LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs

Risto Vaarandi and Mauno Pihelgas

LogCluster - A Data Clustering and Pattern Mining Algorithm for Event Logs

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

Abstract—**Modern IT systems often produce large volumes of event logs, and event pattern discovery is an important log management task. For this purpose, data mining methods have been suggested in many previous works. In this paper, we present the LogCluster algorithm which implements data clustering and line pattern mining for textual event logs. The paper also describes an open source implementation of LogCluster.**

Keywords—event log analysis; mining patterns from event logs; event log clustering; data clustering; data mining

I. INTRODUCTION

During the last decade, data centers and computer networks have grown significantly in processing power, size, and complexity. As a result, organizations commonly have to handle many gigabytes of log data on a daily basis. For example, in our recent paper we have described a security log management system which receives nearly 100 million events each day [1]. In order to ease the management of log data, many research papers have suggested the use of data mining methods for discovering event patterns from event logs [2–20]. This knowledge can be employed for many different purposes like the development of event correlation rules [12–16], detection of system faults and network anomalies [6–9, 19], visualization of relevant event patterns [17, 18], identification and reporting of network traffic patterns [4, 20], and automated building of IDS alarm classifiers [5].

In order to analyze large amounts of textual log data without well-defined structure, several data mining methods have been proposed in the past which focus on the detection of line patterns from textual event logs. Suggested algorithms have been mostly based on data clustering approaches [2, 6, 7, 8, 10, 11]. The algorithms assume that each event is described by a single line in the event log, and each line pattern represents a group of similar events.

In this paper, we propose a novel data clustering algorithm called LogCluster which discovers both frequently occurring line patterns and outlier events from textual event logs. The remainder of this paper is organized as follows – section II provides an overview of related work, section III presents the LogCluster algorithm, section IV describes the LogCluster prototype implementation and experiments for evaluating its performance, and section V concludes the paper.

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 that is designed for mining line patterns and outlier events from textual event logs [2]. During the clustering process, SLCT assigns event log lines that fit the same pattern (e.g., *Interface * down*) to the same cluster, and all detected clusters are reported to the user as line patterns. For finding clusters in log data, the user has to supply the support threshold value *s* to SLCT which defines the minimum number of lines in each cluster. SLCT begins the clustering with a pass over the input data set, in order to identify frequent words which occur at least in *s* lines (word delimiter is customizable and defaults to whitespace). Also, each word is considered with its position in the line. For example, if *s=2* and the data set contains the lines

Interface eth0 down

Interface eth1 down

Interface eth2 up

then words *(Interface,1)* and *(down,3)* occur in three and two lines, respectively, and are thus identified as frequent words. SLCT will then make another pass over the data set and create cluster candidates. When a line is processed during the data pass, all frequent words from the line are joined into a set which will act as a candidate for this line. After the data pass, candidates generated for at least *s* lines are reported as clusters together with their supports (occurrence times). Outliers are identified during an optional data pass and written to a user-specified file. For example, if *s=2* then two cluster candidates *{(Interface,1), (down,3)}* and *{(Interface,1)}* are detected with supports 2 and 1, respectively. Thus, *{(Interface,1), (down,3)}* is the only cluster and is reported to the user as a line pattern *Interface * down* (since there is no word associated with the second position, an asterisk is printed for denoting a wildcard). Reported cluster covers the first two lines, while the line *Interface eth2 up* is considered an outlier.

SLCT has several shortcomings which have been pointed out in some recent works. Firstly, it is not able to detect wildcards after the last word in a line pattern [11]. For instance, if *s=3* for three example lines above, the cluster *{(Interface,1)}* is reported to the user as a line pattern *Interface*, although most users would prefer the pattern *Interface * **. Secondly, since word positions are encoded into words, the algorithm is sensitive to shifts in word positions and delimiter noise [8]. For instance, the line *Interface HQ Link down* would not be assigned to the cluster *Interface * down*, but would rather generate a separate cluster candidate. Finally, low support thresholds can lead to overfitting when larger clusters are split and resulting patterns are too specific [2].

