensitive data being obfuscated in Fig. 1–4.

A. Introduction and Basic Use

All parameters are supplied to the LogCluster tool with command line options. For example, the following command line

logcluster.pl --support=100 --input=/var/log/messages

mines line patterns from */var/log/messages* with support threshold 100. Default word delimiter is whitespace, but custom delimiter can be defined with the *--separator* command line option. In order to mine patterns from several log files, multiple *--input* options can be provided and wildcards can be used in file names (e.g., *--input=/var/log/*.log*). The above command line runs the basic variant of the LogCluster algorithm which involves two passes over */var/log/messages* for finding frequent words and cluster candidates respectively.

```
logcluster.pl --input=suricata.log --support=1000 \
              --wsize=10000 --csize=10000

Feb 27 *{1,1} myhost suricata[17447]: [1:2012708:2]
ET WEB_SERVER HTTP 414 Request URI Too Large
[Classification: Web Application Attack]
[Priority: 1] {TCP} 10.0.19.12:80 -> *{1,1}
Support: 44744

Feb 5 *{1,1} myhost suricata[2223]: [1:2006445:13]
ET WEB_SERVER Possible SQL Injection Attempt SELECT FROM
[Classification: Web Application Attack]
[Priority: 1] {TCP} *{1,1} -> 10.0.33.7:80
Support: 39692

Oct 18 *{1,1} myhost suricata[18941]: [1:2006446:11]
ET WEB_SERVER Possible SQL Injection Attempt UNION SELECT
[Classification: Web Application Attack]
[Priority: 1] {TCP} *{1,1} -> 10.0.3.5:80
Support: 7493

Mar 15 *{1,1} myhost suricata[25554]: [1:2016936:2]
ET WEB_SERVER SQL Injection Local File Access Attempt
Using LOAD_FILE [Classification: Web Application Attack]
[Priority: 1] {TCP} *{1,1} -> 10.0.6.1:80
Support: 3293

Jan 2 *{1,1} myhost suricata[30119]: [1:2101201:10]
GPL WEB_SERVER 403 Forbidden [Classification: Attempted
Information Leak] [Priority: 2] {TCP} 10.0.3.9:80 -> *{1,1}
Support: 2826
…
```

Fig. 1. Sample Suricata IDS alarm patterns.

When mining larger log files, the number of distinct words can be quite large, and with lower support thresholds many cluster candidates could be generated. Therefore, it is expensive to keep all words and cluster candidates in memory when their occurrence counts are established. In order to reduce the memory consumption by filtering out infrequent words, a sketch based technique can be employed which requires an extra data pass. During the data pass, the word sketch of *m* counters is created, where each counter reflects the occurrence counts of many words and acts as a filter (see [7] for full details). A similar method can be used for filtering out infrequent candidates. The number of counters in the word sketch is set with the *--wsize* command line option, while the size of the candidate sketch can be set with the *--csize* option.

Fig. 1 illustrates example line patterns detected by the LogCluster tool from the Suricata IDS log file. The log file covered the period of 6 months and contained 949,920 lines.

Since the support threshold was set to 1000, strong alarm patterns were identified which reflect the days when intensive attacks against specific hosts were conducted. During the mining process, the word and candidate sketches of 10,000 counters were employed. Both sketches involved an additional data pass, and the memory consumption of the LogCluster tool was reduced from 406.2 MB to 13.4 MB.

