Automatic Event Log Abstraction to Support Forensic Investigation

Hudan Studiawan
Murdoch University
Perth, WA 6150, Australia
hudan.studiawan@murdoch.edu.au

Ferdous Sohel
Murdoch University
Perth, WA 6150, Australia
f.sohel@murdoch.edu.au

Christian Payne
Murdoch University
Perth, WA 6150, Australia
c.payne@murdoch.edu.au

ABSTRACT

Abstraction of event logs is the creation of a template that contains the most common words representing all members in a group of event log entries. Abstraction helps the forensic investigators to obtain an overall view of the main events in a log file. Existing log abstraction methods require user input parameters. This manual input is time consuming due to the need to identify the best parameters, especially when a log file is large. We propose an automatic method to facilitate event log abstraction avoiding the need for the user to manually identify suitable parameters. We model event logs as a graph and propose a new graph clustering approach to group log entries. The abstraction is then extracted from each cluster. Experimental results show that the proposed method achieves superior performance compared to existing approaches with an F-measure of 95.35%.

CCS CONCEPTS

• **Applied computing → Investigation techniques**; *System forensics.*

KEYWORDS

event log, log abstraction, graph clustering, log forensics

ACM Reference Format:
Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2020. Automatic Event Log Abstraction to Support Forensic Investigation. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW 2020), February 4–6, 2020, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3373017.3373018

1 INTRODUCTION

The various events of operating systems and other software are recorded in one or more log files. When there is a security incident, the forensic investigators may need to examine these files. As a log file usually contains numerous entries, it takes a lot of time to extract the main activity from the event logs. Therefore, abstraction of event logs is required to identify the most commonly occurring messages in the files. In other words, an abstraction is the set of string patterns that occur most commonly in a particular log file.

ACSW 2020, February 4–6, 2020, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-7697-6/20/02…$15.00
https://doi.org/10.1145/3373017.3373018

Another purpose of abstraction is to parse the event logs because the abstraction will represent every field in a log line [11]. On an enterprise level such as a user's credit account application, event log abstraction is needed for source code profiling and monitoring [14]. The abstraction can also serve as a clustering tool because one abstraction represents a group or cluster of log entries [16]. Furthermore, abstraction can reduce the computation time as one abstraction represents many log lines for anomaly detection and analysis [6, 18].

Event log abstraction is called as system events in [23] and it is extracted from raw logs. The abstraction can be used to detect system failure and problems such as insufficient memory or database errors [13]. The network administrators can utilize an abstraction as a log template. Moreover, abstraction makes the process of checking large log files efficiently [15].

Existing methods such as Simple Log Clustering Tool (SLCT) [27] and LogCluster [28] require one mandatory parameter known as the relative support threshold (*rsupport*). This parameter means that SLCT and LogCluster will find clusters that match at least *rsupport*% of log entries in an input file. However, there are 14 and 26 optional parameters for SLCT and LogCluster, respectively. Iterative Partitioning Log Mining (IPLoM) needs five parameters [16, 17]. LogSig needs the number of clusters as an input [24], but it is not trivial to determine the best number of clusters for a log file. Other techniques, such as Thaler et al. [25, 26], need model training. Some more recent methods also need parameters in order to run. For example, Spell requires a message type threshold [4], while Drain needs a tree depth value, maximum child value, and event log similarity threshold [12].

The aforementioned approaches have two main characteristics. First, they require user inputs. The process of obtaining the optimal input parameters for different cases is difficult. Second, the existing methods need a training step, which takes time while the investigator has to examine the event logs immediately to solve a forensic case. To address these problems, we propose an automatic approach to generate event log abstraction without requiring any user input or model training. The use of more forensic tools with an automatic procedure is also recommended in [22].

Using raw event logs as input, we automatically preprocess them with an automatic log parser [21]. Therefore, the user does not need to define any parsing rules such as using regular expressions. We then group the preprocessed event logs based on the word count because an abstraction that will be extracted from each word in a group of log entries. We then consider the similarity of messages in event logs by refining the previous grouping results with graph-based clustering. The last step of the proposed method extracts the event log abstraction from each cluster.

Contributions. To summarize, the contributions of this paper are as follows.

(1) We propose a novel graph model for logs with a new edge weight namely weighted Hamming similarity.
(2) We propose automatic graph clustering based on Girvan-Newman community detection [8] and a modularity value [19] to obtain the best cluster configurations.
(3) We extract log abstractions from each cluster automatically without the need for user intervention.
(4) The automatic feature of the proposed method still provides the best performance with an F-measure of 95.35%.

This paper is organized as follows. Section 2 gives an overview of related works on event log abstraction. Section 3 defines the problem that is addressed in this paper. Section 4 describes the proposed method. Experiment results are given in Section 5. In Section 6, we provide the conclusion of this research.

2 RELATED WORK AND MOTIVATION

This section reviews the related work on event log abstraction. Subsequently, we demonstrate the advantage of discovering abstractions for log forensic investigation.

2.1 Related Work

We use several terms related to event logs adopted from [22] as follows. An event is an action occurring on a particular device and is recorded in a log entry. An event log is a record of events and usually saved in a log file. Moreover, a log file is a file that records activities from applications or an operating system. Furthermore, a log entry is defined as a single record in a log file.