Reidemeister, Jiang, Munawar and Ward [6, 7, 8] developed a methodology that addresses some of the above shortcomings. The methodology uses event log mining techniques for diagnosing recurrent faults in software systems. First, a modified version of SLCT is used for mining line patterns from labeled event logs. In order to handle clustering errors caused by shifts in word positions and delimiter noise, line patterns from SLCT are clustered with a single-linkage clustering algorithm which employs a variant of the Levenshtein distance function. After that, a common line pattern description is established for each cluster of line patterns. According to [8], single-linkage clustering and post-processing its results add minimal runtime overhead to the clustering by SLCT. The final results are converted into bit vectors and used for building decision-tree classifiers, in order to identify recurrent faults in future event logs.

Another clustering algorithm that mines line patterns from event logs is IPLoM by Makanju, Zincir-Heywood and Milios [10, 11]. Unlike SLCT, IPLoM is a hierarchical clustering algorithm which starts with the entire event log as a single partition, and splits partitions iteratively during three steps. Like SLCT, IPLoM considers words with their positions in event log lines, and is therefore sensitive to shifts in word positions. During the first step, the initial partition is split by assigning lines with the same number of words to the same partition. During the second step, each partition is divided further by identifying the word position with the least number of unique words, and splitting the partition by assigning lines with the same word to the same partition. During the third step, partitions are split based on associations between word pairs. At the final stage of the algorithm, a line pattern is derived for each partition. Due to its hierarchical nature, IPLoM does not need the support threshold, but takes several other parameters (such as partition support threshold and cluster goodness threshold) which impose fine-grained control over splitting of partitions [11]. As argued in [11], one advantage of IPLoM over SLCT is its ability to detect line patterns with wildcard tails (e.g., *Interface * **), and the author has reported higher precision and recall for IPLoM.

III. THE LOGCLUSTER ALGORITHM

The LogCluster algorithm is designed for addressing the shortcomings of existing event log clustering algorithms that were discussed in the previous section. Let $L = \{l_1,\dots,l_n\}$ be a textual event log which consists of n lines, where each line l_i $(1 \leq i \leq n)$ is a complete representation of some event and i is a unique line identifier. We assume that each line $l_i \in L$ is a sequence of k_i words: $l_i = (w_{i,1},\dots,w_{i,k_i})$. LogCluster takes the *support threshold* s $(1 \leq s \leq n)$ as a user given input parameter and divides event log lines into clusters $C_1,\dots,C_m$, so that there are at least s lines in each cluster C_j (i.e., $|C_j| \geq s$) and O is the cluster of outliers: $L = C_1 \cup \dots \cup C_m \cup O$, $O \cap C_j = \varnothing$, $1 \leq j \leq m$. LogCluster views the log clustering problem as a pattern mining problem – each cluster C_j is uniquely identified by its line pattern p_j which matches all lines in the cluster, and in order to detect clusters, LogCluster mines line patterns p_j from the event log. The *support* of pattern p_j and cluster C_j is defined as the number of lines in C_j: $supp(p_j) = supp(C_j) = |C_j|$. Each pattern consists of words and wildcards, e.g., *Interface *{1,3} down* has words *Interface* and *down*, and wildcard **{1,3}* that matches at least 1 and at most 3 words.

In order to find patterns that have the support s or higher, LogCluster relies on the following observation – all words of such patterns must occur at least in s event log lines. Therefore, LogCluster begins its work with the identification of such words. However, unlike SLCT and IPLoM, LogCluster considers each word without its position in the event log line. Formally, let I_w be the set of identifiers of lines that contain the word w: $I_w = \{i \mid l_i \in L, 1 \leq i \leq n, \exists j\ w_{i,j} = w, 1 \leq j \leq k_i\}$. The word w is *frequent* if $|I_w| \geq s$, and the set of all frequent words is denoted by F. According to [2, 3], large event logs often contain many millions of different words, while vast majority of them appear only few times in event logs. In order to take advantage of this property for reducing its memory footprint, LogCluster employs a sketch of h counters $c_0,\dots,c_{h-1}$. During a preliminary pass over event log L, each unique word of every event log line is hashed to an integer from 0 to h-1, and the corresponding sketch counter is incremented. Since the hashing function produces output values $0\dots h$-1 with equal probabilities, each sketch counter reflects the sum of occurrence times of approximately d / h words, where d is the number of unique words in L. However, since most words appear in only few lines of L, most sketch counters will be smaller than support threshold s after the data pass. Thus, corresponding words cannot be frequent, and can be ignored during the following pass over L for finding frequent words.

