Information Systems 60 (2016) 13–33

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Complex log file synthesis for rapid sandbox-benchmarking of security- and computer network analysis tools

Markus Wurzenberger [a,*], Florian Skopik [a], Giuseppe Settanni [a], Wolfgang Scherrer [b]

[a] Austrian Institute of Technology, Digital Safety and Security Department, Donau-City-Strasse 1, 1220 Vienna, Austria
[b] Vienna University of Technology, Institute of Statistics and Mathematical Methods in Economics, Wiedner Hauptstrasse 8, 1040 Vienna, Austria

ARTICLE INFO

Article history:
Received 18 December 2015
Accepted 17 February 2016
Recommended by: D. Shasha
Available online 26 February 2016

Keywords:
Log line clustering
Markov chains
Log file analysis
Log data modeling
IDS deployment optimization

ABSTRACT

Today Information and Communications Technology (ICT) networks are a dominating component of our daily life. Centralized logging allows keeping track of events occurring in ICT networks. Therefore a central log store is essential for timely detection of problems such as service quality degradations, performance issues or especially security-relevant cyber attacks. There exist various software tools such as security information and event management (SIEM) systems, log analysis tools and anomaly detection systems, which exploit log data to achieve this. While there are many products on the market, based on different approaches, the identification of the most efficient solution for a specific infrastructure, and the optimal configuration is still an unsolved problem. Today's general test environments do not sufficiently account for the specific properties of individual infrastructure setups. Thus, tests in these environments are usually not representative. However, testing on the real running productive systems exposes the network infrastructure to dangerous or unstable situations. The solution to this dilemma is the design and implementation of a highly realistic test environment, i.e. sandbox solution, that follows a different – novel – approach. The idea is to generate realistic network event sequence (NES) data that reflects the actual system behavior and which is then used to challenge network analysis software tools with varying configurations safely and realistically offline. In this paper we define a model, based on log line clustering and Markov chain simulation to create this synthetic log data. The presented model requires only a small set of real network data as an input to understand the complex real system behavior. Based on the input's characteristics highly realistic customer specified NES data is generated. To prove the applicability of the concept developed in this work, we conclude the paper with an illustrative example of evaluation and test of an existing anomaly detection system by using generated NES data.

1. Introduction

Centralized logging is becoming more and more the key to efficient operations of large-scale interconnected information infrastructures. A central log store is an invaluable source to discover problems in time, such as

* Corresponding author.
E-mail addresses: markus.wurzenberger@ait.ac.at (M. Wurzenberger), florian.skopik@ait.ac.at (F. Skopik), giuseppe.settanni@ait.ac.at (G. Settanni), wolfgang.scherrer@tuwien.ac.at (W. Scherrer).

http://dx.doi.org/10.1016/j.is.2016.02.006
0306-4379/

service quality degradations, performance issues or security-relevant cyber attacks – just to name a few. A wide variety of analysis solutions exist that promise to detect and flag system issues reflected in log files with only little or no human involvement at all. Big data analysis is the new means to discover unprecedented information in large volumes of data. Log file analysis is just one application area.

However, despite the fact that much progress has been made in the application of big data analysis for investigating logging data, there seems to be one key element missing that is required to facilitate the wide adoption of these technique. There is no appropriate solution available to pinpoint the specific requirements on an analysis solution given a certain infrastructure. The underlying problem is that due to the high degree of interconnectedness of distributed systems as well as their application- and domain-specific customization and configuration, there are hardly two networked systems that work exactly the same way. Therefore, we argue that each and every system is unique, either as a result of its configuration, its application domain and/or the way it is utilized.

As a consequence, it is hard for system operators to decide which solution (and configuration) is the best fitting one for their specific infrastructure and their specific purposes. Today, there are only unsatisfying options for said system operators to decide on a logging- and log analysis solution. They can either infer some conclusions (i) from quite general evaluations in testbeds, or (ii) test with somewhat simplified data only which mostly do not reflect the real-world properties sufficiently; or (iii) run tests on their productive systems – which enables the most realistic evaluation of a logging system and its configuration, however at the same time might expose the infrastructure to dangerous or unstable situations. The latter is especially true in case automatic decisions are derived based on the output of a log analysis solution, such as the reaction of an intrusion prevention system due to discovered security violations, or network performance optimizations due to identified resource allocation problems.

Eventually, a novel approach is required which allows an offline evaluation of newly deployed logging- and log analysis solutions and at the same time stresses this system with the most realistic input data possible. Additionally, all this must be done in a cost-effective manner to guarantee high adoption by system integrators and operators. Thus, in this paper we propose a three step solution: first, small samples of real log data are extracted from an already running system. The sample must be long enough to describe normal system behavior with the usual complexity, however it can be short enough to be manually screened for privacy-relevant data, such as usernames and internal urls. In a second step, this data is being analyzed by the means of log line clustering (to identify similar events) and correlation (to identify common sequences of events). The results of this analysis phase are captured as Markov chains. In the third and last step, the captured model is input a large-volume log data generation process with configurable complexity and variability – eventually the important input to test and evaluate logging and log analysis solutions offline without influencing the productive system where the initial data is coming from.

The contributions of this paper – basically the delivery of the most important building blocks of the described system – are the following:

- *Log Data Analysis Approach*: We introduce and describe in detail a novel approach to analyze real log data and capture its unique characteristics. In particular, our approach makes use of log line clustering to discover and describe common events reflected by log lines, and Markov chains to model the correlation and interdependencies of those.
- *Log Data Generation*: Once we got an understanding about the structure and properties of short sequences of log data from a real system, we apply a new approach to generate large volumes of synthetic log data that follow precisely the properties of the analyzed log data before. The advantage here, compared to a simple record & replay-system is that the complexity of the output data can be controlled during the generation and therefore, data to evaluate different kinds of systems can be produced. Additionally, this approach enables us to introduce variations into the produced set (such as time stamps, IP addresses, and system names) – similar to the real world.
- *Evaluation of the Log Data Analysis and Generation Approach*: We show detailed evaluation results to underpin the high quality of the log data generated with our approach. For this purpose we also define novel metrics.
- *Illustrative Application of the Log Data Analysis and Generation Approach*: Finally we highlight the application of the proposed approach in the security domain with focus on anomaly detection/Intrusion Detection Systems (IDS). Here, we first demonstrate the capability of our approach to evaluate such security solutions. Hence, we present an illustratively and exemplarily evaluation of an anomaly detection system which can be part of a security information and event management (SIEM) system and also be used as an IDS. Furthermore we also indicate, how attacks can be simulated and therefore the detection capability can be tested. Moreover we define a step-by-step enrollment process for IDSs based on actual standards from the International Organization for Standardization (ISO) and the National Institute of Standards and Technology (NIST). Based on this deployment process we then argue which steps can be simplified and optimized by applying our approach and highlight its benefits.

The remainder of the paper is structured as follows: Section 2 defines the proposed model for generating synthetic log data. Afterwards Section 3 evaluates and discusses the previously defined model. In Section 4 an illustrative example of application for the generated log data is presented. Further Section 5 defines a step-by-step enrollment process for IDSs and demonstrates the application areas of the proposed approach. Section 6 summarizes related work before Section 7 concludes the paper and provides an outlook on future work.

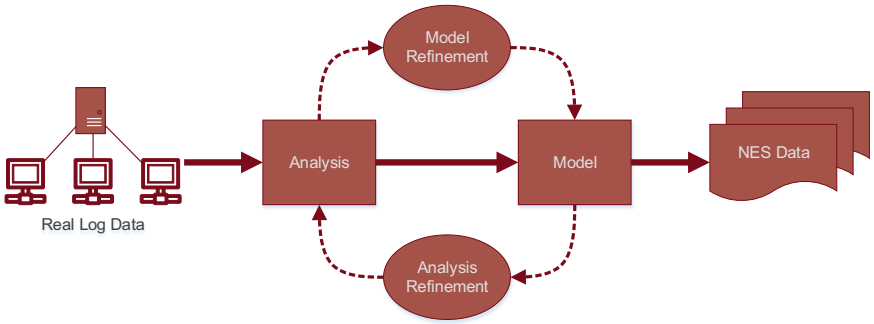

Fig. 1. The concept of the proposed approach for building a log data model.

2. Theoretical model

The following section describes the theoretical model that underpins our novel approach for generating highly realistic synthetic log data, which is called network event sequence (NES) data in the following. Fig. 1 illustrates the concept of the proposed approach for building a log data model. Since the characteristics of a log file depend on the properties of the monitored system, a part of real log data is required. Analysis of this input log data allows us to identify characterizing properties. Based on the input data a model is built, which then can be used for generating realistic NES data. Furthermore, iterative and interactive refinement of the analysis and the model allows us to modify the complexity of NES data. This means that test data for quick to in-depth analysis and evaluation of different software applications, especially anomaly detection systems (ADS), can be generated.

For putting the proposed concept into practice, our model combines log line clustering, Markov chain simulation and other methods of probability theory and statistic. Considering Fig. 1 the analysis part is covered by log line clustering and the model part by Markov chain simulation. Therefore, existing methods are extended, refined and further developed. The proposed approach is based on the following four main functions:

(i) log line clustering,
(ii) assigning log lines to clusters,
(iii) arranging clusters,
(iv) populating log lines.