B. *Preprocessing Input*

While Fig. 1 provides an example of discovering relevant patterns from a raw log file, quite often the detection of meaningful patterns requires elaborate preprocessing of event logs (e.g., dropping irrelevant events, removal of timestamps, and rewriting specific parts of event log lines). In many cases, such tasks require dedicated scripts that store preprocessed log data to disk. For avoiding this overhead, the LogCluster tool provides native support for flexible input preprocessing. If a regular expression is supplied with the *--lfilter* option, only the lines matching this regular expression are clustered. Also, if the regular expression sets match variables, the variables can be used in the format string defined with the *--template* option, in order to convert matching event log lines in memory before further processing. Fig. 2 depicts a LogCluster application example for SSH daemon syslog events where timestamp, hostname, and program name were discarded during clustering (for instance, the event log line *Mar 27 12:01:33 server113 sshd[15437]: test message* was converted to *test message*). The *authpriv.log* file contained over 7 million lines from hundreds of UNIX servers, while 98,920 lines matched the *--lfilter* option and were converted. The mining was conducted with the relative support threshold of 1% (that means setting support threshold to 1% of the number of clustered lines, i.e., 989). Also, 2200 outlier event log lines were detected and written to the *outliers.log* file. Most outliers reflected SSH probing of non-existing user accounts by the organizational vulnerability scanning engine, but some outliers were also error messages that manifested system faults and configuration errors.

In some cases, regular expression based filtering and conversion might not be sufficient for complex preprocessing tasks. For addressing this issue, the LogCluster tool also supports the *--lcfunc* option which takes a definition of an anonymous Perl function for its value. The function is compiled when LogCluster starts, and the compiled code is invoked for filtering and converting each event log line. An event log line is passed to the function as its only input parameter, and in order to indicate the line should not be considered during clustering, Perl *undef* value must be returned from the function. If any other value is returned, it replaces the original event log line. For example, if the option

--lcfunc='sub { if ($_[0] =~ s/192\.168\.\d{1,3}\.\d{1,3}/ip-192.168/g) { return $_[0]; } return undef; }'

is employed, LogCluster only considers lines which contain IP addresses from the *192.168.0.0/16* network, and each such address is replaced with the string *ip-192.168*. Finally, since providing longer Perl functions in command line is not convenient, input preprocessing routines can be defined in a separate Perl module and used through the *--lcfunc* interface. For example, if the option

--lcfunc='require "/opt/logcluster/perlmod/Test.pm"; sub { Test::lineconvert($_[0]); }'

is provided, the function *lineconvert()* from the module */opt/logcluster/perlmod/Test.pm* is invoked for filtering and converting each event log line.

```
logcluster.pl --input=authpriv.log --rsupport=1 --aggrsup \
--lfilter='sshd\[\d+\]: (?<msgtext>.+)' --template='$+{msgtext}' \
--outliers=outliers.log

pam_unix(sshd:session): session opened for user *{1,1} by (uid=0)
Support: 26708

Accepted publickey for *{1,1} from *{1,1} port *{1,1} ssh2
Support: 24160

pam_unix(sshd:auth): authentication failure; logname= uid=0 euid=0
tty=ssh ruser= *{1,2}
Support: 1362
…

# examples of outlier events from outliers.log

Mar 18 04:43:43 server112 sshd[22902]: Failed password for
  invalid user emailswitch from 10.31.97.21 port 46668 ssh2
Mar 18 04:43:50 server112 sshd[22936]: Failed password for
  invalid user admin from 10.31.97.21 port 46686 ssh2
Mar 18 04:44:06 server112 sshd[23000]: Failed password for
  invalid user manage from 10.31.97.21 port 46726 ssh2
Mar 18 04:44:53 server112 sshd[23056]: Failed password for
  invalid user cisco from 10.31.97.21 port 46841 ssh2
Mar 18 06:31:38 server29 sshd[12133]: PAM unable to dlopen
  (/lib64/security/pam_oddjob_mkhomedir.so):
  /lib64/security/pam_oddjob_mkhomedir.so: cannot open shared
  object file: No such file or directory
```

Fig. 2. Sample SSH daemon event patterns and outlier events.