After frequent words have been identified, LogCluster makes another pass over event log L and creates cluster candidates. For each line in the event log, LogCluster extracts all frequent words from the line and arranges the words as a tuple, retaining their original order in the line. The tuple will serve as an identifier of the cluster candidate, and the line is assigned to this candidate. If the given candidate does not exist, it is initialized with the support counter set to 1, and its line pattern is created from the line. If the candidate exists, its support counter is incremented and its line pattern is adjusted to cover the current line. Note that LogCluster does not memorize individual lines assigned to a cluster candidate.

For example, if the event log line is *Interface DMZ-link down at node router2*, and words *Interface*, *down*, *at*, and *node* are frequent, the line is assigned to the candidate identified by the tuple *(Interface, down, at, node)*. If this candidate does not exist, it will be initialized by setting its line pattern to *Interface *{1,1} down at node *{1,1}* and its support counter to 1 (wildcard **{1,1}* matches any single word). If the next line which produces the same candidate identifier is *Interface HQ link down at node router2*, the candidate support counter is incremented to 2. Also, its line pattern is set to *Interface *{1,2} down at node *{1,1}*, making the pattern to match at least one but not more than two words between *Interface* and *down*. Fig. 1 describes the candidate generation procedure in full details.

```
Procedure: Generate_Candidates
Input: event log L = {l1,…,ln}
       set of frequent words F
Output: set of cluster candidates X

X := ∅
for (id = 1; id <= n; ++id) do
  tuple := ()
  vars := ()
  i := 0; v := 0
  for each w in (w_id,1,…,w_id,k_id) do
    if (w ∈ F) then
      tuple[i] := w
      vars[i] := v
      ++i; v := 0
    else
      ++v
    fi
  done
  vars[i] := v
  k := # of elements in tuple
  if (k > 0) then
    if (∃Y ∈ X, Y.tuple == tuple) then
      ++Y.support
      for (i := 0; i < k+1; ++i) do
        if (Y.varmin[i] > vars[i]) then
          Y.varmin[i] := vars[i]
        fi
        if (Y.varmax[i] < vars[i]) then
          Y.varmax[i] := vars[i]
        fi
      done
    else
      initialize new candidate Y
      Y.tuple := tuple
      Y.support := 1
      for (i := 0; i < k+1; ++i) do
        Y.varmin[i] := vars[i]
        Y.varmax[i] := vars[i]
      done
      X := X ∪ { Y }
    fi
    Y.pattern = ()
    j: = 0
    for (i := 0; i < k; ++i) do
      if (Y.varmax[i] > 0) then
        min := Y.varmin[i]
        max := Y.varmax[i]
        Y.pattern[j] := "*{min,max}"
        ++j
      fi
      Y.pattern[j] := tuple[i]
      ++j
    done
    if (Y.varmax[k] > 0) then
      min := Y.varmin[k]
      max := Y.varmax[k]
      Y.pattern[j] := "*{min,max}"
    fi
  fi
done
return X
```

Fig. 1. Candidate generation procedure of LogCluster.

After the data pass for generating cluster candidates is complete, LogCluster drops all candidates with the support counter value smaller than support threshold s, and reports remaining candidates as clusters. For each cluster, its line pattern and support are reported, while outliers are identified during additional pass over event log L. Due to the nature of its frequent word detection and candidate generation procedures, LogCluster is not sensitive to shifts in word positions and is able to detect patterns with wildcard tails.