During step (i) clusters are built by generating log line descriptions. Based on these descriptions regular expressions are created. Afterwards in (ii) these regular expressions are used for assigning the log lines to the clusters. In (iii) a Markov chain approach is applied for arranging the log line clusters in the NES data file. Finally in (iv) the log lines are populated with content. Therefore, on the one hand time stamps are generated and on the other hand we present three different approaches for generating log line content for various applications. In the following subsections the proposed model is described in more details. The symbols we use in this section are summarized in Table 1.

2.1. Log line clustering

First we characterize the operating principle of the clustering algorithm we use to divide the log lines of a considered log file into clusters. To perform log line clustering we apply an algorithm which was invented by Risto Vaarandi and first published in [1]. The algorithm has been especially developed for detecting word clusters in log files [2]. Furthermore, there already exists a C implementation – Simple Logfile Clustering Tool (SLCT) [3] – of the algorithm, which is open source and easy to adapt to our needs. The remaining section is partly based on [1,4].

According to Vaarandi, every *data point A* in the *data space D* corresponds to one log line in a log file. The *dimension* $n_D \in \mathbb{N}$ of the data space D is defined as the maximum number of words per line in the log file. Therefore, we define *words* of a log line the substrings of the line, separated by white spaces. Every data point A has *categorical attributes* $i_1, \ldots, i_{n_D}$. Categorical attributes are equal to the words of the log line corresponding to the data point A. The *values* $v_1, \ldots, v_{n_D}$ o $i_1, \ldots, i_{n_D}$ are strings. The j-th word of a log line is the value of the j-th attribute, where $i_j = v_j$. To simplify the attribute labels $i_1, \ldots, i_{n_D}$ are equal to the position of the word they correspond to, i.e. $i_1 = 1, i_2 = 2, \ldots, i_{n_D} = n_D$. As a result one data point $A \in D$ is a vector of strings of the form as shown in the following equation:

$$A = (x_1 = v_1, \ldots, x_{n_D} = v_{n_D}). \tag{1}$$

We define the *line length* ($l \in \mathbb{N}$) of a log line as the number of words in the line. Due to the fact that not all log lines have the same line length, all entries i_k with $l < k \leq n_D$ are set to null, i.e. they are empty.

Furthermore, let $J \subseteq \{1, 2, \ldots, n_D\}$ be a subset of indices, a *region* R is defined as a subset of D ($R \subseteq D$), where all data points $A \in R$ have the same values v_j for all $j \in J$,

$$R = \{A \in D | x_j = v_j, \ \forall j \in J\}. \tag{2}$$

This implies that $R_{fixedAttributes} = \{(i_j, v_j) | j \in J\}$ is defined as the *set of fixed attributes* of the region R. If the cardinality of $R_{fixedAttributes}$ is 1, $|R_{fixedAttributes}| = 1$ (i.e. R has just one fixed attribute), R is called 1-region. Furthermore, a *dense region* is defined as a region, which contains at least N data points, i.e. $|R| \geq N$, whereby $N \in \mathbb{N}$ is the *support threshold value*, which is specified by the user.

Table 1
Symbols for the theoretical model.

Description	*Symbol*
Dataspace	D
Dataspace dimension	n_D
Region	R
Datapoint	A
Attributes	$i_1, \ldots, i_{n_D}$
Values	$v_1, \ldots, v_{n_D}$
Log line length	l
Support threshold value	N
j-th cluster	C_i
Cluster value of the i-th cluster	cv_i
Number of clusters	n_C
Cluster size (number of lines per cluster)	r
Wild card symbol	$*$
Markov chain state space	S
Markov chain state	s
Transition matrix (in total)	$T\,(t_{ij})$
Transition matrix (probabilities)	$P\,(p_{ij})$
Initial distribution	$\mu^{(0)}$
Size of the real log file	M
Size of the NES data file	L
Time stamps	ts
Time differences	td
Vector of time differences	TD

The clustering algorithm can be structured into three steps:

1. summarize,
2. build cluster candidates,
3. select clusters from the list of candidates.

During the first step the algorithm examines the whole log file line by line and identifies all dense 1-regions. This step corresponds to mining frequent words from the log file. Note that the algorithm also takes into account the position of a word in the line. For example the 1-regions with the sets of fixed attributes {(1, 'example')} and {(3, 'example')} do not necessarily contain the same data points (log lines) and the word 'example' may have different meanings at different positions. Since frequent words correspond to dense 1-regions, a word has to occur at least N times at the same position to be considered frequent, whereby N is the support threshold value specified by the user.

After the data summarization step, the algorithm scans the log file a second time, to build *cluster candidates*. During this step all cluster candidates are stored in a table and a *support value*, which specifies how often a candidate has been generated, is associated. The algorithm processes the log file line by line and if a log line can be assigned to one or more dense 1-regions, i.e. one or more frequent words have been found in the line, the algorithm generates a cluster candidate. If the candidate is not yet stored in the table, it is added with the support value 1. Otherwise the candidate's support value is increased by 1. A cluster candidate is generated as follows: again, $J \subseteq \{1, 2, \ldots, n_D\}$ is a subset of indices with cardinality $|J| = d$ and $d \in \{1, 2, \ldots, n_D\}$. If the processed log line can be assigned to d dense 1-regions with the d fixed attributes (i_j, v_j), (i_j, v_j), with $j \in J$, the generated cluster candidate is a region R with the set of fixed attributes $R_{fixedAttributes} = \{(i_j, v_j) | j \in J\}$. For example, if the processed log line is `Connection from 192.168.1.1` and during the summarization step the algorithm found one dense 1-region with the fixed attribute (1, 'Connection') and another one with the fixed attribute (2, 'from'), the generated cluster candidate is the region R, with the set of fixed attributes $R_{fixedAttributes} = \{(1, \text{'Connection'}), (2, \text{'from'})\}$. Note that at most one cluster candidate per line can be generated. Therefore, the support value does not specify the number of lines, matching a cluster, it more or less specifies from how many lines a cluster candidate would be generated. The following example shows which kind of cluster candidates are not generated:

Example 2.1. If 'one', 'two' and 'three' are frequent words in a log file and they only occur in the combinations 'one two' and 'one three', then only these two combinations are considered as cluster candidates, 'one', 'two' and 'three' are not a cluster candidate.

During the last step the algorithm selects the *clusters* C_i from the table of candidates. Therefore, it goes through the table of cluster candidates and all dense regions are selected as clusters. Remember, dense regions are regions with a support value equal or greater than the support threshold value N. In other words, if at least N log lines are assigned to a region, it is considered as a cluster. Because of the definition of a region each cluster matches a specific line pattern. Hence, the cluster with the set of fixed attributes

$$\{(1, \text{'Connection'}), (2, \text{'from'}), (4, \text{'to'})\}$$

corresponds to the line pattern `Connection from` $*$ to, if the dimension of the data space D is $n_D = 4$. There the $*$ symbol represents a *wild card*, i.e. it serves as a placeholder for words which are not part of the set of fixed attributes of the cluster described by the line pattern.

As the described procedure shows, the proposed algorithm searches for dense regions R in subspaces of the data space D. The output of the clustering algorithm are clusters and their descriptions. Note that at this stage no log lines are assigned to the clusters. We address this part of the model in Section 2.2.

To perform the proposed log line clustering, we adapted SLCT [3], an already existing C implementation of the introduced clustering algorithm. Therefore, first the wild card symbol has to be specified for every considered log file; it may be the case that the default used symbol $*$ also represents a single word of length 1. For example the log line

`database mysql−normal` #011#01173640
`Query`#011SELECT $*$

could suggest the cluster with the cluster description

`database mysql−normal` $*$ `Query`#011SELECT $*$.

In this case it is not clear that the second $*$ represents a word instead of a wild card. Therefore, a unique character or sequence of characters, which is not occurring in the

whole log file has to be specified for representing the wild cards.

The main input parameter of SLCT is the user-specified support threshold value $N \in \mathbb{N}$, which can be given as an absolute number or in percentage (i.e. a proportion of the number of log lines in the considered log file).

Given that just the log line content is to be clustered, the time stamps should not influence the clustering. It is possible to assume that a user-specified number of bytes in the beginning of each log line are to be ignored during clustering.

The output of the clustering algorithm consists of a list of cluster descriptions and it is possible with SLCT to store all lines which are not matching to any cluster in a text file. SLCT was originally invented for detecting outliers in log files [1,4,5]. It is reasonable to parse with SLCT again through the outlier file with the same support threshold value N and by doing this new cluster candidates can be generated and occasionally also new clusters, which can be added to the list of clusters. Remembering Example 2.1 also 'one' may then become a cluster. This procedure is repeated until the length of the outlier file is smaller than the support threshold value or no new clusters can be generated. Algorithm 1 illustrates the procedure. The function `runSLCT(`*file*`,` *value*`)` runs SLCT on a specified file with a given support threshold value, `storeOutliers(`*file*`,`*clusterDescr*`.)` stores all log lines of a log file which are not matching any of the cluster descriptions generated by SLCT in a text file and the function `lengthIncreased(`*list*`)` checks if the list of cluster description increased in the last iteration.

Algorithm 1. Creation of the cluster descriptions.