C. *Defining Word Classes*

In many cases, infrequent words share the same format that is not detected during clustering. For example, the program name of syslog messages is often followed by frequently changing process ID, creating many infrequent words for the same program (e.g., *sshd[18991]:* and *sshd[7655]:*). For discovering such regularities, LogCluster supports the creation of word classes according to user-defined rules, where each word class represents many infrequent words and captures their commonalities. If a regular expression is given with the *--wfilter* option, word classes are set up for all words that match this expression. Word class creation involves searching the word for all substrings that match the regular expression supplied with the *--wsearch* option, and replacing these substrings with the string provided with the *--wreplace* option. For example, with the following command line options

--wfilter='^\w+\[\d+\]:$' --wsearch='\[\d+\]' --wreplace='[PID]'

the word class *sshd[PID]:* is created for words *sshd[1321]:* and *sshd[9583]:*, while the word class *suricata[PID]:* is set up for words *suricata[2133]:* and *suricata[17743]:*. Word classes are treated like regular words by LogCluster. If a word class is frequent, it replaces all corresponding infrequent words during the clustering process.

The LogCluster tool also features a more powerful *--wcfunc* option which allows for the creation of word classes with a user-defined Perl function. Unlike with regular expression based options, multiple word classes can be created for a word, and the classes are returned from the function as a Perl list. The order of word classes in the list defines their priority – if the word is infrequent, the first frequent word class from the list replaces the word. For example, if the

--wcfunc='sub { if ($_[0] =~ /^Chrome\/(\d+)/) { return ("Chrome/$1", "Chrome"); } }'

option is provided, word classes *Chrome/49* and *Chrome* are created for the word *Chrome/49.0.2623.87*, with *Chrome/49* having precedence over *Chrome*. If the word class *Chrome/49* and the word *Chrome/49.0.2623.87* are infrequent, then word class *Chrome* replaces *Chrome/49.0.2623.87* during the clustering process. As with the *--lcfunc* option described in the previous subsection, more complex word class creation routines can be separated into Perl modules (the LogCluster distribution includes one such example module).

D. Joining Clusters

As discussed in section II, clusters can be split needlessly with lower support thresholds. As a result, instead of generic meaningful line patterns too specific patterns are detected. For addressing the overfitting problem, LogCluster supports two heuristics for joining clusters that make resulting line patterns more comprehensible to the human analyst. The first heuristic is enabled with the *--aggrsup* option and is applied to cluster candidates before clusters are selected. The heuristic allows cluster overlaps, in order to increase the support of clusters with more generic patterns – for each cluster candidate, other candidates with more specific line patterns are identified, and their lines are also assigned to the given candidate. For instance, if there are two candidates with line patterns *Interface *{1,1} down* and *Interface eth0 down*, lines of the second candidate are also assigned to the first candidate, and the support of the first candidate is incremented by the support of the second candidate. The use of the *--aggrsup* option has been illustrated in Fig. 2.

The second heuristic is applied after clusters have been selected from candidates. The heuristic relies on word weight functions that are described below. Suppose that a and b are frequent words, m denotes the number of event log lines where a appears, and n denotes the number of event log lines where both a and b appear. Provided that $n > 0$ (i.e., there is at least one event log line where both a and b are present), dependency from a to b is defined as:

$$dep(a, b) = n / m$$

Note that $dep(a, a) = 1$ and $0 < dep(a, b) \leq 1$, with higher value of $dep(a, b)$ indicating higher likelihood of observing b when a is present. For measuring how strongly each word in the line pattern is correlated to other words in this pattern, LogCluster defines several word weight functions which return values from 0 to 1. If $w_1,\ldots,w_k$ are words in the line pattern, the word weight function f_1 is defined as:

$$f_1(w_i) = \textstyle\sum_{j=1}^{k} dep(w_j, w_i) / k$$

The smaller the value of $f_1(w_i)$, the less likely it is to observe w_i with other words of the pattern. For identifying words with insufficient weights in line patterns, the *word weight threshold* is defined with the *--wweight* option. For example, if *--wweight=0.5* while $dep(Interface, eth0)$ = 0.07 and $dep(down, eth0)$ = 0.07, then the word *eth0* has insufficient weight in the pattern *Interface eth0 down*, since $f_1(eth0)$ = (0.07 + 0.07 + 1) / 3 = 0.38. The cluster joining heuristic replaces such words with special tokens in cluster identifier tuples, and joins two clusters if their modified tuples are identical. Then a new line pattern is derived for the joint cluster by creating lists from words with insufficient weights and joining wildcards (see [7] for full details). For instance, suppose there are two clusters with identifier tuples *(Interface, eth0, down)* and *(Interface, eth1, down)*, and line patterns *Interface eth0 down* and *Interface eth1 down*. If words *eth0* and *eth1* have insufficient weights in their respective patterns, modified identifier tuples for both clusters are *(Interface, TOKEN, down)*, and two clusters are thus joined into a new cluster with the line pattern *Interface (eth0|eth1) down*. Since overfitting introduces words with lower weights into patterns, the cluster joining heuristic helps to reduce the number of such patterns by joining them into more meaningful patterns which are built around strongly associated words.

The f_1 word weight function has some drawbacks that have motivated the development of additional functions in recent versions of LogCluster. Firstly, if k is the number of words in the pattern, then $1/k \leq f_1(w_i) \leq 1$, since $dep(w_i, w_i)$ = 1 (i.e., f_1 allows the word to contribute $1/k$ to its own weight). Therefore, if a pattern has few words, its words will be assigned higher weights than words of longer patterns. This bias becomes more noticeable if the same word appears several times in the pattern, for example, *interface *{1,1} down: interface *{1,2} fault*. The word weight function f_2 addresses this shortcoming by first identifying the set of unique words U for the pattern. For instance, for the previous example pattern $U = \{interface, down:, fault\}$. If U contains p words (i.e., $p = |U|$), f_2 is defined as follows:

$$f_2(w) = ((\textstyle\sum_{v \in U} dep(v, w)) - 1) / (p - 1), \quad \text{if } p > 1$$

$$f_2(w) = 1, \quad \text{if } p = 1$$

For example, if $dep(Interface, eth0)$ = 0.07 and $dep(down, eth0)$ = 0.07, then $f_1(eth0)$ = 0.38 for line pattern *Interface eth0 down*, while $f_2(eth0)$ = (0.07 + 0.07 + 1 - 1) / 2 = 0.07. By not allowing the word to contribute to its own weight, f_2 calculates word weights in a fair way for shorter patterns.

One shortcoming of f_1 and f_2 weight functions is their inability to assign higher weights to groups of strongly associated words. For instance, suppose the line pattern is *kernel: interface *{1,1} down*, and the words *interface* and *down* always appear together (i.e., $dep(interface, down) = dep(down, interface)$ = 1). Also, suppose the word *kernel:* is always present when words *interface* and *down* appear (i.e., $dep(interface, kernel:) = dep(down, kernel:)$ = 1), while only 4% of event log lines that contain *kernel:* also contain the words *interface* and *down* (i.e., $dep(kernel:, interface) = dep(kernel:, down)$ = 0.04). Thus, $f_1(kernel:)$ = 1 and $f_1(interface) = f_1(down)$ = 0.68, although word weights should be distributed more evenly in this pattern, considering that words *interface* and *down* form a very strong sub-pattern. For achieving this purpose, the mutual dependency between frequent words a and b is defined as:

$$mdep(a, b) = (dep(a, b) + dep(b, a)) / 2$$

Note that $mdep(a, b) = mdep(b, a)$ and $0 < mdep(a, b) \leq 1$. Also, high values of $mdep(a, b)$ indicate that words a and b usually appear together, while low values reflect the lack of such strong association. If $w_1,\ldots,w_k$ are words in the line pattern, the word weight function f_3 is defined as:

$$f_3(w_i) = \sum_{j=1}^{k} mdep(w_j, w_i) / k$$

Like the f_2 function, the word weight function f_4 employs the set of unique words U for the pattern, and is defined as follows (note that $p = |U|$):