When pattern mining is conducted with lower support threshold values, LogCluster is (similarly to SLCT) prone to overfitting – larger clusters might be split into smaller clusters with too specific line patterns. For example, the cluster with a pattern *Interface *{1,1} down* could be split into clusters with patterns *Interface *{1,1} down*, *Interface eth1 down*, and *Interface eth2 down*. Furthermore, meaningful generic patterns (e.g., *Interface *{1,1} down*) might disappear during cluster splitting. In order to address the overfitting problem, LogCluster employs two optional heuristics for increasing the support of more generic cluster candidates and for joining clusters. The first heuristic is called *Aggregate_Supports* and is applied after the candidate generation procedure has been completed, immediately before clusters are selected. The heuristic involves finding candidates with more specific line patterns for each candidate, and adding supports of such candidates to the support of the given candidate. For instance, if candidates *User bob login from 10.1.1.1*, *User *{1,1} login from 10.1.1.1*, and *User *{1,1} login from *{1,1}* have supports 5, 10, and 100, respectively, the support of the candidate *User *{1,1} login from *{1,1}* will be increased to 115. In other words, this heuristic allows clusters to overlap.

The second heuristic is called *Join_Clusters* and is applied after clusters have been selected from candidates. For each frequent word $w \in F$, we define the set C_w as follows: $C_w = \{f \mid f \in F, I_w \cap I_f \neq \emptyset\}$ (i.e., C_w contains all frequent words that co-occur with w in event log lines). If $w' \in C_w$ (i.e., w' co-occurs with w), we define *dependency* from w to w' as $dep(w, w') = |I_w \cap I_{w'}| / |I_w|$. In other words, $dep(w, w')$ reflects how frequently w' occurs in lines which contain w. Also, note that $0 < dep(w, w') \leq 1$. If $w_1,\ldots,w_k$ are frequent words of a line pattern (i.e., the corresponding cluster is identified by the tuple $(w_1,\ldots,w_k)$), the *weight* of the word w_i in this pattern is calculated as follows: $weight(w_i) = \sum_{j=1}^{k} dep(w_j, w_i) / k$. Note that since $dep(w_i, w_i) = 1$, then $1/k \leq weight(w_i) \leq 1$. Intuitively, the weight of the word indicates how strongly correlated the word is with other words in the pattern. For example, suppose the line pattern is *Daemon testd killed*, and words *Daemon* and *killed* always appear together, while the word *testd* never occurs without *Daemon* and *killed*. Thus, *weight(Daemon)* and *weight(killed)* are both 1. Also, if only 2.5% of lines that contain both *Daemon* and *killed* also contain *testd*, then *weight(testd)* = (1 + 0.025 + 0.025) / 3 = 0.35. (We plan to implement more weight functions in the future versions of the LogCluster prototype.)

The *Join_Clusters* heuristic takes the user supplied word weight threshold t as its input parameter ($0 < t \leq 1$). For each cluster, a secondary identifier is created and initialized to the cluster's regular identifier tuple. Also, words with weights smaller than t are identified in the cluster's line pattern, and each such word is replaced with a special token in the secondary identifier. Finally, clusters with identical secondary identifiers are joined. When two or more clusters are joined, the support of the joint cluster is set to the sum of supports of original clusters, and the line pattern of the joint cluster is adjusted to represent the lines in all original clusters.

```
Procedure: Join_Clusters
Input: set of clusters C = {C₁,…,Cₚ}
       word weight threshold t
       word weight function W()
Output: set of clusters C′ = {C′₁,…,C′ₘ}, m ≤ p

C′ := ∅
for (j = 1; j <= p; ++j) do
  tuple := Cⱼ.tuple
  k := # of elements in tuple
  for (i := 0; i < k; ++i) do
    if (W(tuple, i) < t) then
      tuple[i] := TOKEN
    fi
  done
  if (∃Y ∈ C′, Y.tuple == tuple) then
    Y.support := Y.support + Cⱼ.support
    for (i := 0; i < k+1; ++i) do
      if (Y.varmin[i] > Cⱼ.varmin[i]) then
        Y.varmin[i] := Cⱼ.varmin[i]
      fi
      if (Y.varmax[i] < Cⱼ.varmax[i]) then
        Y.varmax[i] := Cⱼ.varmax[i]
      fi
    done
  else
    initialize new cluster Y
    Y.tuple := tuple
    Y.support := Cⱼ.support
    for (i := 0; i < k+1; ++i) do
      Y.varmin[i] := Cⱼ.varmin[i]
      Y.varmax[i] := Cⱼ.varmax[i]
      if (i < k AND Y.tuple[i] == TOKEN) then
        Y.wordlist[i] := ∅
      fi
    done
    C′ := C′ ∪ { Y }
  fi
  Y.pattern := ()
  j: = 0
  for (i := 0; i < k; ++i) do
    if (Y.varmax[i] > 0) then
      min := Y.varmin[i]
      max := Y.varmax[i]
      Y.pattern[j] := "*{min,max}"
      ++j
    fi
    if (Y.tuple[i] == TOKEN) then
      if (Cⱼ.tuple[i] ∉ Y.wordlist[i]) then
        Y.wordlist[i] :=
          Y.wordlist[i] ∪ { Cⱼ.tuple[i] }
      fi
      Y.pattern[j] := "( elements of
          Y.wordlist[i] separated by | )"
    else
      Y.pattern[j] := Y.tuple[i]
    fi
    ++j
  done
  if (Y.varmax[k] > 0) then
    min := Y.varmin[k]
    max := Y.varmax[k]
    Y.pattern[j] := "*{min,max}"
  fi
done
return C′
```