Data: *logFile*, *supportThresholdValue*
Result: *clusterDescriptions*
1 *clusterDescriptions* ← runSLCT(*logFile*, *supportThresholdValue*)
2 *outlier* ← storeOutliers(*logFile*, *clusterDescriptions*)
3 **while** length(*Outlier*) ≥ *supportThresholdValue* ∧ *lengthIncreased*(*clusterDescriptions*) **do**
4 *clusterDescriptions* ← runSLCT(*outlier*, *supportThresholdValue*)
5 *outlier* ← storeOutliers(*outlier*, *clusterDescriptions*)
6 **end**

In order to use SLCT in our model we configured the algorithm so that it also allows overlapping clusters. This means that after creating the table of cluster candidates, the algorithm scans the log file once more and recalculates the support value. The support value of each candidate matched by a processed log line is therefore raised by one, so that more cluster candidates are considered as clusters, which results in a more detailed clustering.

2.2. Assigning log lines to clusters

After generating clusters the log lines have to be assigned to the clusters. We first create regular expressions based on the cluster descriptions. By means of the regular expression the model can decide if a log line matches a cluster or not.

Only using regular expressions for assigning log lines to clusters would raise the issue that one log line could match more than one cluster, i.e. the clustering would be fuzzy. The following example points out this issue. We consider the log line

`Connection from 192.168.1.1 port 123`

and the two clusters:

1. `Connect from 192.168.1.1 port` ∗ ,
2. `Connect from` ∗ port ∗ .

In this case the considered log line matches to the regular expression of both cluster descriptions. However, in our model we allow that a log line belongs only to one cluster. The reasons for this are discussed later in Section 2.3.

To achieve that every log line belongs only to one cluster, the definition of a metric is needed for deciding to which cluster a log line should be assigned, if the line matches to more than one cluster. A log line should be assigned to the most accurate cluster. If the output of our clustering algorithm could be arranged in a *graph theoretical tree* [6] with the same properties of a *dendrogram* [7], obtained with *hierarchical clustering*, this could be achieved easily. A dendrogram corresponds to a graph theoretical *in-tree*, in which each node has a pointer to its parent node. This means that only one path connects every leaf node with the root node, i.e. there are no circles. The most accurate cluster for a log line would be the leaf node (matching the considered log line) with the largest distance to the root node. If more than one cluster fulfills this conditions, the leaf node with the second largest distance to the root node has to be considered, and so on, until only one cluster fulfills the conditions. Since the output of our clustering algorithm cannot be arranged in this way, we adapt the discussed concept as follows.

We calculate the *vector of cluster values* cv (cf. Eq. (3)) for every cluster C_i, with $i \in \{1, \ldots, n_C\}$, where $n_C \in \mathbb{N}$ is the number of generated clusters. The cluster value cv_i is defined as the number of fixed attributes a cluster consists of

$$cv = (cv_1, \ldots, cv_{n_C}). \tag{3}$$

For example the cv of the cluster `Connect from` ∗ port ∗ is 3, because it consists of the fixed attributes

{(1, 'Connection'), (2, 'from'), (4, 'to')}.

All clusters matching to a log line, are stored in a list, by using the regular expressions which are generated from the cluster descriptions. The log lines are then assigned to the cluster with the highest cluster value cv_i in the list. If there is more than one cluster with the highest cluster value, the cluster with the second highest cluster value is considered. This solution corresponds to the concept discussed before, where every log line is assigned to its most accurate cluster. The hierarchy in our model is based on the cluster values cv.

Every line which does not match any cluster is considered as an *outlier*. It is also possible to configure SLCT in

a way that log lines can belong to more than one cluster, i.e. overlapping clusters are allowed. Choosing this option results in a larger number of clusters, because while deciding which cluster candidates become clusters, the support value for every cluster candidate matched by a log line is increased. Thus, the sum of all support values differs to the log file's length. Furthermore, the proposed metric we use to assign the log lines to the clusters enables creating distinct clusters. If a log line matches more than one cluster, the clusters have a common root and usually one of the clusters characterizes this root node. The other clusters then can be sorted in a hierarchy as children, i.e. leaf nodes of the root node. It is also possible that one leaf node has more than one parent node. This raises also no difficulty since we use the proposed metric. Considering the cluster descriptions `a * *`, with cluster value '$cv_1 = 1$', `a * c`, '$cv_2 = 2$', `a b *`, '$cv_3 = 2$' and `a b c`, '$cv_4 = 3$', the first one would be the root node with children `a * c` and `a b *`. Since the cluster value cv_4 of `a b c` is larger than the others, `a b c` characterizes a more specific cluster and is considered as son of `a * c` and `a b *`. Since this hierarchy exists, it is also no problem in the proposed model if a cluster includes a lower number of lines than the support threshold value. All lines, which are assigned to more specific clusters also match to their parent clusters. Hence, it might happen that in the end there are also clusters, where no line is assigned to.

Algorithm 2 illustrates the procedure we use for assigning log lines to their most accurate cluster.

Algorithm 2. Assigning log lines to their most accurate cluster.

```
Data: logFile, clusterDescriptions
Result: clusters
for 1 ≤ i ≤ length(clusterDescriptions) do
    clusterValue_i ←
      calculateClusterValue(clusterDescription_i)
end
for 1 ≤ i ≤ length(logFile) do
    for 1 ≤ j ≤ length(clusterDescriptions) do
        if matches(logFile_i, clusterDescriptions_j)
          then
            matchingClusters ← clusters_j
        end
    end
    mostAccurateCluster ←
      findMostAccurateCluster(matchingClusters)
    mostAccurateCluster ← logFile_i
end
```

Algorithm 3 characterizes, how the most accurate cluster is chosen. The function `getClusterByValue` (*cluster*, *cv*) returns the cluster, corresponding to the previously chosen cluster value.

Algorithm 3. Determining the most accurate cluster matching to a log line.

```
Data: matchingClusters,
  clusterValuesOfMatchingClusters
Result: mostAccurateCluster
sortedValues ←
  sort(clusterValuesOfMatchingClusters)
Data: matchingClusters,
index ← length(sortedValues)
while occurrence(sortedValues_index) ≠ 1 do
    index ← index − 1
    if index < 0 then
        considered line is an outlier
        return
    end
end
mostAccurateCluster ←
  getClusterByValue(matchingClusters,
  sortedValues_index)
```

2.3. Arranging clusters in the NES data file

The next step in the proposed model is to arrange the clusters in the generated NES data file. We apply a Markov chain approach [8,9], which is based on the generation of a series of random events. In the remaining section, we describe how to arrange the clusters in the generated NES data file by simulating a homogeneous Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space S, transition matrix P and initial distribution $\mu^{(0)}$.

First, we calculate the transition matrix P. In Section 2.2 we already mentioned that we need distinct clusters. We achieve this by choosing the most accurate cluster. Hence, it is possible to calculate the probability at which one cluster follows another one. The number of transitions t_{ij} from cluster C_i to cluster C_j with $i,j \in \{1,\ldots,n_C+1\}$ is stored in a Matrix $T \in \mathbb{N}^{(n_C+1)\times(n_C+1)}$:

$$T = \left(\begin{bmatrix} t_{11} & \cdots & t_{1n_C+1} \\ \vdots & \ddots & \vdots \\ t_{n_C+11} & \cdots & t_{n_C+1n_C+1} \end{bmatrix} \right). \tag{4}$$

The last index here is n_C+1 (instead of n_C, the number of clusters), because the outliers are considered as an extra cluster.

The transition probabilities p_{ij} from cluster C_i to cluster C_j are calculated, as shown in the following equation:

$$p_{ij} = \frac{t_{ij}}{\sum_{k=1}^{n_C+1} t_{ik}}, \quad \text{for all } i,j \in \{1,\ldots,n_C+1\}. \tag{5}$$

The transition probabilities are then stored in a transition Matrix $P \in \mathbb{R}^{(n_C+1)\times(n_C+1)}$ (cf. Eq. (6)), also called *stochastic matrix* [10].

$$P = \left(\begin{bmatrix} p_{11} & \cdots & p_{1n_C+1} \\ \vdots & \ddots & \vdots \\ p_{n_C+11} & \cdots & p_{n_C+1n_C+1} \end{bmatrix} \right) \tag{6}$$

Since P is a stochastic matrix each row of P represents a probability distribution. This implies that the elements p_{ij} of the transition matrix P have to satisfy conditions in the following equations:

$$0 \le p_{ij} \le 1 \quad \text{for all } i,j \in \{1,\ldots,n_C+1\}, \tag{7}$$

$$\sum_{j=1}^{n_C+1} p_{ij} = 1 \quad \text{for all } i \in \{1,\ldots,n_C+1\}. \tag{8}$$

In the next step we estimate the initial distribution $\mu^{(0)} \in \mathbb{R}^{n_C+1}$. The elements of the initial distribution $\mu^{(0)}$ are the ratios between the row sums and the total sum of

elements of T as pointed out in the following equation:

$$\mu_i^{(0)} = \frac{\sum_{l=1}^{n_C+1} t_{il}}{\sum_{k=1}^{n_C+1}\sum_{l=1}^{n_C+1} t_{kl}} \quad \text{for all } i \in \{1,\ldots,n_C+1\}. \tag{9}$$

Since $\mu^{(0)}$. represents a probability distribution it has to satisfy the conditions in the following equations:

$$0 \le \mu_i^{(0)} \le 1 \quad \text{for all } i \in \{1,\ldots,n_C+1\}, \tag{10}$$

$$\sum_{i=1}^{n_C+1} \mu_i^{(0)} = 1 \quad \text{for all } i \in \{1,\ldots,n_C+1\}. \tag{11}$$

Finally the clusters in the generated NES data file can be arranged by simulating the Markov chain $\{X_t; t \in \mathbb{N}\}$, with the state space $S = \{C_1,\ldots,C_{n_C+1}\}$, transition matrix P and initial distribution $\mu^{(0)}$. For simulating the Markov chain and to arrange the clusters in the generated NES data file, we apply an approach proposed in [11, Chapter 3]. To perform the simulation of the Markov chain we primarily need:

- a sequence $\{U_t; t \in \mathbb{N}\}$ of independent and identical distributed random numbers, uniformly distributed in the unit interval $[0,1]$,
- an initiation function ψ and
- an update function ϕ.