$$f_4(w) = ((\sum_{v \in U} mdep(v, w)) - 1) / (p - 1), \quad \text{if } p > 1$$

$$f_4(w) = 1, \quad \text{if } p = 1$$

For instance, in the case of the previous line pattern example $mdep(interface, down) = (1 + 1) / 2 = 1$, $mdep(kernel:, interface) = (0.04 + 1) / 2 = 0.52$, and $mdep(kernel:, down) = (0.04 + 1) / 2 = 0.52$. Therefore, $f_3(kernel:) = (1 + 0.52 + 0.52) / 3 = 0.68$, while $f_3(interface) = (0.52 + 1 + 1) / 3 = 0.84$ and $f_3(down) = (0.52 + 1 + 1) / 3 = 0.84$. In other words, compared to f_1, the f_3 function assigns more weight to words *interface* and *down* due to their strong association.

The LogCluster tool allows for choosing the weight function with the *--weightf* option, e.g., *--weightf=3* selects the f_3 function. Also, the *--color* option highlights words with insufficient weights in reported line patterns. Finally, detected cluster and word dependency information can be stored to the dump file given with the *--writedump* option. The data from previously created dump file can be loaded with the *--readdump* option during further runs of LogCluster. This is useful for quick evaluation of different word weight thresholds and functions without repeating the full clustering process that is computationally expensive. For example, the following command line

logcluster.pl --readdump=cluster.dump --wweight=0.75

loads the cluster and word dependency data from dump file *cluster.dump*, and joins clusters with the word weight threshold 0.75 and word weight function f_1 (the default function).

E. Case Studies

This subsection presents some log analysis scenarios that utilize previously described features of the LogCluster tool. Fig. 3 provides an example of employing LogCluster in a mid-size private organization, in order to create daily e-mail reports from syslog events of critical servers. Nearly 12 million events are collected each day with 150-300 detected line patterns. For producing more meaningful patterns, word classes are created for words which contain punctuation marks by replacing letters, digits, and other non-punctuation characters with character *X*. Replacement is not done if non-punctuation characters are followed by *[* or = character, in order to preserve keywords and program names which precede these characters (e.g., word class *to=<X@X.X>* is created for the word *to=<user@example.com>*). Also, the word weight threshold 0.5 and word weight function f_2 are used for joining clusters. The results of the clustering are written into the */tmp/logcluster-rotate.dmp* dump file, in order to facilitate quick additional analysis with different word weight thresholds and functions. Fig. 3 also depicts some example patterns detected with the LogCluster tool. The first pattern in Fig. 3 manifests an attempt to use the organizational DNS server for conducting DNS reflection and amplification attacks. The second pattern represents SSH account probing from several Internet hosts against a number of servers and routers of the organization. Remaining patterns reflect various attempts to distribute spam through the organizational mail server.

```
logcluster.pl --input=/var/log/all.log --rsupport=0.01 \
--wfilter='[[:punct:]]' --wsearch='[^[:punct:]]++(?![[=])' \
--wreplace=X --writedump=/tmp/logcluster-rotate.dmp \
--wweight=0.5 --weightf=2 --csize=100000 --wsize=100000

Mar 29 X:X:X nameserver2 named[10307]: security: info: client
(X.X.X.X#X:|10.0.137.69#25345:) view authoritative: query (cache)
('X.X.X.X.X.X/X/X'|'X.X.X.X/X/X'|'X.X.X.X.X/X/X'|'domain.nu/MX/IN'
|'isc.org/ANY/IN') denied
Support: 198152

Mar 29 X:X:X (backupserver|vps1|nameserver1|router1|vps2|router2
|mailserver|logserver|vps3|vps4) sshd[X]: pam_unix(sshd:auth):
authentication failure; logname= uid=0 euid=0 tty=ssh ruser=
(rhost=10.3.202.120|rhost=10.88.177.98) user=root
Support: 18112

Mar 26 X:X:X mailserver X/smtpd[X]: NOQUEUE: reject: RCPT from
exch001.example.com[10.52.134.35]: 454 4.7.1 <user@example.com>:
Relay access denied; from=<> to=<user@example.com> proto=ESMTP
helo=<webmail.example.com>
Support: 941

Mar 17 X:X:X mailserver X/smtpd[X]: warning: hostname
host94165.example.com does not resolve to address X.X.X.X
Support: 1217

Mar 9 X:X:X mailserver X/smtpd[X]: NOQUEUE: reject: RCPT from
unknown[X.X.X.X]: 554 5.7.1 Service unavailable; Client host
[X.X.X.X] blocked using cbl.abuseat.org;
Blocked - see X://X.X.X/X.X?ip=X.X.X.X; from=<X@X.X> to=<X@X.X>
proto=ESMTP *{1,1}
Support: 1219
…
```