Fig. 2. Cluster joining heuristic of LogCluster.

For example, if two clusters have patterns *Interface *{1,1} down at node router1* and *Interface *{2,3} down at node router2*, and words *router* and *router2* have insufficient weights, the clusters are joined into a new cluster with the line pattern *Interface *{1,3} down at node (router1|router2)*. Fig. 2 describes the details of the *Join_Clusters* heuristic. Since the line pattern of a joint cluster consists of strongly correlated words, it is less likely to suffer from overfitting. Also, words with insufficient weights are incorporated into the line pattern as lists of alternatives, representing the knowledge from original patterns in a compact way without data loss. Finally, joining clusters will reduce their number and will thus make cluster reviewing easier for the human expert.

Fig. 3 summarizes all techniques presented in this section and outlines the LogCluster algorithm. In the next section, we describe the LogCluster implementation and its performance.

IV. LogCluster Implementation and Performance

For assessing the performance of the LogCluster algorithm, we have created its publicly available GNU GPLv2 licensed prototype implementation in Perl. The implementation is a UNIX command line tool that can be downloaded from *http://ristov.github.io/logcluster*. Apart from its clustering capabilities, the LogCluster tool supports a number of data preprocessing options which are summarized below. In order to focus on specific lines during pattern mining, a regular expression filter can be defined with the *--lfilter* command line option. For instance, with *--lfilter='sshd\[\d+\]:'* patterns are detected for *sshd* syslog messages (e.g., *May 10 11:07:12 myhost sshd[4711]: Connection from 10.1.1.1 port 5662*).

```
Procedure: LogCluster
Input: event log L = {l₁,…,lₙ}
       support threshold s
       word sketch size h  (optional)
       word weight threshold t  (optional)
       word weight function W()  (optional)
       boolean for invoking Aggregate_Supports
                   procedure A  (optional)
       file of outliers ofile  (optional)
Output: set of clusters C = {C₁,…,Cₘ}
        the cluster of outliers O (optional)

1. if (defined(h)) then
   make a pass over L and build the word sketch
   of size h for filtering out infrequent words
   at step 2
2. make a pass over L and find the set of
   frequent words:  F := {w | |I_w| ≥ s}
3. if (defined(t)) then
   make a pass over L and find dependencies for
   frequent words: {dep(w, w′) | w ∈ F, w′ ∈ C_w}
4. make a pass over L and find the set of cluster
   candidates X:  X := Generate_Candidates(L, F)
5. if (defined(A) AND A == TRUE) then
   invoke Aggregate_Supports() procedure
6. find the set of clusters C
   C := {Y ∈ X | supp(Y) ≥ s}
7. if (defined(t)) then
   join clusters:  C := Join_Clusters(C, t, W)
8. report line patterns and their supports
   for clusters from set C
9. if (defined(ofile)) then
   make a pass over L and write outliers to ofile
```

Fig. 3. The LogCluster algorithm.