SLCT was one of the earliest methods used for event log abstraction and involved both frequent-itemset mining and density-based clustering [27]. However, SLCT does not consider the position of the word in each line. Subsequently, the LogCluster method [28] improved SLCT by taking the word position into account. However, it should be noted that SLCT and LogCluster were not intended to be abstraction methods but rather their main aim was to cluster event logs. To represent each cluster, SLCT and LogCluster extract the most frequent line patterns. Subsequently, these patterns become the abstractions of the logs. One mandatory parameter for LogCluster is *rsupport* value. This parameter means that LogCluster will find clusters that match at least *rsupport*% of log entries in an input file. Additionally, LogCluster has many optional parameters, such as *wsize* for filtering out infrequent words from the word frequency estimation process and *csize* for filtering out cluster candidates which match less than *rsupport* log entries.

IPLoM has four phases: grouping by word count, grouping by word position, grouping by bijection search, and discovering abstraction for each group [16, 17]. The last step of IPLoM gives a label to the formed clusters and this label becomes the abstraction. There are five parameters for IPLoM, specifically the file support threshold, partition support threshold, upper bound and lower bound for bijection search, and cluster goodness threshold [17]. LKE (Log Key Extraction) uses hierarchical-based clustering to group the log lines [6]. LKE then joins two clusters if there are any two log entries with an edit distance smaller than an input threshold from the user. Another method known as LogSig uses a technique similar to classic k-means clustering to group the event log messages [24]. Therefore, we need to supply the number of clusters k to obtain the grouping results.

Input log file: auth.log
Output abstractions:
#1 Mar * * nssal * removing removable location: *
#2 Mar 8 * nssal * Invalid user * from *
#3 Mar 8 * nssal * Failed password for * from * port * ssh2
...

Figure 1: An illustration of event log abstraction generated from a log file to assist a forensic investigation.

More latest work on event log abstraction are LogMine [10], Spell [4], and Drain [12]. LogMine groups log entries incrementally before generating an abstraction for each cluster [10]. LogMine is designed to support scalability to process large log files using the map-reduce technique. Spell method creates real-time abstractions as input log entries are created [4]. Spell uses the longest common subsequence (LCS) technique to get abstractions and an input threshold is used to set the appropriate LCS length. Recent research, namely the Drain method, proposes a fixed-depth parse tree to model event logs [12]. The abstraction is generated by traversing the tree with several restrictions from the input parameter, specifically tree depth value, maximum child value, and log similarity threshold.

The aforementioned works on event log abstraction were not explicitly intended to be applied to forensic investigations. However, two papers discuss event log abstraction for forensic analysis using deep learning technique [25, 26]. Thaler et al. [25] find a log signature using a supervised method with the combination of a feed-forward neural network and long short-term memory (LSTM). Afterwards, Thaler et al. [26] upgraded the method by applying an unsupervised approach using autoencoders with LSTM cells.

2.2 Motivating Example

For a given log file, a forensic investigator can acquire general insights about what has occurred in a specific situation from event log abstraction. For example in Figure 1, the input file is an auth.log file containing authentication-related events on a Linux operating system. An activity that needs further examination as shown in Figure 1 is `removing removable location`. This means that there is an unplugged removable disk. This may or may not be a normal activity as an attacker may have physical access to the restricted computer. Other two activities, `failed password` and `invalid user`, are suspicious. Event `invalid user` means that there has been a typical brute-force attack where the attacker or botnet has attempted to use all usernames and passwords from a dictionary. On the other hand, event `failed password` means that an attacker has repeatedly attempted to acquire the password using an automatic tool.

In summary, by discovering log abstractions, we want to provide a general insight from a log file without having to inspect it line by line. For a suspicious abstraction, an investigator can examine into log entries in more depth to obtain further details.

Cluster #1:
Jan 18 09:31:32 victoria dhclient: DHCPACK from 10.0.2.2
Jan 18 10:56:40 victoria dhclient: DHCPACK from 10.0.2.2
Feb 6 13:31:12 victoria dhclient: DHCPACK from 10.0.2.5

Abstraction #1:
* * * victoria dhclient: DHCPACK from *

Cluster #2:
Feb 6 12:56:48 victoria init: Switching to runlevel: 0
Jan 18 17:13:49 victoria init: Switching to runlevel: 6
Feb 6 13:03:53 victoria init: Switching to runlevel: 6

Abstraction #2:
* * * victoria init: Switching to runlevel: *

Figure 2: Examples of event log abstraction that is extraced from each cluster.

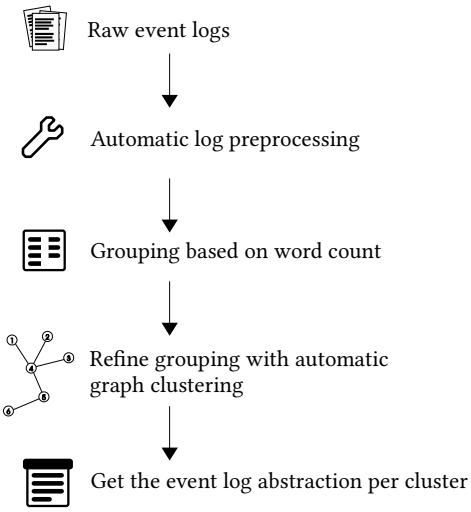

Figure 3: The proposed method for automatic event log abstraction.

3 PROBLEM DESCRIPTION

Abstraction of event logs is a template that contains the most common words representing all members in a group of event log entries. In other words, the aim of event log abstraction is to extract a log template from a set of log entries [4]. An abstraction method distinguishes the constant part and variable part of each log entry [11]. The constant part is a word coming from a system or application logging module. The variable part is the word that changes and comes from a system or application that is running. An example of a variable part is the IP address and port number. An illustration of event log abstraction is presented in Fig. 2. The wildcard or asterisk symbol (*) indicates the variable part in a group of log entries. This symbol is commonly used in event log abstraction literature.