Fig. 3. Sample attack patterns detected from syslog events.

In some cases, it might not be convenient to cluster the event log with one LogCluster run, since higher support threshold might yield too many outliers, while with lower support threshold a large number of clusters might be produced. This problem often appears for event logs which contain events from many servers and programs, and feature meaningful line patterns with a wide variety of supports. For addressing this problem, LogCluster can be used iteratively, clustering results from previous execution(s) at each step. Fig. 4 provides an example of iterative clustering of the *mail.log* file which contained syslog messages with *mail* facility from a number of mail servers. During the first iteration with relative support threshold 0.1%, each event log line was converted to a program name string, so that detected line patterns indicated programs that have produced most log messages in *mail.log*. A cluster for the *sendmail* daemon was discovered, and during the second iteration with relative support threshold 0.1% it was split into smaller clusters by analyzing the message text after the program name. The second iteration yielded 268 patterns that reflected normal system activity and 10,264 outliers. The outliers were clustered further with support threshold 50. As a result, 105 patterns and 2018 outliers were detected, with many patterns and outliers representing error conditions and abnormal events such as connection attempts from spammers.

```
logcluster.pl --input=mail.log --rsupport=0.1 \
--lfilter=' ([\w\/.-]+)\[\d+\]: ' --template='$1[PID]:'

sendmail[PID]:
Support: 1007754
…

logcluster.pl --input=mail.log –rsupport=0.1 --aggrsup \
--lfilter='sendmail\[\d+\]: (.+)' --template='$1' \
--separator='(?:\s+|=)' --outliers=outliers.log

*{1,1} from *{1,5} size *{1,1} class 0, nrcpts 1, msgid *{1,5}
proto ESMTP, daemon MTA, relay *{1,5}
Support: 161976

STARTTLS client, relay *{1,1} version TLSv1/SSLv3, verify OK,
cipher AES128-SHA, bits 128/128
Support: 71062
…

logcluster.pl --input=outliers.log --support=50 --aggrsup \
--lfilter 'sendmail\[\d+\]: (.+)' --template '$1' \
--separator='(?:\s+|=)' --outliers=outliers2.log

*{1,1} ruleset check_rcpt, arg1 *{1,1} relay *{1,2} reject 550
5.1.1 *{1,1} User unknown
Support: 441

*{1,1} SYSERR(root): collect: I/O error on connection from *{1,1}
from *{1,2}
Support: 104
…

# examples of outlier events from outliers2.log

Mar 29 03:51:28 mailserver sendmail[30101]: ruleset=check_relay,
  arg1=box.example.com, arg2=127.0.0.1, reject=550 5.7.1
  Rejected: 10.193.172.92 listed at xbl.spamhaus.org
Mar 29 08:28:07 mailserver sendmail[6276]: XXX: mail.example.com
  [10.109.254.117]: Possible SMTP RCPT flood, throttling.
Mar 29 10:23:58 mailserver sendmail[22746]: XXX:
  ruleset=check_mail, arg1=<pzfsibdlkj@sjfqc.biz>,
  relay=[10.240.79.7], reject=553 5.1.8 <pzfsibdlkj@sjfqc.biz>...
  Domain of sender address pzfsibdlkj@sjfqc.biz does not exist
```

Fig. 4. Iterative analysis of mail server events.