The initiation function is a function $\psi: [0,1] \to S$, which we use to generate the starting value X_0. We assume that ψ is piecewise constant on the interval $[0,1]$ and for each $s \in S$ the total length of the intervals, on which $\psi(x) = s$, is equal to $\mu^{(0)}(s)$. This corresponds to Eq. (12). Note that $\mathbb{1}_{\{s\}}(x)$ defines the so-called indicator function:

$$\int_0^1 \mathbb{1}_{\{s\}}(\psi(x))dx = \mu^{(0)}(s) \quad \text{for all } s \in S, \tag{12}$$

$$\mathbb{1}_{\{s\}}(x) = \begin{cases} 1, & \text{if } x = s, \\ 0, & \text{else.} \end{cases} \tag{13}$$

Given such an initiation function ψ, we can use the first random number U_0 to generate the starting value X_0 by setting $X_0 = \psi(U_0)$. Thus we get the correct distribution of X_0, because for any $s \in S$ Eq. (14) is valid:

$$P(X_0 = s) = P(\psi(U_0) = s) = \int_0^1 \mathbb{1}_{\{s\}}(\psi(x))\, dx \overset{(12)}{=} \mu^{(0)}(s). \tag{14}$$

Based on the previously mentioned properties, ψ (defined in Eq. (15)) represents a valid initiation function, being piecewise constant on the interval $[0,1]$ and satisfying the following equation:

$$\psi(x) = \begin{cases} C_1 & \text{for } x \in \left[0, \mu^{(0)}(C_1)\right), \\ C_2 & \text{for } x \in \left[\mu^{(0)}(C_1), \mu^{(0)}(C_1)+\mu^{(0)}(C_2)\right), \\ \vdots & \\ C_i & \text{for } x \in \left[\sum_{j=1}^{i-1}\mu^{(0)}(C_j), \sum_{j=1}^{i}\mu^{(0)}(C_j)\right), \\ \vdots & \\ C_{n_C+1} & \text{for } x \in \left[\sum_{j=1}^{n_C+1}\mu^{(0)}(C_j), 1\right]. \end{cases} \tag{15}$$

The update function $\phi: S \times [0,1] \to S$ is a function, we use to generate the value X_{t+1} from X_t and U_{t+1} for any $t \in \mathbb{N}^{>0}$. We assume that the function $\phi(s_i, x)$ is for fixed $s_i \in S$ piecewise constant (when ϕ is considered as function of x). Furthermore, for each fixed $s_i, s_j \in S$, the total length of the intervals, on which $\phi(s_i, x) = s_j$, is equal to the transition probability p_{ij}. This corresponds to the following equation:

$$\int_0^1 \mathbb{1}_{\{s_j\}}(\phi(s_i,x))\, dx = p_{ij} \quad \text{for all } s_i, s_j \in S. \tag{16}$$

If ϕ satisfies Eq. (16) then Eq. (17) is valid:

$$\begin{aligned} P(X_{t+1} = s_j | X_t = s_i) &= P(\phi(s_i, U_{t+1}) = s_j | X_t = s_i) \\ &= P(\phi(s_i, U_{t+1} = s_j)) = \int_0^1 \mathbb{1}_{\{s_j\}}(\phi(s_i,x))\, dx \overset{(16)}{=} p_{ij}. \end{aligned} \tag{17}$$

$P(\phi(s_i, U_{t+1}) = s_j | X_t = s_i) = P(\phi(s_i, U_{t+1} = s_j))$ is valid in Eq. (17), because U_{t+1} is independent of $(U_1,\ldots,U_t)$, and thus also of X_t. Due to the same reason the probability remains the same if we condition with the values $(X_0,\ldots,X_{t-1})$. Hence the described procedure provides a correct simulation of the Markov chain.

Based on the previously mentioned properties, ϕ (defined for each $C_i \in S$ in Eq. (18)) represents a valid update function, being piecewise constant on the interval $[0,1]$ and satisfying the following equation:

$$\phi(C_i, x) = \begin{cases} C_1 & \text{for } x \in \left[0, p_{i1}\right), \\ C_2 & \text{for } x \in \left[p_{i1}, p_{i1}+p_{i2}\right)), \\ \vdots & \\ C_j & \text{for } x \in \left[\sum_{l=1}^{j-1} p_{il}, \sum_{l=1}^{j} p_{il}\right), \\ \vdots & \\ C_{n_C+1} & \text{for } x \in \left[\sum_{l=1}^{n_C+1} p_{il}, 1\right]. \end{cases} \tag{18}$$

We described a procedure which allows simulation of the homogeneous Markov chain $\{X_t; t \in \mathbb{N}\}$, with state space $S = \{C_1,\ldots,C_{n_D+1}\}$, initial distribution $\mu^{(0)}$ and transition matrix P. Using a sequence $\{U_t; t \in \mathbb{N}\}$ of independent and identical distributed random numbers, uniformly distributed on the unit interval $[0,1]$, we obtain Eq. (19). The number of values to be generated can be specified by the user:

$$\begin{aligned} &X_0 = \psi(U_0), \\ &X_i = \phi(X_{i-1}, U_i) \quad i \in \mathbb{N}\backslash\{0\}. \end{aligned} \tag{19}$$

In case the input log file models a irreducible Markov chain, the generated Markov chain has also to be irreducible. This means that starting from any state $s_i \in S$, each state $s_j \in S$ has to be reachable in any number of steps. If this is not possible our model does not reach every cluster C_i, or it deadlocks in a small set of clusters. It can be decided if the Markov chain is irreducible by checking Eq. (20), where $B \in \mathbb{R}^{(n_C+1)\times(n_C+1)}$ and I is the (n_C+1)-dimensional unit matrix. The Markov chain then is irreducible if all entries of B are unequal to zero:

$$B = I + P + P^2 + \cdots + P^{n_C+1}. \tag{20}$$

Note that at this stage we have only ordered the clusters previously generated. Each value X_i of the simulated

Markov chain represents a placeholder for a log line, which matches the cluster that the value X_i corresponds to. After determining the correct order of log line clusters in the output file time stamps and log line content have to be generated.

2.4. Populating log lines

A log line consists of two main building blocks: a time stamp and content. While the time stamp defines when the log line was created, the content specifies which event is described by the log line. There exist different formats for time stamps. In the following we use the format `MMM dd hh:mm:ss`, where `MMM` defines the month, `dd` the day and `hh:mm:ss` the time, when the log line was created. The following section deals with generating time stamps and log line content for the generated log file.

2.4.1. Generating time stamps

For generating time stamps, we have to make some assumptions. In the proposed model we assume that the time stamps are independent from the log line content, since the occurrence of the logged event is independent from the time. But the time stamps are depending on the cluster the log line belongs to. In other words, the interval between two consecutive log lines depends on the clusters the log lines belong to. Therefore we assume that the time it takes until the next log line occurs depends on the cluster the last log line belongs to.

We define the *size* of the considered log file $M \in \mathbb{N}$, the number of lines it contains. Hence, there are *M time stamps* ts_j, with $j \in \{0, \ldots, M-1\}$, in the log file and $M-1$ transitions between log lines. This also means that there can be $M-1$ *time differences* td_j, with $j \in \{1, \ldots, M-1\}$, calculated,

$$td_j = ts_j - ts_{j-1}, \quad j \in \{1, \ldots, M-1\}. \tag{21}$$

For every cluster C_i a sequence of time differences TD_i, with $i \in \{1, \ldots, n_C+1\}$ specifying the cluster, is stored. If the log line j belongs to cluster C_i, the time difference td_j is added to $TD_i \in \mathbb{N}^r$, where $r \in \mathbb{N}$ is the size of the cluster C_i, i.e. the number of lines assigned to cluster C_i.

We then build an empirical distribution function (EDF) [12] based on the elements of TD_i. An EDF is defined as follows:

Definition 2.2. Let $X_1, \ldots, X_n$ be elements of a sample. A function $F: \mathbb{R} \to [0,1]$ as defined in Eq. (22) is named a *empirical distribution function*.

$$F(x) := \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{(-\infty, x]}(X_i) \tag{22}$$

The EDF describes the distribution of elements of TD_i. Therefore we use the quantile function (inverse cumulative distribution function) Q [12] of the EDF F (defined in Definition 2.3) to generate random time differences with the same distribution as in the input file.