Formally, we denote all raw log entries from a log file as $L = \{l_1, l_2, \ldots, l_{|L|}\}$, where l_i is a particular log entry and $|L|$ is the length of L. Given a set of raw logs L, we need to find event log abstractions $A = \{A_1, A_2, \ldots, A_n\}$. Before finding A, we need to identify n clusters $\{C_1, C_2, \ldots, C_n\}$ from L. An abstraction is extracted from each cluster.

Furthermore, each abstraction A_i is a sequence of words, $A_i = \{t_{i1}, t_{i2}, \ldots, t_{i|A_i|}\}$, where t_i is a i-th word extracted from C_i and $|A_i|$ is the length of A_i. The procedure checks each l_j in each C_i. On the other hand, l_j consists of words t_k, where $1 \leq k \leq |l_j|$. For each $t_k \in l_i$, if t_k has the same word index position for all log entries in C_i, $\forall l_j \in C_i$, then t_k becomes the constant part of abstraction A_i, $t_{ik} = t_k$. Otherwise, t_k will be replaced with wildcard character and acts as the variable part of the abstraction A_i, $t_{ik} = *$.

Several names are used for event log abstraction. Some papers refer to it as the event log signature [25, 26] or message signature [24]. SLCT [27] and LogCluster [28] define abstraction as line patterns while IPLoM [16] uses the term a line format. In the other literature, abstraction is referred to an event log template [15], event log parsing [11], system events [23], or log keys [6]. In this paper, we choose the term “event log abstraction" as used in [13, 14, 17] and define it as a template containing words and wildcard characters representing all members in a group of event log entries.

4 THE PROPOSED METHOD

The proposed method is summarized in Figure 3. First, we have raw event logs as digital evidence. Second, we preprocess the logs automatically without regular expression or other rule-based parsers. Then we group the preprocessed logs based on the word count. To consider the word similarity between messages, we then group the previously processed messages by means of automatic graph-based clustering. The last step extracts the abstraction for each cluster found. Each step is described in a separate subsection below.

4.1 Event Log Preprocessing

The preprocessing stage involves two main steps: log parsing and creating unique messages from given raw logs. Event log parsing is a procedure to split and label each field in a log entry. We parse the log files using nerlogparser [21], a log parsing tool which is based on named entity recognition (NER). In the area of natural language processing, NER is a process used to extract named entities from text. In the context of log files, nerlogparser defines named entities as words or phrases containing common fields in a log entry such as timestamp, host name, or service name. Therefore, identifying named entities is equivalent to determining each field in a log entry. The nerlogparser uses bidirectional long short-term memory networks to perform NER [21].

Automatic event log parsing with nerlogparser is illustrated in Fig. 4. The main advantage of nerlogparser is that it supports fully automatic parsing because it provides a pre-trained model. Therefore, the investigators do not need to define any rules or regular expressions. It is also a generic tool as there is only one model file to parse various types of log files. Note that event log parsing discussed in this paper is different from [11] as they refer to event log parsing as event log abstraction.

In the second preprocessing step, we extract unique messages from the log entries. A unique message is a message from the parsing step that different to the others. Each message from a particular

Input:
Jan 18 09:31:32 victoria dhclient: DHCPACK from 10.0.2.2

Process: automatic parsing with the nerlogparser tool

Output:
Timestamp : Jan 18 09:31:32
Host name : victoria
Service : dhclient:
Message : DHCPACK from 10.0.2.2

Figure 4: An example of event log parsing for a log entry.

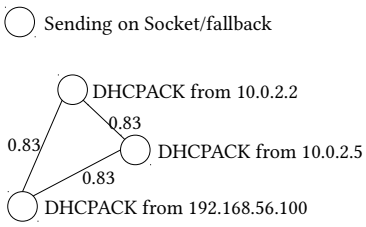

Figure 5: Illustration of the proposed graph model where a unique message is modeled as a vertex and weighted Hamming similarity as an edge weight.

log entry will be attached to a unique message. As we have described in Section 3, all raw logs are denoted as L. Formally, we define L' as the parsed logs from L. We then extract unique messages from L' and these are denoted by $M = \{m_1, m_2, \ldots, m_{|M|}\}$ where $|M|$ is the number of unique messages. A particular log entry l_i will be attached to one of these unique messages.

Note that we focus on the abstraction of the message for each group of log entries. Therefore at this stage, the proposed abstraction does not consider other fields such as timestamps or host name. These are considered in the final stage when extracting the abstraction.

4.2 Grouping based on Word Count

The next step in event log abstraction is to group the extracted unique messages based on word count. We split the discovered unique messages based on space character then count the word length. This assumption is based on the fact that an abstraction is a representation of a group of log entries having the same length. The same message templates are likely to have the same word count. This assumption is based on the IPLoM method [12, 17]. An abstraction is extracted from the always-occurring word in a group of unique messages with the same length.

At this stage, it is trivial to extract an abstraction as the log entries have been grouped based on their length. However, in a real-life log file, the log entries are very diverse in terms of number of words or vocabularies, so we need to refine these groups based on the string similarity. We discuss this issue in the next section.