IV. Conclusion

In this paper, we have presented the LogCluster tool for mining line patterns and outlier events from textual event logs. We have also described several scenarios of discovering security incidents and anomalous events with this tool. For a more detailed information on its performance and comparison with other log clustering algorithms, the reader is referred to our recent paper [7].

For the future work, we plan to harness the LogCluster tool for insider threat detection and to modify the LogCluster algorithm for stream mining purposes. The LogCluster tool has been released under the terms of GNU GPLv2 and is available from *http://ristov.github.io/logcluster*.

Acknowledgment

The authors thank Mr. Kaido Raiend, Mr. Ain Rasva, Mr. Raimo Peterson, Mrs. Katrin Kriiska, Prof. Olaf M. Maennel, and Dr. Rain Ottis for supporting this work.

References

[1] C. Lonvick, "The BSD syslog Protocol," RFC3164, 2001.

[2] R. Gerhards, "The Syslog Protocol," RFC5424, 2009.

[3] http://www.rsyslog.com

[4] https://www.balabit.com/network-security/syslog-ng

[5] https://www.elastic.co

[6] 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.

[7] 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.

[8] 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.

[9] Risto Vaarandi, "Mining Event Logs with SLCT and LogHound," in Proceedings of the 2008 IEEE/IFIP Network Operations and Management Symposium, pp. 1071-1074.

[10] Risto Vaarandi and Kārlis Podiņš, "Network IDS Alert Classification with Frequent Itemset Mining and Data Clustering," in Proceedings of the 2010 International Conference on Network and Service Management, pp. 451-456.

[11] Thomas Reidemeister, Miao Jiang and Paul A.S. Ward, "Mining Unstructured Log Files for Recurrent Fault Diagnosis," in Proceedings of the 2011 IEEE/IFIP International Symposium on Integrated Network Management, pp. 377-384.

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

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

[14] Mika Klemettinen, "A Knowledge Discovery Methodology for Telecommunication Network Alarm Databases," PhD thesis, University of Helsinki, 1999.

[15] Qingguo Zheng, Ke Xu, Weifeng Lv and Shilong Ma, "Intelligent Search of Correlated Alarms from Database Containing Noise Data," in Proceedings of the 2002 IEEE/IFIP Network Operations and Management Symposium, pp. 405-419.

[16] 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.

[17] James J. Treinen and Ramakrishna Thurimella, "A Framework for the Application of Association Rule Mining in Large Intrusion Detection Infrastructures," in Proceedings of the 2006 Symposium on Recent Advances in Intrusion Detection, LNCS Vol. 4219, Springer, pp. 1-18.

[18] Chris Clifton and Gary Gengo, "Developing Custom Intrusion Detection Filters Using Data Mining," in Proceedings of the 2000 IEEE Military Communications Conference, pp. 440-443.

[19] Jon Stearley, "Towards Informatic Analysis of Syslogs," in Proceedings of the 2004 IEEE International Conference on Cluster Computing, pp. 309–318.

[20] Adetokunbo Makanju, Stephen Brooks, A. Nur Zincir-Heywood and Evangelos E. Milios, "LogView: Visualizing Event Log Clusters," in Proceedings of the 6th Annual Conference on Privacy, Security and Trust, pp. 99-108, 2008.

[21] Daniela Brauckhoff, Xenofontas Dimitropoulos, Arno Wagner and Kavè Salamatian, "Anomaly Extraction in Backbone Networks using Association Rules," in Proceedings of the 2009 ACM SIGCOMM Internet Measurement Conference, pp. 28-34.

[22] Eduard Glatz, Stelios Mavromatidis, Bernhard Ager and Xenofontas Dimitropoulos, "Visualizing big network traffic data using frequent pattern mining and hypergraphs," Computing Vol. 96(1), Springer, pp. 27-38, 2014.

[23] Risto Vaarandi, "Simple Event Correlator for real-time security log monitoring," Hakin9 Magazine 1/2006 (6), pp. 28-39, 2006.