Table 2
Symbols for the evaluation.

Description	*Symbol*
Log file length	n
Length of the i-th log line	l_i
i-th cluster	C_i
Cluster value of the i-th cluster	cv_i
Support threshold value	N
Mean coverage rate	*MCR*
Number of outliers	*NoO*
Number of clusters	*NoC*
Difference of the relative cluster frequencies	*DRCF*
Network event sequence	NES
Original log file	LF_{orig}
Generated NES data file	LF_{NES}

Definition 2.3. Let F be an EDF, then Q in Eq. (23) defines the *quantile function* of F:

$$Q(p) = F^{-1}(p) := \inf\{x \in \mathbb{R} | p \le F(x)\}, \quad 0 < p \le 1. \tag{23}$$

We now use a random number U and the quantile function Q, for generating the time stamps. The function $td_{rand}(U, TD_i): [0,1] \times \mathbb{N}^r \mapsto \mathbb{N}$, defined in Eq. (24), where F is the EDF based on the elements of TD_i, generates a random time difference, based on the distribution of the values of TD_i:

$$td_{rand}(U, TD_i) := \inf\{x \in \mathbb{N} | U \le F(x)\}. \tag{24}$$

The time stamp for every generated log line is calculated as shown in Eq. (25), where TD_i is the vector of time differences of the cluster C_i, the generated log line belongs to and $L \in \mathbb{N}$ defines the *size* of the generated NES data file. Note that the first time stamp ts_1 has to be specified by the user:

$$ts_j = ts_{j-1} + td_{rand}(U, TD_i), \quad j = 2, \ldots, L. \tag{25}$$

2.4.2. Generating log line content

The remainder of this section describes our approach for generating log line content. We provide three options which allow us to generate log line content of different complexity. This is relevant for example for applications, where the log line length matters. Our approach for generating log line content is based on the cluster descriptions. The three options mainly differ in the way the wild card symbol ∗ in the cluster descriptions is replaced. In the following the cluster with the description

```
Connect from * port *
```

serves as example.

The first option is the most straightforward and simple one. The log line content simply consists of the cluster description, without wild card symbols. For example if a line, which belongs to the example cluster, is generated it looks as follows:

```
MMM dd hh : mm : ss Connect from port
```

where `MMM dd hh:mm:ss` represents the time stamp. This is a good option to generate a NES data file which only

Table 3
Overview of the three common types of test data: synthetic, real and semi-synthetic, including their advantages and disadvantages [13].

Data origin	*Advantage*	*Disadvantage*
Synthetic	Easy to (re-)produce, has desired properties, no unknown properties	No realistic 'noise' mostly simplified situations
Real	Realistic test basis	Bad scalability (user input, varying scenarios), privacy issues, attack on own system needed
Semi-synthetic	More realistic than synthetic data, easier to produce than real data	Simplified and biased if an insufficient synthetic user model applied

reproduces the line sequence of the considered input log file.

```
MMM dd hh : mm : ss Connect from < < < < < < < < < < port < < <
```

Next we introduce an opportunity which allows us to generate log line content of higher complexity. The wild card symbols in the cluster descriptions are replaced with words which also occur in the input log file at the same position. This can be achieved by using a similar approach as the one proposed for generating the time distances td_{rand}, previously in this section. For every wild card symbol $*$ all occurring words at its position are stored in a list. Note that words even are stored if they already occurred so that also their relative frequency is correct. Afterwards, while generating the log line content, every wild card symbol is replaced by one word out of the related list. This works exactly the same way as the generation of time differences *td*, which is shown in Eq. (24). For example we consider the example cluster and the log lines

1. `Connect from 192.168.1.1 port 123`
2. `Connect from 192.168.1.7 port 456`.

Both log lines match to the example cluster. Here the first wild card can be replaced by `192.168.1.1` or `192.168.1.7` and the second one by `123` or `456`. Thus, there are four different options of log lines which can be generated if a log line belonging to the example cluster is produced. This procedure has the advantage that the distribution of the log line length in the generated NES data file resembles the one of the input log files. Furthermore IP addresses are only replaced by IP addresses, and since the generated NES data file can be arbitrarily long, some randomness is kept.

The third proposed option can be used for more specific purposes. The wild card symbols are replaced by sequences of a character which is not part of any cluster description. To choose the length of the sequences we use again the same approach as for generating the time difference *td*. For every wild card the length of the words it replaces is stored in a list. Since reoccurring values are also stored, using a function as in Eq. (24) generates word length values with the correct distribution. Considering the example in the previous paragraph, $<$ is a unique symbol. The first wild card replaces a word with 11 characters, and the second one a word with three characters. Therefore, the log line

would be generated.

This option for generating log line content is useful, to evaluate an algorithm which depends on frequent words and the log line length. An example is automatic pattern generation algorithms which try to find frequent patterns. Here the patterns should cover the words that define the cluster descriptions. For easier analysis the rare content is replaced by a sequence of a unique character, of the length corresponding to the length of the replaced content.

In case the cluster representing the outliers occurs only a time stamp has to be generated. For the content a line of the list of outliers is chosen.

3. Evaluation and discussion

The following section deals with the evaluation of the proposed approach for generating realistic NES data. Here we focus on verifying the high quality of the generated NES data. Later Sections 4 and 5 aim at a quantitative evaluation by demonstrating the application areas of the proposed approach. For evaluating and testing the introduced model for generating NES data we implemented a prototype – Log File Generator (LFG)[1] – of the previously defined model as a Java application. To perform the log line clustering as presented in Section 2.1, we adopted SLCT [1], which provides a C implementation of the applied clustering algorithm. The other parts of the model have been implemented from scratch.

The section is organized as follows: first we describe the input data we use for the evaluation. Afterwards the effects caused by changing the support threshold value *N* are analyzed and criteria for choosing the optimal support threshold value are discussed. Finally we evaluate the Markov chain simulation and the wild card replacement. The symbols we use in this section are summarized in Table 2.

[1] https://github.com/MarWur/LFG.git.

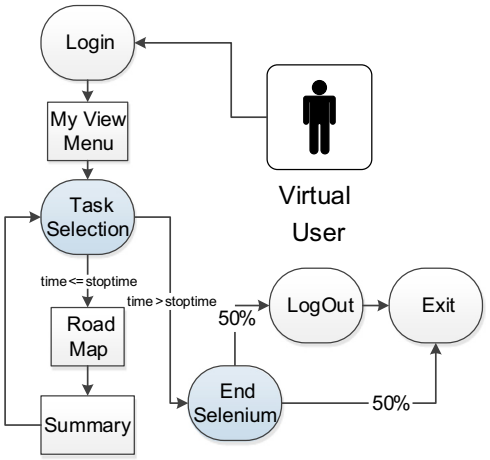

Fig. 2. This figure shows configuration I, which we use for generating semi-synthetic log data. The complexity in this configuration is kept relatively low. As long as the (passed) time is smaller than the recorded time (specifies how long the user actions are logged) a simulated user is accessing the same two webpages again and again. In this configuration only the time when the user does this is variable. After the recorded time expires, the user logs out and then exits with a probability of 50% and exits directly with a probability of 50%.

Table 4
Properties of the exploited semi-synthetic log files.

Data set	*Simulated users*	*Recorded time (h)*	*Data set length (lines)*	*Used configuration*
U1C1	1	10	484.239	Config I
U4C1	4	10	1.887.824	Config I
U1C2	1	10	413.106	Config II
U4C2	4	10	1.600.217	Config II

3.1. Log data for testing the proposed model

For the evaluation of the presented approach on the one hand we use semi-synthetic log data and on the other hand exploit a reference log file we took from a running productive system. The properties of these log files and the reason why we chose those are discussed in the following.

3.1.1. Generating semi-synthetic log data

For evaluating the quality of the proposed model's output according to the complexity of the input data, using real data only is not sufficient. The difficulty is that the complexity of different log data sets from a running productive system cannot be easily compared. Therefore we decided to generate semi-synthetic log files (cf. Table 3) for testing the proposed approach. Table 3 outlines the differences between synthetic, real and semi-synthetic test data.

For generating semi-synthetic log data, we applied the approach presented in [13]. This method allows us to generate log files of any size and of different complexities for one specific given system. Virtual users perform specified actions on a Web platform, running the MANTIS Bug Tracker System [13]. In the log files the events from a Web server, a database and a firewall are included. Furthermore, it is possible to specify the number of users operating on the system. Because of the fact that one can choose which actions are performed in which order by which probability, the complexity of the generated log data can be adjusted easily. It is further possible to configure the distribution of the time intervals between two consecutive user actions. Therefore highly realistic conditions can be simulated. The actions that the virtual users can perform vary from clicking on links to looking up and creating entries in the database. But the opportunity of simulating various systems with the approach presented in [13] is limited due to the required resources and the required time setting up such a system. Since the logged platform is also used in real settings by companies for managing bugs in their software, the produced semi-synthetic log data is representative.