4.3 Graph Model for Log Messages

Before performing automatic graph-based clustering, we describe the proposed graph model for log messages as illustrated in Figure 5. For every discovered count-based word group, we construct an undirected and edge-weighted graph, defined as $G = (V, E)$. Every unique message $m \in M$ is now represented as a vertex v in the set V. E is a set of weighted edges. The edge e_{ij} from v_i to v_j is created when the weighted Hamming similarity between two messages in v_i and v_j is greater than zero. Note that $i \geq 1, j \leq |V|$, and $i \neq j$. We denote $w(e_{ij})$ as the edge weight. The edge weight $w(e_{ij})$ is a value of the weighted Hamming similarity between the two vertices in a range of $[0, 1]$. As shown in Figure 5, a log message that shares no similarity with other vertices will be naturally separated from the main connected component of the graph. This property is the main advantage of the proposed graph model.

We maintain a vertex property $p(v)$ which comprises the members in a vertex v. We can determine this value by maintaining a list that contains the index i from $l_i \in L$. The list represents which individual row of a log entry l refer to a unique message m. In addition, this list will make the proposed method easier to traverse back to the original raw logs when retrieving the clustering results.

The edge weight $w(e_{ij})$ in the graph G is the weighted Hamming similarity H' between two unique messages represented in two vertices, respectively. Before calculating H', we define a basic identity function, $\text{sim}(t_i, t_j)$, for Hamming similarity between two words t_i and t_j:

$$\text{sim}(t_i, t_j) = \begin{cases} 1, & \text{if } t_i = t_j \\ 0, & \text{if } t_i \neq t_j \end{cases} \tag{1}$$

Subsequently, the Hamming similarity H between two log messages (m_x, m_y) is defined as:

$$H(m_x, m_y) = \sum_{i=1}^{|m_x|} \text{sim}(t_{xi}, t_{yi}), t_{xi} \in m_x, t_{yi} \in m_y \tag{2}$$

where $|m_x|$ is the length of log message m_x. Note that m_x and m_y already have the same length as discussed in Section 4.2. The reason for choosing Hamming similarity is that the edge weight should take into account the position of a word in a log entry because the abstraction will be generated based on word order. Moreover, we modify the Hamming similarity to be weighted Hamming similarity because the positions of words are important for calculating similarity and we want to give more weight to similarity between string messages.

As suggested in [6], the constant parts of a log message tend to be placed at the beginning of a log entry. Therefore, words at the beginning of a log entry are more likely to be part of event log abstraction than words at the end of a log entry. Formally, we denote weighted Hamming similarity H' between two log messages (m_x, m_y) as follows:

$$H'(m_x, m_y) = \frac{\sum_{i=1}^{|m_x|} (|m_x| + 1 - i)\ \text{sim}(t_{xi}, t_{yi})}{\sum_{i=1}^{|m_x|} i} \tag{3}$$

where $(|m_x| + 1 - i)$ is one-based index and it is reversed to give high similarity for the words occuring at the beginning of a log entry. Similar to Eq. 2, we also define $t_{xi} \in m_x$ and $t_{yi} \in m_y$ in Eq. 3.

Algorithm 1: The proposed automatic graph clustering.

```
Input: graph
Output: best_cluster
Procedure GetClusters(graph):
    best_cluster ← dict()
    max_modularity ← -1
    clusters ← GirvanNewman(graph)
    for cluster in clusters do
        modularity ← get_modularity(cluster, graph)
        if max_modularity < modularity then
            max_modularity ← modularity
            best_cluster ← cluster
        end
    end
End Procedure
```

Algorithm 2: Building micro-clusters automatically.

```
Input: graph
Output: final_clusters
Procedure AutomaticClustering(graph):
    clusters ← GetClusters(graph)
    final_clusters ← dict()
    for cluster in clusters do
        if len(cluster) ≤ 3 then    // micro-cluster size
            final_clusters.update(cluster)
        else
            AutomaticClustering(cluster) // recursion
        end
    end
End Procedure
```

4.4 Grouping with Automatic Graph Clustering

The previous grouping step considers only the word count. Therefore, the next step is to cluster the log entries based on the similarity of the messages. To provide an automatic clustering approach with the proposed graph model, we use Girvan-Newman community detection [8] and the modularity value [19] as the basis for log clustering. In this paper, a community refers to a cluster in a particular graph. Furthermore, a cluster can be viewed as a subgraph in a particular graph. Note that we represent each discovered count-based word group as a graph as discussed in the previous section.

Get clusters. The proposed procedure for automatic clustering is shown in Algorithm 1. In the GetClusters procedure, it first initializes best_cluster as a dictionary data structure and max_modularity to -1 where modularity lies between -1 and 1. We use the NetworkX graph library [9] to get clusters using the Girvan-Newman method. The GirvanNewman function is a Python generator, meaning that it generates a new cluster configuration when called in each loop.

To retrieve the clusters, the Girvan-Newman method identifies edges in a graph that are mostly between other pairs of vertices. It uses an edge betweenness value which is the number of shortest paths between pairs of vertices [8]. It then progressively removes the edge with highest edge betweenness value. We modify the Girvan-Newman method by changing the rule to cluster the vertices. The original Girvan-Newman uses edge betweenness as the criteria to group vertices. Instead of using edge betweenness, we use the lightest edge $w_{\min}$ to remove edges when constructing the cluster as defined below:

$$w_{\min} = \min_{\forall w \in w(E)} (w) \tag{4}$$

where $w(E)$ is all edge weights in a processed graph G. The reason for this modification is that in our case, the light edge weight means that the two messages that are represented in two vertices are not similar as the weighted Hamming similarity value is low. By removing less similar edge weight, the clustering procedure will yield clusters that have the most similar messages.