If a template string is given with the *--template* option, match variables set by the regular expression of the *--lfilter* option are substituted in the template string, and the resulting string replaces the original event log line during the mining. For example, with the use of *--lfilter='(sshd\[\d+\]: .+)'* and *--template='$1'* options, timestamps and hostnames are removed from *sshd* syslog messages before any other processing. If a regular expression is given with the *--separator* option, any sequence of characters that matches this expression is treated as a word delimiter (word delimiter defaults to whitespace).

Existing line pattern mining tools treat words as atoms during the mining process, and make no attempt to discover potential structure inside words (the only exception is SLCT which includes a simple post-processing option for detecting constant heads and tails for wildcards). In order to address this shortcoming, LogCluster implements several options for masking specific word parts and creating word classes. If a word matches the regular expression given with the *--wfilter* option, a word class is created for the word by searching it for substrings that match another regular expression provided with the *--wsearch* option. All matching substrings are then replaced with the string specified with the *--wreplace* option. For example, with the use of *--wfilter='='*, *--wsearch='=.+'*, and *--wreplace='=VALUE'* options, word classes are created for words which contain the equal sign (=) by replacing the characters after the equal sign with the string *VALUE*. Thus, for words *pid=12763* and *user=bob*, classes *pid=VALUE* and *user=VALUE* are created. If a word is infrequent but its word class is frequent, the word class replaces the word during the mining process and will be treated like a frequent word. Since classes can represent many infrequent words, their presence in line patterns provides valuable information about regularities in word structure that would not be detected otherwise.

For evaluating the performance of LogCluster and comparing it with other algorithms, we conducted a number of experiments with larger event logs. For the sake of fair comparison, we re-implemented the public C-based version of SLCT in Perl. Since the implementations of IPLoM and the algorithm by Reidemeister et al. are not publicly available, we were unable to study their source code for creating their exact prototypes. However, because the algorithm by Reidemeister et al. uses SLCT and has a similar time complexity (see section II), its runtimes are closely approximated by results for SLCT. During our experiments, we used 6 logs from a large institution of a national critical information infrastructure of an EU state. The logs cover 24 hour timespan (May 8, 2015), and originate from a wide range of sources, including database systems, web proxies, mail servers, firewalls, and network devices. We also used an availability monitoring system event log from the NATO CCD COE Locked Shields 2015 cyber defense exercise which covers the entire two-day exercise and contains Nagios events. During the experiments, we clustered each log file three times with support thresholds set to 1%, 0.5% and 0.1% of lines in the log. We also used the word sketch of 100,000 counters (parameter h in Fig. 3) for both LogCluster and SLCT, and did not employ *Aggregate_Supports* and *Join_Clusters* heuristics. Therefore, both LogCluster and SLCT were configured to make three passes over the data set, in order to build the word sketch during the first pass, detect frequent words during the second pass, and generate cluster candidates during the third pass. All experiments were conducted on a Linux virtual server with Intel Xeon E5-2680 CPU and 64GB of memory, and Table I outlines the results. Since LogCluster and SLCT implementations are both single-threaded and their CPU utilization was 100% according to Linux *time* utility during all 21 experiments, each runtime in Table I closely matches the consumed CPU time.