For evaluating the proposed approach we generated 4 different log files by applying the approach from [13]. In order to simulate different levels of complexity we implemented two configurations – configuration I (low complexity, cf. Fig. 2) and configuration II (high complexity, see [13]). For generating the log files the user activity was logged for 10 h. Table 4 shows that the data set length, i.e. number of log lines, is mostly effected by the number of simulated users. In both cases (running one virtual user and running four concurrent virtual users) changing from configuration I to configuration II generated around 15% less log lines. This happens because in configuration II there are more options for the virtual users to choose their actions, because in configuration II there are more actions which raise a longer waiting time until a virtual user starts his next action.

3.1.2. Reference log data

We further demonstrate that our novel approach also works for reference log data obtained from a running productive system. Therefore we use a system consisting of an Apache Web server, a firewall server and a monitoring server. On the Apache server a static website is set up. The connections between the Web server and the monitoring server are observed by the firewall. The occurring network events are logged in the considered log file.

For the evaluation we consider a part of the log file, where 24 h are logged. The file has a length of 436.613 log lines. The configuration of the system, where the log data is obtained from, k is also called *REF* in the following.

3.2. Analysis of the effects caused by changing the support threshold value N

The support threshold value N, which specifies how many lines at least have to be assigned to a cluster candidate to become a cluster, is the main input parameter of the proposed model. In the following section we analyze how changing the support threshold value N effects the output of the clustering algorithm described in Section 2.1. The clustering algorithm should achieve the following two objectives:

(i) the cluster description of the cluster a log line is assigned to should cover a large percentage of the log line content,
(ii) there should be a low number of outliers.

To evaluate the clustering algorithm, we ran SLCT with support threshold values N from 5% (0.05) to 0.1% (0.001), decreasing N by 0.1% (0.001) in every iteration. We did this for all of the four test log datasets summarized in Table 4 and the reference log file described in Section 3.1.2. We then analyzed the *Mean Coverage Rate* (*MCR*), the *Number of Outliers (NoO)* and the *Number of Clusters* (*NoC*).

3.2.1. The mean coverage rate (MCR)

The *MCR* is a metric, which gives knowledge about the proportion of the log line, which is covered by the matching cluster's description. For calculating the *MCR*, we define $n \in \mathbb{N}$ as the length, i.e. number of lines, of the considered log file, l_i as the length of the ith log line of the considered log file, and cv_i as the cluster value (cf. Eq. (3)) of the cluster the ith log line has been assigned to. Then the *MCR* of a log file can be calculated as shown in Eq. (26), where $\frac{cv_i}{l_i}$, with $i \in \{1, \ldots, n\}$ specifies the coverage rate for every log line:

$$MCR = \frac{1}{n} \sum_{i=1}^{n} \frac{cv_i}{l_i}. \tag{26}$$

The progression of the *MCR* is shown in Fig. 3 and some of the interesting values are summarized in Table 5. Fig. 3 demonstrates that the *MCR* mainly depends on the configuration used to build the log files. It is independent from the number of simulated users and also from the length of the log files. The *MCR* mainly depends on the used configuration because every virtual user acts with the same probability. Therefore the distribution of the occurring log lines is independent from the number of simulated virtual users. Similar results can be expected for the progression of the number of clusters. According to the *MCR* the clustering algorithm performs a bit better with (the less complex) configuration I. The largest gap between the files which use configuration I and the files which use configuration II can be recognized for support threshold values $N \in [0.008, 0.028]$.

In contrast to the semi-synthetic log files, the *MCR* for the reference log file (cf. Section 3.1.2) is already very high for large support threshold values. This suggests a lower complexity of the system. The fact that the *MCR* sometimes decreases (with decreasing N) can be explained by the circumstance that depending on the support threshold value the log lines are assigned to different clusters with different cluster descriptions.

3.2.2. The number of outliers (NoO)

Since the *NoO* is represented in total numbers, Fig. 4 suggests that the *NoO* progression depends on either the configuration or the number of simulated virtual users, which refers to the log file length. The graphs of the *NoO*

Table 5
Important *MCR* values. See also Fig. 3.

N	*U1C1*	*U4C1*	*U1C2*	*U4C2*	*REF*
0.05	0.4721	0.4808	0.4705	0.4747	0.7423
0.03	0.5583	0.5639	0.5184	0.5225	0.8049
0.028	0.6662	0.6661	0.5365	0.5424	0.7909
0.013	0.7409	0.7206	0.6242	0.6295	0.8807
0.008	0.7736	0.7493	0.7664	0.7560	0.9195
0.001	0.9486	0.9320	0.8953	0.8978	0.9165

Table 6
Important *NoO* values. See also Fig. 4.

N	*U1C1*	*U4C1*	*U1C2*	*U4C2*	*REF*
0.05	6860	102,327	10,312	37,141	114
0.03	6860	25,466	10,312	37,141	114
0.023	6860	25,466	40	40	114
0.013	40	40	40	40	114
0.001	1134	20	0	0	114

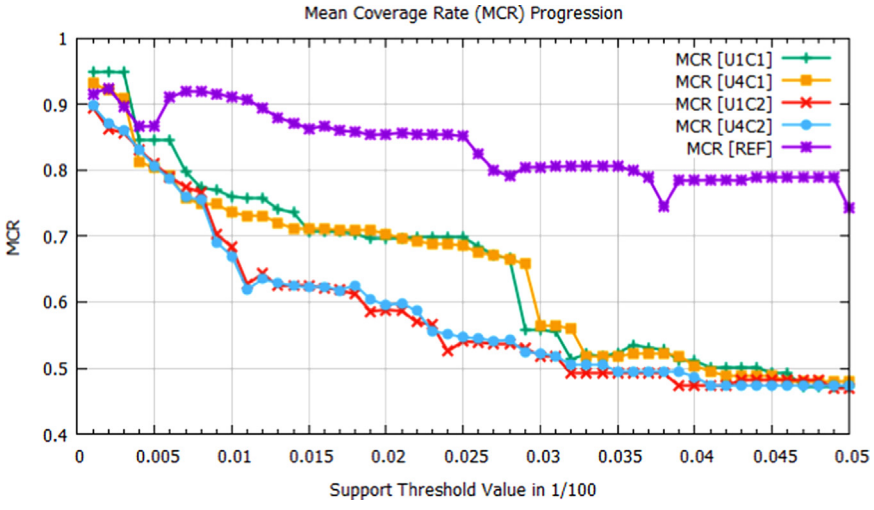

Fig. 3. Progression of the *MCR* for the log files described in Table 4 and the reference log file described in Section 3.1.2. Important values are summarized in Table 5.

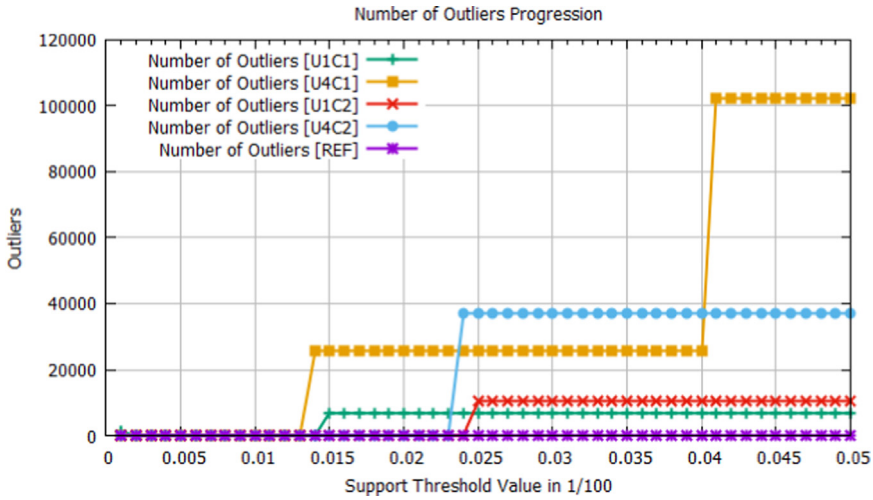

Fig. 4. Progression of the *NoO* for the log files described in Table 4 and the reference log file described in Section 3.1.2. Important values are summarized in Table 6.

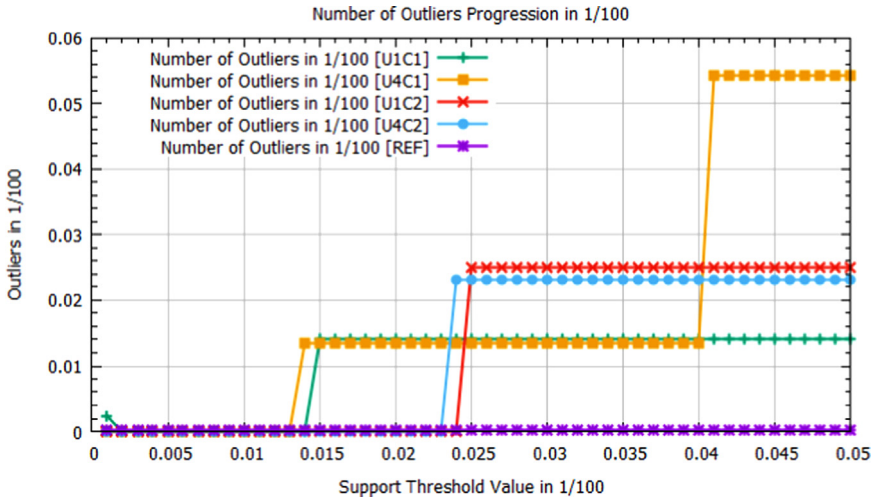

Fig. 5. Progression of the *NoO* in $\frac{1}{100}$ for the log files described in Table 4 and the reference log file described in Section 3.1.2.

are also more constant, than the ones of the *MCR* progression. The *NoO* of the files in which four virtual users were active is significantly higher than the *NoO* of the files in which only one virtual user has been simulated. But the graphs regarding the same configuration show a similar trend. This can better be seen in Fig. 5, where $\frac{NoO}{n}$, i.e. the percentage of outliers, is plotted. The *NoO* for the log files, where configuration II has been used, is nearly 0 for support threshold values $N \leq 0.03$. For the log files where configuration I has been used the *NoO* is nearly 0 for $N \leq 0.013$.