For each cluster configuration, we then calculate the modularity value. Formally, modularity for weighted graph Q is denoted as follows [19]:

$$Q = \frac{1}{2q} \sum_{i,j} \left[w(e_{ij}) - \frac{z_i z_j}{2q} \right] \delta(C_i, C_j) \tag{5}$$

where $w(e_{i,j})$ denotes the edge weight between vertices v_i and v_j and z_i is the sum of weighted degree of v_i where $z_i = \sum_j w(e_{ij})$. Moreover, C_i is the cluster where vertex v_i is belong to, the $\delta(C_i, C_j)$ function is 1 if $C_i = C_j$ and 0 otherwise, and $q = \frac{1}{2} \sum_{i,j} w(e_{ij})$. The best cluster configuration is determined by the highest modularity value. Finally, the procedure returns the best cluster configuration after certain iterations.

Build micro-clusters. The main algorithm for event log clustering is shown in the AutomaticClustering procedure from Algorithm 2. This procedure finds micro-clusters from the given graph. The "automatic" feature here means that the forensic investigator does not need to input any parameters. The initial valid clusters are connected components of the generated graph. The components are a set of subgraphs such that there are no edges between them. A component that has only one vertex builds its own cluster. In addition, we process the subgraph component that has two or more vertices.

Furthermore, the AutomaticClustering procedure calls GetClusters to obtain all clusters from the input graph. For each cluster, the procedure checks its vertices members. If it has more than three vertices, the procedure will run AutomaticClustering recursively. It will stop the recursion when the size of the cluster is three or less vertices. Three vertices can form a simple subgraph. On the other hand, too many vertices can make a subgraph too complex as it can capture too much of variations in the messages. Therefore, the abstraction may not be meaningful as there will be too many wildcard symbols. Consequently, we use three vertices as the size of micro-clusters.

The aim of the recursion is to build micro-clusters with the size of three or less vertices because the entries in a log file have various messages. The micro-cluster will enable the grouping of diverse messages. The micro-clusters that are not similar to others stay as they are in the abstraction extraction step. In a log file with diverse log messages, many messages do not bear any similarity to other messages so they will construct their own cluster.

At this stage, the clusters now have a new composition with better grouping since we consider both the word count and the string similarity. For each cluster, we remember each log entry line number as denoted in vertex property $p(v)$, so we can traverse back and get the log entries for each cluster.

Algorithm 3: The proposed method for automatic event log abstraction.

```
Input: log_file
Output: final_abstractions
Procedure AutomaticAbstraction(log_file):
    parsed_logs ← preprocess(log_file)
    length_groups ← get_length_groups(parsed_logs)
    abstractions ← dict()
    for length_group in length_groups do
        graph ← create_graph(length_group)
        clusters ← AutomaticClustering(graph)
        for cluster in clusters do
            abstraction ← get_abstraction(cluster)
            abstractions.update(abstraction)
        end
    end
    merged_abstractions ← merge(abstractions)
    final_abstractions ← get_final(merged_abstractions)
End Procedure
```

4.5 Extraction of Event Log Abstraction

The next step is to automatically extract an abstraction for each micro-cluster found. We need to find the same word with the same index position, which then becomes the *constant part* of the abstraction. If a particular word differs from other words in the same index position, this becomes the *variable part* of the abstraction. Therefore, these words will be replaced by an asterisk symbol. The overall process of the proposed method including the extraction of abstraction discussed in this section is summarized in Algorithm 3. The algorithm contains cross-references to the relevant sections of this paper to assist the reader in reviewing the explanation of each step.

To obtain the word indices for the asterisk symbol, we first determine all indices for all words, then find all indices for all the constant words in all log entries in a particular cluster. The word indices for the asterisk sign I_{ast} are acquired from all indices, I, subtracted by the constant word indices, I'. Thus, $I_{ast} = I - I'$ and we replace the word that has an index position in I_{ast} with an asterisk character.

Merging abstractions. An abstraction is extracted from a micro-cluster and there is a possibility that this abstraction will be very similar to others. Therefore in the next step, we apply a merging procedure in order to obtain more compact abstractions. First, we find pair combinations (A_i, A_j) from all abstractions to be compared. Two abstractions A_i and A_j will continue to be checked for merging if there is a weighted Hamming similarity between them ($H' > 0$).

We consider that an abstraction A_i as a sequence of words, $A_i = \{t_{i1}, t_{i2}, \ldots, t_{i|A_i|}\}$ and $A_j = \{t_{j1}, t_{j2}, \ldots, t_{j|A_j|}\}$. For each t_{ix} and t_{jx} where $x = 1, 2, \ldots, |A_i|$, we check the possibility of merging A_i and A_j based on the following rules:

(1) If $t_{ix} = t_{jx}$ then merge = true, and
(2) If $t_{ix} \neq t_{jx} \wedge (t_{ix} = * \vee t_{jx} = *)$ then merge = true.

If t_{ix} and t_{jx} do not follow these rules, then we will not merge two abstractions A_i and A_j. For the comparison checking of t_{ix} and t_{jx}, we do not consider symbol characters, numbers, and punctuation to get more intuitive abstraction merging results. For example, the IP address and port number are considered as a variable part of the abstraction.

Final abstractions. In all previous steps, we consider only the message field in a log entry. In the final abstraction extraction step, we consider all other fields such as timestamp, host name, and service name. These fields are also important in assisting a forensic investigation.