TABLE I. PERFORMANCE OF LOGCLUSTER AND SLCT

Row #	Event log type	Event log size in megabytes	Event log size in lines	Support threshold	Number of clusters found by LogCluster	LogCluster runtime in seconds	Number of clusters found by SLCT	SLCT runtime in seconds
1	Authorization messages	3800.1	7,757,440	7,757	49	3146.42	89	1969.04
2	Authorization messages	3800.1	7,757,440	38,787	32	3070.18	37	1892.41
3	Authorization messages	3800.1	7,757,440	77,574	9	3050.20	15	1911.93
4	UNIX daemon messages	740.2	5,778,847	5,778	150	692.08	158	479.90
5	UNIX daemon messages	740.2	5,778,847	28,894	40	682.95	44	462.85
6	UNIX daemon messages	740.2	5,778,847	57,788	12	667.82	16	470.48
7	Application messages	9363.0	34,516,290	34,516	109	5225.32	114	3674.47
8	Application messages	9363.0	34,516,290	172,581	16	4891.51	25	3559.36
9	Application messages	9363.0	34,516,290	345,162	5	4765.09	8	3517.67
10	Network device messages	4705.0	12,522,620	12,522	193	3181.97	195	2015.52
11	Network device messages	4705.0	12,522,620	62,613	31	3083.16	33	2000.98
12	Network device messages	4705.0	12,522,620	125,226	17	3080.66	19	1945.69
13	Web proxy messages	16681.5	49,376,464	49,376	105	8487.37	111	5409.23
14	Web proxy messages	16681.5	49,376,464	246,882	14	8128.34	14	5277.54
15	Web proxy messages	16681.5	49,376,464	493,764	5	8081.30	5	5244.96
16	Mail server messages	246.0	1,230,532	1,230	129	144.42	139	96.34
17	Mail server messages	246.0	1,230,532	6,152	40	141.83	40	96.85
18	Mail server messages	246.0	1,230,532	12,305	21	142.34	23	94.12
19	Nagios messages	391.9	3,400,185	3,400	45	435.76	46	316.77
20	Nagios messages	391.9	3,400,185	17,000	39	412.08	41	320.26
21	Nagios messages	391.9	3,400,185	34,001	19	409.87	22	318.25

```
May 8 *{1,1} myserver dhcpd: DHCPREQUEST for
*{1,2} from *{1,2} via *{1,4}

May 8 *{3,3} Note: no *{1,3} sensors

May 8 *{3,3} RT_IPSEC: %USER-3-RT_IPSEC_REPLAY:
Replay packet detected on IPSec tunnel on *{1,1}
with tunnel ID *{1,1} From *{1,1} to *{1,1} ESP,
SPI *{1,1} SEQ *{1,1}

May 8 *{1,1} myserver httpd: client *{1,1} request
GET *{1,1} HTTP/1.1 referer *{1,1} User-agent
Mozilla/5.0 *{3,4} rv:37.0) Gecko/20100101
Firefox/37.0 *{0,1}

May 8 *{1,1} myserver httpd: client *{1,1} request
GET *{1,1} HTTP/1.1 referer *{1,1} User-agent
Mozilla/5.0 (Windows NT *{1,3} AppleWebKit/537.36
(KHTML, like Gecko) Chrome/42.0.2311.135
Safari/537.36
```

Fig. 4. Sample clusters detected by LogCluster (for the reasons of privacy, sensitive data have been obfuscated).

As results indicate, SLCT was 1.28–1.62 times faster than LogCluster. This is due to the simpler candidate generation procedure of SLCT – when processing individual event log lines, SLCT does not have to check the line patterns of candidates and adjust them if needed. However, both algorithms require considerable amount of time for clustering very large log files. For example, for processing the largest event log of 16.3GB (rows 13-15 in Table I), SLCT needed about 1.5 hours, while for LogCluster the runtime exceeded 2 hours. In contrast, the C-based version of SLCT accomplishes the same three tasks in 18-19 minutes. Therefore, we expect a C implementation of LogCluster to be significantly faster.

According to Table I, LogCluster finds less clusters than SLCT during all experiments (some clusters are depicted in Fig. 4). The reviewing of detected clusters revealed that unlike SLCT, LogCluster was able to discover a single cluster for lines where frequent words were separated with a variable number of infrequent words. For example, the first cluster in Fig. 4 properly captures all DHCP request events. In contrast, SLCT discovered two clusters *May 8 * myserver dhcpd: DHCPREQUEST for * from * * via* and *May 8 * myserver dhcpd: DHCPREQUEST for * * from * * via* which still do not cover all possible event formats. Also, the last two clusters in Fig. 4 represent all HTTP requests originating from the latest stable versions of Firefox browser on all OS platforms and Chrome browser on all Windows platforms, respectively (all OS platform strings are matched by **{3,4}* for Firefox, while *Windows NT *{1,3}* matches all Windows platform strings for Chrome). Like in the previous case, SLCT was unable to discover equivalent two clusters that would concisely capture HTTP request events for these two browser types.