The *NoO* for the reference log file (cf. Section 3.1.2) is constantly 114, which corresponds to 0.03% of the file length. That the *NoO* is already low for high support threshold values could be expected, since also the *MCR* was already large for high support threshold values. Since the graphs for the *NoO* of the semi-synthetic log files are also piecewise constant, the constant course of the *NoO* of the reference log file is consistent with these results.

3.2.3. The number of clusters (NoC)

The progression of the *NoC* is shown in Fig. 6 and some relevant values are summarized in Table 7. As previously mentioned the *NoC* mainly depends on the used configuration. Therefore the graphs in Fig. 6 belonging to the log files which have been generated using the same configuration are nearly congruent. One would expect a bigger *NoC* for the log files with the more complex configuration II, since they contain more different log lines. But for support threshold value $N \in [0.011, 0.05]$ no big differences in the *NoC* can be recognized. On this interval the *NoC* also does not increase very fast. For $N < 0.011$ the *NoC* of all log files increases faster. Then also the *NoC* for files with configuration II gets bigger than the *NoC* of files with configuration I. For support threshold value $N > 0.023$ the *NoC* in configuration I is even higher than the *NoC* of log files using configuration II. This phenomenon can be explained as follows: in Section 2.2 we mentioned that the clusters generated with SLCT can be arranged in a kind of

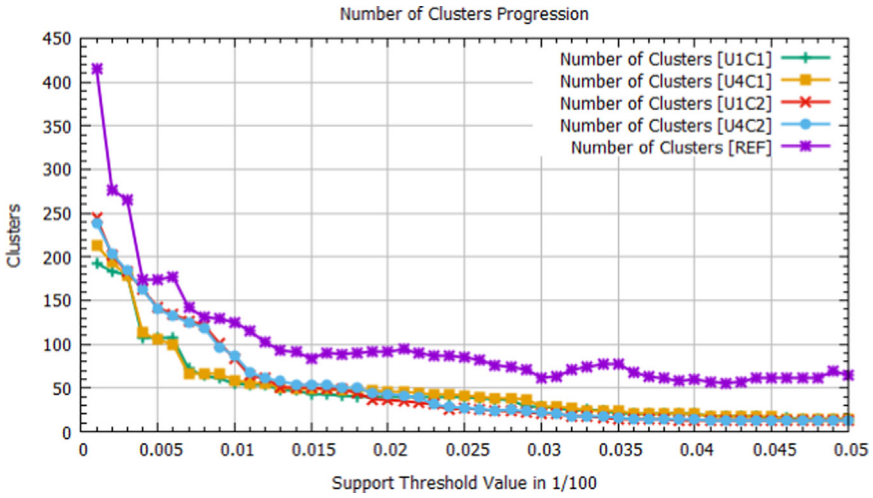

Fig. 6. Progression of the *NoC* for the log files described in Table 4 and the reference log file described in Section 3.1.2. Important values are summarized in Table 7.

Table 7
Important *NoC* values. See also Fig. 6.

N	*U1C1*	*U4C1*	*U1C2*	*U4C2*	*REF*
0.05	15	15	12	12	65
0.028	36	38	23	25	74
0.013	48	51	51	58	93
0.008	64	67	122	119	131
0.003	179	178	183	184	265
0.001	193	213	245	239	415

graph theoretical tree. When decreasing the support threshold value *N* more specific clusters are generated which often have the same roots as clusters from previous iterations. When *N* becomes smaller SLCT starts earlier to build more specific clusters for simpler log files, such as the ones obtained with configuration I.

The *NoC* of the reference log file (cf. Section 3.1.2) is always lager than the *NoC* of the semi-synthetic log files. But the course of the graph in Fig. 6 is similar to the graphs of the semi-synthetic log data.

3.2.4. How to choose a suitable support threshold value N

The *MCR* and the *NoO* can be used to predict a support threshold value *N* that fulfills the two objectives mentioned at the beginning of the section:

(i) the cluster description of the cluster a log line is assigned to should cover a large percentage of the log line content,
(i) there should be a low number of outliers.

To address objective (i) a criterion could be a specific threshold value for the *MCR*, such as $MCR \geq 0.7$. Table 8 shows, for which *N* this assumption is fulfilled.

To address objective (ii) the *NoO* must be considered. Since the *NoO* depends on the log file length, the fraction of the *NoO* and log file length which corresponds to the percentage of outliers (cf. Fig. 5) is of importance. Again a threshold value for the rate of outliers $\frac{NoO}{n}$ is chosen. For example $\frac{NoO}{n} \leq 0.01$ is assumed, which means less than 1% outliers. Table 9 shows for which *N* the in-equation holds.

To fulfill both requirements ($MCR \geq 0.7$ and $\frac{NoO}{n} \leq 0.01$) we have to consider for each log file the minimum support threshold values *N* between Tables 8 and 9. The results are shown in Table 10.

The results of this analysis show that for the reference log file (cf. Section 3.1.2) even a support threshold value higher than 5% of the log file length can be sufficient, since for $N = 5\%$ the *MCR* is already higher than 0.7 and the $\frac{NoO}{n}$ lower than 0.01.

The presented procedure for determining an accurate support threshold value *N* can be applied for any threshold values for the criteria regarding the *MCR* and $\frac{NoO}{n}$.

3.3. Evaluating the Markov Chain approach

In this section we evaluate the output of the Markov chain simulation we applied to generate NES data (LF_{NES}). On the one hand we want to show that the transitions between consecutive clusters reflect the sequence of the log lines in the original log file (LF_{orig}), and on the other hand we want to show that we generate meaningful log line content. Therefore, we first just look at the transitions without considering the log line content. Afterwards we also evaluate how replacing the wild cards influences the log file model.

3.3.1. Evaluating the transitions

Since we use a Markov chain simulation for generating LF_{NES}, the transition probabilities of LF_{orig} and LF_{NES} are by construction the same if the number of generated lines tends to infinity. First we look at the transitions without considering the log line content. We generated for each LF_{orig} a LF_{NES}, using the support threshold value given in Table 10. For analyzing the transitions in LF_{NES} only the cluster of each generated log line is stored.

First we consider the *cluster relative frequencies CRF*. The *CRF* of a cluster C_i after $m \in \mathbb{N}$ lines is calculated as shown in Eq. (27), where $\mathbb{1}_{\{C_i\}}$ is the indicator function (cf.

Eq. (13)), which is 1, if the jth generated line l_j is an element of cluster C_i and 0 if not:

$$CRF(C_i, m) = \frac{1}{m} \sum_{j=1}^{m} \mathbb{1}_{\{C_i\}}(l_j), \quad \text{for all } i \in \{1, \ldots, NoC\}. \tag{27}$$

In Fig. 7 we consider the progression of the *difference of the relative cluster frequencies DRCF* between LF_{orig} and LF_{NES}.

Table 8
Support threshold value N with $MCR \geq 0.7$ per log file.

Log file	$N \leq$
U1C1	0.018
U4C1	0.020
U1C2	0.009
U4C2	0.008
REF	0.05

Table 9
Support threshold value N with $\frac{NoO}{n} \leq 0.01$ per log file.

Log file	$N \leq$
U1C1	0.014
U4C1	0.013
U1C2	0.024
U4C2	0.023
REF	0.05

Table 10
Options for N according to the assumptions $MCR \geq 0.7$ and $\frac{NoO}{n} \leq 0.01$ per log file.

Log file	$N \leq$ (in $\frac{1}{100}$)	$N \leq$ (in lines)	$MCR \geq$	$\frac{NoO}{n} \leq$	*Cluster*
U1C1	0.014	6779	0.7363	0.000083	47
U4C1	0.013	24,541	0.7206	0.000021	51
U1C2	0.009	3717	0.7037	0.000048	101
U4C2	0.008	12,801	0.7560	0.000012	119
REF	0.05	21,831	0.7423	0.000261	65

The *DRCF* of a log file after generating m log lines is calculated as shown in Eq. (28), where *relFreq* returns the relative frequency of a cluster in LF_{orig} (cf. Eq. (29), where $n \in \mathbb{N}$ specifies the length of LF_{orig}):

$$DRCF(m) = \frac{1}{NoC} \sum_{i=1}^{NoC} \left| relFreq(C_i^{orig}) - CRF(C_i^{NES}, m) \right|, \tag{28}$$

$$relFreq(C_i) = CRF(C_i, n), \quad \text{for all } i \in \{1, \ldots, NoC\}. \tag{29}$$

Fig. 7 shows the progression of the *DRCF* for the five considered log files while generating two million log lines.