For each abstraction A_i, we check each parsed log entry that is saved in the vertex property $p(v)$. We retrieve all fields excluding the message field. For all fields from l_j (excluding the message field) where l_j is a log entry assigned to an abstraction A_i, we extract the constant and variable part and it is denoted as B_i The final abstraction result F_i is a concatenation of B_i and A_i. Formally, $F_i = \text{concat}(B_i, A_i)$.

Table 1: List of Public Forensic Datasets

Dataset	# Files	# Lines	# Abs
Digital Corpora (Casper) [7]	15	11,086	3,422
DFRWS 2009 (Jhuisi) [2]	25	11,737	3,488
DFRWS 2009 (Nssal) [2]	40	107,093	5,573
DFRWS 2016 (DF16) [3]	1	3,304	102
Honeynet Challenge 7 (Honey) [1]	12	8,712	2,039

5 EXPERIMENTAL RESULTS

In this section, we describe the experiments used to test the performance of the proposed method. We discuss event log datasets and explain the experimental settings. We then compare the performance of the proposed method with existing approaches when applied to various datasets.

5.1 Public Digital Forensic Datasets

This experiment uses five public datasets related to event log forensics as summarized in Table 1 where # Abs means the number of abstractions for all log files in a particular dataset. These datasets represent various scenarios related to a forensic investigation. Open datasets were selected to aid reproduction of these experimental results and to allow future comparisons with subsequent methods. The first dataset is a disk image namely nps-2009-casper-rw from Digital Corpora [7]. It is an ext3 file system dump from a bootable USB. The user of this disk image browses several US Government websites. This dataset provides event logs, especially various logs from a Linux system.

The Digital Forensic Research Workshop (DFRWS) provides an annual forensic challenge which is associated with event log forensics. In the DFRWS Forensic Challenge 2009 [2], the researcher has to examine various log files to trace an attacker who illegally transferred secret data. There were two hosts involved namely "jhuisi" and "nssal". These two hosts were Sony PlayStation 3 (PS3) devices that running a Linux operating system. The DFRWS challenge in 2016 [3] provided logs from a Software-Defined Network (SDN) environment. We have carved and parsed the log files from an SDN OpenVSwitch (OVS) switch for this experiment.

Table 2: Parameter Settings for Experiments

Method	Parameter settings
Proposed method	none
IPLoM [17]	file support threshold = 0
	partition support threshold = 0
	upper bound = 0.9
	lower bound = 0.25
	cluster goodness threshold = 0.175
LogSig [24]	number of cluster = ground truth cluster
Drain [12]	tree depth = 4
	similarity threshold = 0.4
	maximum child = 100
LogMine [10]	levels of pattern hierarchy = 2
	maximum distance = 0.001
	distance weight = 1
Spell [4]	message type threshold = 0.5

The Honeynet Forensic Challenge 7 2011 [1] also provides a disk image of a compromised Linux server and a forensic analysis is required in order to understand what has occurred. From the provided hard disk dump, we extracted the log files and parsed them. For all datasets except DFRWS 2016, we recovered the directory `/var/log/` from the forensic disk images. In this directory, we retrieved some common log files such as authentication logs, kernel logs, and system logs.

5.2 Experimental Settings

We run the experiments on a computer with an Intel Core i7 processor, 4 GB RAM, with Linux Ubuntu 16.04 LTS. The proposed method is implemented with Python 3 programming language. We use the NetworkX library [9] to manipulate the graph and perform graph clustering based on the Girvan-Newman method.

To measure the performance of the proposed method, we manually build the ground truth for event log abstractions. To build the ground truth, we created a dictionary of essential words and phrases from operating system log files. For example, "accepted password", "connection closed", and "failed password". For each word and phrase in the dictionary, we checked if a log entry contains them. We grouped all log entries based on these words and phrases. Note that each word or phrase created one group. The next step is to check the word length in each group. We grouped all log entries based on the word length. From this final group, we checked for the constant and variable parts and then extracted the abstraction. The abstractions from all of the final groups become the ground truth.

We use the F-measure as the evaluation metric. We follow the definition of F-measure in [11, 12] as these papers also related to event log abstractions. The F-measure is related to several values specifically true positive (TP), false positive (FP), and false negative (FN). A true positive is when a method assigns two log messages with the same abstractions to the same abstractions. A false positive is calculated when a method assigns two log messages with the different abstractions to the same abstractions. A false negative is when a method assigns two log messages with the same abstractions to the different abstractions.

Before calculating F-measure, we need to find precision and recall as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

Furthermore, the F-measure is defined as:

$$\text{F-measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{8}$$

For each dataset, we calculate the F-measure for each file and then obtain the mean F-measure for all log files.

5.3 Comparison with Existing Methods

We compare the proposed method with five other methods, specifically IPLoM [17], LogSig [24], Drain [12], LogMine [10], and Spell [4]. All source codes, datasets, and abstraction ground truth are publicly available in a GitHub repository[1]. We use the source code of existing methods from [11, 12]. The proposed method, IPLoM, LogSig, LogMine, and Drain are implemented in Python 3. However, Spell is implemented with Python 2. Therefore, Spell is located in a different repository[2].

Before running the experiments, we define the parameters used by existing methods as shown in Table 2. Note that the proposed method is parameter-less, so we do not need to specify any inputs. The parameter for LogSig is the number of clusters. Since it is not an easy task to get the most appropriate number of clusters in various log files, we directly get this value from ground truth. The choice of the parameters for all methods is based on the original papers and we also refer to the source code implementation in [11, 12]. Based on previous works [11, 12], these parameter values have been demonstrated to produce the best performance.