When evaluating the *Join_Clusters* heuristic, we found that word weight thresholds (parameter t in Fig. 3) between 0.5 and 0.8 produced the best joint clusters. Fig. 5 displays three sample joint clusters which were detected from the mail server and Nagios logs (rows 16-21 in Table I). Fig. 5 also illustrates data preprocessing capabilities of the LogCluster tool. For the mail server log, a word class is created for each word which contains punctuation marks, so that all sequences of non-punctuation characters which are not followed by the equal sign (=) or opening square bracket ([) are replaced with a single *X* character. For the Nagios log, word classes are employed for masking blue team numbers in host names, and also, trailing timestamps are removed from each event log line with *--lfilter* and *--template* options. The first two clusters in Fig. 5 are both created by joining three clusters, while the last cluster is the union of twelve clusters which represent Nagios SSH service check events for 192 servers.

```
logcluster.pl --support=12305 \
--input=mail.log --wfilter='[[:punct:]]' \
--wsearch='[^[:punct:]]++(?![[=])' \
--wreplace=X --wweight=0.75

May 8 X:X:X (myserver1|myserver2|myserver3)
sendmail[X]: STARTTLS=client,
(relay=relayserver1,|relay=relayserver2,
|relay=relayserver3,) version=TLSv1/SSLv3,
(verify=FAIL,|verify=OK,) (cipher=DHE-RSA-AES256-
SHA,|cipher=AES128-SHA,|cipher=RC4-SHA,)
(bits=256/256|bits=128/128)

May 8 X:X:X (myserver1|myserver2|myserver3)
sendmail[X]: X: from=<myrobot@mydomain>, size=X,
class=0, nrcpts=1, msgid=<X.X@X.X>,
bodytype=8BITMIME, proto=ESMTP, daemon=MTA,
(relay=relayserver1|relay=relayserver2)
([ipaddress1]|[ipaddress2])


logcluster.pl --support=3400 \
--input=ls15.log --separator='["|\s]+' \
--lfilter='^(.*)(?:\|"\d+"){2}' --template='$1' \
--wfilter='blue\d\d' --wsearch='blue\d\d' \
--wreplace='blueNN' --wweight=0.5

(ws4-01.lab.blueNN.ex|ws4-04.lab.blueNN.ex
|ws4-03.int.blueNN.ex|ws4-04.int.blueNN.ex
|ws4-02.int.blueNN.ex|ws4-05.lab.blueNN.ex
|ws4-05.int.blueNN.ex|dlna.lab.blueNN.ex
|ws4-01.int.blueNN.ex|ws4-02.lab.blueNN.ex
|ws4-03.lab.blueNN.ex|git.lab.blueNN.ex)
(ssh|ssh.ipv6) OK SSH OK -
(OpenSSH_6.6.1p1|OpenSSH_5.9p1|OpenSSH_6.6.1_hpn1
3v11) (Ubuntu-2ubuntu2|FreeBSD-20140420|Debian-
5ubuntu1|Debian-5ubuntu1.4) (protocol 2.0)
```

Fig. 5. Sample joint clusters detected by LogCluster (for the reasons of privacy, sensitive data have been obfuscated).

V. Conclusion

In this paper, we have described the LogCluster algorithm for mining patterns from event logs. For future work, we plan to explore hierarchical event log clustering techniques. We also plan to implement the LogCluster algorithm in C, and use LogCluster for automated building of user behavior profiles.

Acknowledgment

The authors thank NATO CCD COE for making Locked Shields 2015 event logs available for this research. The authors also thank Mr. Kaido Raiend, Mr. Ants Leitmäe, Mr. Andrus Tamm, Dr. Paul Leis and Mr. Ain Rasva for their support.

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, "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.

[3] Risto Vaarandi, "A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs," in *Proceedings of the 2004 IFIP International Conference on Intelligence in Communication Systems*, LNCS Vol. 3283, Springer, pp. 293-308.

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

[5] 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.

[6] Thomas Reidemeister, Mohammad A. Munawar and Paul A.S. Ward, "Identifying Symptoms of Recurrent Faults in Log Files of Distributed Information Systems," in *Proceedings of the 2010 IEEE/IFIP Network Operations and Management Symposium*, pp. 187-194.

[7] 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.

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

[9] 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.

[10] Adetokunbo Makanju, A. Nur Zincir-Heywood and Evangelos E. Milios, "Clustering Event Logs using Iterative Partitioning," in *Proceedings of the 2009 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1255-1264.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.