To run an objective and fair comparison, we use the nerlogparser tool [21] in preprocessing step for all methods. By using nerlogparser, we do not need to define any regular expressions to parse the log files from all datasets. We also give the same process when discovering abstractions. For all methods, we first extract an abstraction from the messages and then concatenate them with an abstraction from other fields from the log entries in the same cluster.

Table 3 shows the comparison of the proposed method with five other methods. IPLoM uses heuristic rules that are constructed based on the characteristics of log messages specifically grouping by word position and bijection search. The bijective relationship in a group of log entries can accurately capture the most frequently occurring words. Therefore, IPLoM demonstrates good performance for four datasets. The best IPLoM performance is found for the Casper dataset with a 93.72% F-measure while the proposed method achieves 94.94%.

With the number of clusters supplied from ground truth, LogSig shows a fair performance with the best F-measure of 85.50% achieved for the Honey dataset. LogSig cannot cluster log messages precisely although the number of clusters is set to be the same value as

[1] https://github.com/studiawan/pylogabstract
[2] https://github.com/studiawan/spell

Table 3: F-measure Value Comparison (in %) of The Proposed Method and Five Other Methods

Method	Datasets				
	Casper	Jhuisi	Nssal	DF16	Honey
IPLoM [17]	93.72	87.58	88.15	10.90	89.31
LogSig [24]	77.16	80.33	79.71	12.00	85.50
Drain [12]	90.01	87.49	89.05	17.50	94.12
LogMine [10]	72.06	74.17	66.17	14.60	77.49
Spell [4]	82.00	82.02	79.00	10.80	83.70
Proposed method	**94.94**	**96.32**	**92.11**	**97.10**	**96.26**

ground truth. The reason is that the clustering is performed based on a local search algorithm. As explained in the LogSig paper [24], the use of the search-based optimization algorithm can lead to local optima. For instance, message `session closed for user root` and `session closed for user ubuntu` are assigned to two different clusters because in the grouping search process they are already trapped in local optima.

Drain performs well on four of the five datasets. The reason is that Drain considers the first few words in a log entry as contributing most significantly to its abstraction. These few words are also used to construct a fixed-depth tree. To avoid over-clustering, Drain does not employ words that have no digits when traversing the tree data structure. For the Honey dataset, Drain can achieve an F-measure value (94.12%) close to that of the proposed method.

The overall lowest F-measure is shown by the LogMine method. LogMine performs over-clustering for all datasets because the clustering process is conducted incrementally for each log entry. If a log entry similarity with an existing cluster representation is less than the given threshold, it will be grouped with that particular cluster. In addition, LogMine's clustering procedure is also sensitive to parameter settings [10]. Although LogMine has a log-merging procedure, it was unable to merge accurately in some cases. For example with the Casper dataset, the message `Registered protocol family 20` and `Registered protocol family 1` should be assigned in one cluster, but LogMine did not merge them.

Spell also shows a fair performance compared to the proposed method. Spell mainly employs the longest common subsequence (LCS) technique to obtain the abstractions. Unfortunately, LCS cannot capture any potential abstraction that has separate substrings. LCS will capture the longest substring, but it apparently fails to capture the second longest substring. This situation can lead to the under-clustering of log entries. The best F-measure for Spell is 83.70% for the Honey dataset which is quite below that of the proposed method with 96.26%.

DFRWS 2016 is the most challenging dataset overall. In this dataset, log entries are dominated by a message containing term `br0<->tcp:192.168.1.1:6633:` with 2,908 of total 3,304 entries. There is another very similar word from this dominating word, specifically `WARN br0<->ssl:192.168.1.1:6633:`. The string similarity technique in existing methods cannot separate these two similar words which are located at the beginning of log messages. Therefore, these log entries are assigned in a same cluster. As a consequence, the existing methods produce low F-measures between 10.80% to 17.50%. On the other hand, we develop weighted Hamming similarity which prioritizes the words occurring at the beginning of a log entry. As a result, we obtain 97.10% F-measure for DFRWS 2016 dataset.

In general, the proposed method achieves the best F-measure with an overall mean score of 95.35%. There are several reasons for this superior performance. First, we employ a new distance measure named weighted Hamming similarity to calculate the similarity between two words in a group of log entries. This similarity not only considers the order, but also gives higher weight to the words at the beginning of a log entry. The proposed graph clustering works by progressively removing the lightest weighted Hamming similarity. Intuitively, this novel approach can produce the best cluster configuration. Therefore, the proposed method achieves the best performance in terms of discovering abstractions as supported by the F-measure score.

5.4 Over-clustering and Under-clustering

The most important procedure in discovering event log abstractions is the clustering step. If the clustering is performed well, then good abstractions will be produced. Based on our intensive experiments on five datasets, the existing methods produce over-clustering results. Therefore, they generate abstractions that are too detailed.

The takeaway message is that we need to get the best cluster composition from event logs. This enables us to obtain the most descriptive abstractions as each abstraction represents one cluster. The intuitive and descriptive event log abstractions are important as the system administrators or the forensic investigators will be able to analyze the more detailed problems found in the log messages based on information from the abstractions.

However, over-clustering and under-clustering problem are not applicable to the LogSig method. The reason is that LogSig has a problem with the local search algorithm and not the number of clusters. Note that the number of clusters for LogSig is configured based on the ground truth for the experiments in this paper.

6 CONCLUSION AND FUTURE WORK

This paper proposes an automatic method of event log abstraction. Being automatic, there is no need for a forensic investigator to supply any parameters. This is a significant improvement as the existing approaches either need many user inputs or need a model training. The proposed method achieves the highest mean F-measure of 95.35% across all datasets.

Future work will focus on integrating the automatic event log abstraction with event reconstruction [5] and anomaly detection [20] for forensic purposes. This integration will complete the reconstruction process with a general insight of the whole logs. Therefore, the investigator is able to more easily identify any malicious activities in the event log files.

ACKNOWLEDGMENTS

This work is supported by the Indonesia Lecturer Scholarship (BUDI) from Indonesia Endowment Fund for Education (LPDP), Ministry of Finance, Republic of Indonesia.

REFERENCES

[1] Guillaume Arcas, Hugo Gonzales, and Julia Cheng. 2011. Challenge 7 of the Honeynet Project Forensic Challenge 2011 - Forensic analysis of a compromised server. Retrieved August 21, 2017 from https://old.honeynet.org/challenges/2011_7_compromised_server

[2] Eoghan Casey and Golden G. Richard III. 2009. DFRWS Forensic Challenge 2009. Retrieved August 21, 2017 from http://old.dfrws.org/2009/challenge/index.shtml

[3] CMAND. 2016. DFRWS Forensic Challenge 2016 from Center for Measurement and Analysis of Network Data (CMAND). Retrieved August 21, 2017 from http://old.dfrws.org/2016/challenge/index.shtml

[4] Min Du and Feifei Li. 2016. Spell: Streaming parsing of system event logs. In *Proceedings of the 16th IEEE International Conference on Data Mining*. 859–864.

[5] Xiaoyu Du, Paul Ledwith, and Mark Scanlon. 2018. Deduplicated disk image evidence acquisition and forensically-sound reconstruction. In *Proceedings of the 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. 1674–1679.

[6] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 9th IEEE International Conference on Data Mining*. 149–158.

[7] Simson Garfinkel. 2009. nps-2009-casper-rw: An ext3 file system from a bootable USB. Retrieved August 21, 2017 from http://downloads.digitalcorpora.org/corpora/drives/nps-2009-casper-rw/

[8] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826.

[9] Aric Hagberg, Daniel Schult, and Pieter Swart. 2008. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*. 11–15.

[10] Hossein Hamooni, Biplob Debnath, Jianwu Xu, Hui Zhang, Guofei Jiang, and Abdullah Mueen. 2016. LogMine: Fast pattern recognition for log analytics. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 1573–1582.

[11] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. 2016. An evaluation study on log parsing and its use in log mining. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[12] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. 2017. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of the IEEE International Conference on Web Services*. 33–40.

[13] Liang Huang, Xiaodi Ke, Kenny Wong, and Serge Mankovskii. 2010. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. 313–326.

[14] Zhen Ming Jiang, Ahmed E. Hassan, Parminder Flora, and Gilbert Hamann. 2008. Abstracting execution logs to execution events for enterprise applications. In *Proceedings of the 8th on Quality Software*. 181–186.

[15] Satoru Kobayashi, Kensuke Fukuda, and Hiroshi Esaki. 2014. Towards an NLP-based log template generation algorithm for system log analysis. In *Proceedings of The 9th International Conference on Future Internet Technologies*. 11:1–11:4.

[16] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2009. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1255–1264.

[17] Adetokunbo Makanju, A. Nur Zincir-Heywood, and Evangelos E. Milios. 2012. A lightweight algorithm for message type extraction in system application logs. *IEEE Transactions on Knowledge and Data Engineering* 24, 11 (2012), 1921–1936.

[18] Meiyappan Nagappan and Mladen A. Vouk. 2010. Abstracting log lines to log event types for mining software system logs. In *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*. 114–117.

[19] M. E. J. Newman. 2004. Analysis of weighted networks. *Physical Review E* 70, 5 (2004), 056131.

[20] Hudan Studiawan, Christian Payne, and Ferdous Sohel. 2017. Graph clustering and anomaly detection of access control log for forensic purposes. *Digital Investigation* 21 (2017), 76–87.

[21] Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2018. Automatic log parser to support forensic analysis. In *Proceedings of the 16th Australian Digital Forensics Conference*. 1–10.

[22] Hudan Studiawan, Ferdous Sohel, and Christian Payne. 2019. A survey on forensic investigation of operating system logs. *Digital Investigation* 29 (2019), 1 – 20.

[23] Liang Tang and Tao Li. 2010. LogTree: A framework for generating system events from raw textual logs. In *Proceedings of the 10th IEEE International Conference on Data Mining*. 491–500.

[24] Liang Tang, Tao Li, and Chang-Shing Perng. 2011. LogSig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. 785–794.

[25] Stefan Thaler, Vlado Menkonvski, and Milan Petković. 2017. Towards a neural language model for signature extraction from forensic logs. In *Proceedings of the 5th International Symposium on Digital Forensic and Security*. 1–6.

[26] Stefan Thaler, Vlado Menkovski, and Milan Petković. 2017. Towards unsupervised signature extraction of forensic logs. In *Proceedings of the 26th Benelux Conference on Machine Learning*. 154–159.

[27] Risto Vaarandi. 2003. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the IEEE Workshop on IP Operations and Management*. 119–126.

[28] Risto Vaarandi and Mauno Pihelgas. 2015. LogCluster - a data clustering and pattern mining algorithm for event logs. In *Proceedings of the 11th International Conference on Network and Service Management*. 1–7.