The graph points out that the *DRFC* mainly depends on the number of clusters (cf. Table 10), since the more different clusters are built the more log lines have to be generated until a specific distribution is reached. The largest gap between the different log files can be recognized during generating the first 400.000 lines. Furthermore the figure demonstrates, that with an increasing number of generated log lines the *DRCF* converges to zero. This could be expected since the transition probabilities of LF_{NES} converge towards the transition probabilities of LF_{orig}, when the number of generated log lines tends to infinity. Furthermore the figure shows that the *DRCF* of each log file is already smaller than 0.01, i.e. 1%, when the number of generated log lines reaches the length of LF_{orig}.

Since the outliers are considered as an own cluster during the generation step (cf. Section 2.3), the relative frequency of the outliers in LF_{NES} must be similar to the relative frequency of the outliers in LF_{orig}.

In Fig. 8 we illustrate the difference between the transitions of LF_{orig} and LF_{NES} in case of configuration U1C1; both files have the same length (484.239 lines). We calculate T^{diff} (cf. Eq. (30)) the difference of the transition matrices T^{orig} and T^{NES} (cf. Eq. (4)). We normalize the difference over the log file length n so that the cluster size does not effect the value, i.e. we consider the difference between the relative frequency of the transitions:

$$t_{ij}^{diff} = \frac{\left| t_{ij}^{orig} - t_{ij}^{NES} \right|}{n}, \quad \text{for all } i,j \in \{1, \ldots, NoC\}. \tag{30}$$

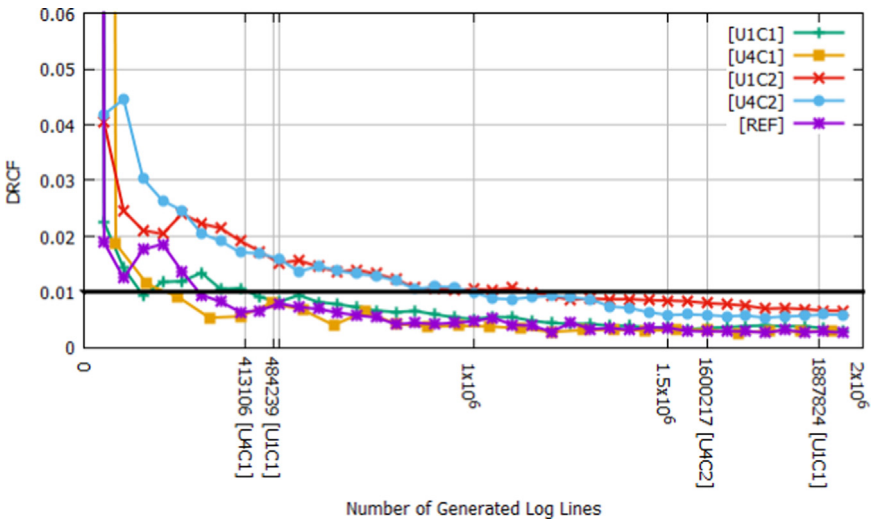

Fig. 7. Progression of the *DRCF*. The black line marks the threshold *DRCF*=0.01. Furthermore the lengths of LF_{orig} are marked.

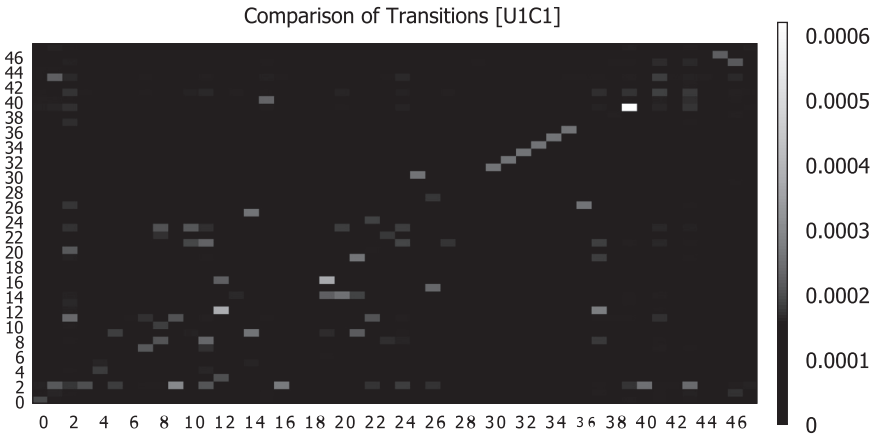

Fig. 8. Differences of the relative frequencies of the transitions between LF_{orig} and LF_{NES} with the configuration U1C1.

In Fig. 8 the darker the field of a transition is, the lower is the difference between the transitions in LF_{orig} and in LF_{NES}. Since the transition matrices are sparse (2072 of 2304 transitions are zero) most of the fields are black. The maximum difference is $\max_{i,j} t_{ij}^{diff} = 0.00062$, i.e. the maximum failure is 0.062%. This result matches with the analysis of the *DRCF*.

3.3.2. Evaluating the wild card replacement

In the following section we evaluate the quality of the wild card replacement mechanism. The wild cards are replaced by using the probability distribution which describes the relative frequency of the words which occur in LF_{orig} at the position of the wild card. As a result the relative frequency of the words replacing the wild cards is the same as in LF_{orig}.

To ensure that we do not create any log lines completely different from the lines occurring in LF_{orig}, we run the clustering algorithm again on LF_{NES}, with the same support threshold value N we used before for generating them (cf. Table 10). Also LF_{NES} have the same length as the related LF_{orig}. Afterwards we compare the clusters we obtain from LF_{orig} and the related LF_{NES}. The results of this analysis are summarized in Table 11. If LF_{NES} does not have the same length as the related LF_{orig}, the support threshold value must be modified. If LF_{NES} is longer than the related LF_{orig}, a larger number of clusters, which are more specific than the ones of LF_{orig} can be expected. To avoid this for example if LF_{NES} is twice as long as the related LF_{orig}, the support threshold value chosen for the analysis must be twice as big as the one used for generating the log file. If LF_{NES} is shorter than LF_{orig}, it is the other way around.

Columns 2 and 3 of Table 11 compare the number of clusters found in LF_{orig} and in the related LF_{NES}. For all configurations more clusters have been found in LF_{NES}. This happens, because in the log line content generation process, more similar lines can be produced, which leads to more specific and more detailed clusters. The 4th column shows how many clusters are found in both files. Between 54% and 64% of the clusters are equal. Column 5 shows how many of the different clusters found in LF_{NES} are subclusters of the clusters of LF_{orig}. A cluster is considered as a subcluster if it is more specific than another cluster. None of the clusters found in LF_{NES}, which are different, are a supcluster. A supcluster is a more generic cluster. Since we allowed SLCT to generate overlapping clusters also generic clusters (clusters, where the description only describes a small part of a log line) are found. Therefore it was predictable that there would be no new generic clusters generated. The last column points out that every cluster found in LF_{NES} describes lines of LF_{orig}. Table 11 shows that for configuration I there exist generated clusters which are different from the clusters of LF_{orig} and they are neither subclusters nor supclusters. But since all clusters describe lines of LF_{orig}, we can be sure that we have not generated a group of log lines significantly different from LF_{orig}, and big enough to form a new cluster. Furthermore a manual analysis of the cluster description shows that similar clusters can be found in the set of clusters of LF_{orig}. Moreover, the rate of outliers occurring in LF_{NES} is the same as in LF_{orig}.

3.4. Findings of the evaluation

The evaluation demonstrates that the proposed model is able to produce NES data of high quality, i.e. with realistic distribution and complexity, consuming only a low mount of resources. We further described how the optimal input parameter for the exploited clustering algorithm can be evaluated. Therefore, applying the optimal support threshold value the clustering algorithm meets two major objectives: (i) the generated cluster descriptions cover a huge part of the log lines in LF_{orig} and (ii) only a small number of outliers is detected. Even so it can be evaluated if other clustering algorithms provide better results and improve the performance.

Furthermore, we validated that the Markov chain simulation preserves the relative frequency of the clusters. Also the number of transitions relative to the log file length between two clusters remains nearly the same. This demonstrates that the log line chronology in LF_{NES} is approximately the same as in LF_{orig}. For this part, future work will test if a second order Markov chain simulation improves the results. Also different transition matrices for different time intervals can be used to raise the realism of NES data.

Table 11
The table compares the number of clusters in LF_{orig} and the related LF_{NES}. Again LF_{NES} have the same length as LF_{orig}. Furthermore it summarizes the number of clusters occurring in both log files. Moreover it is shown how many of the different clusters from LF_{NES} are sub- or supclusters of the clusters of the related LF_{orig}. The column named *match* indicates how many of the clusters from LF_{NES} describe lines of LF_{orig}.

Log file	*C. orig*	*C. NES*	*Equal-C.*	*Sub-C.*	*Sup-C.*	*Match*
U1C1	47	48	26	19	0	48
U4C1	51	56	36	17	0	56
U1C2	101	103	65	38	0	103
U4C2	119	137	78	59	0	137
Real	65	109	16	92	1	109

Finally we showed that the wild card replacement mechanism we implemented does not create completely new log lines, which would not be found in LF_{orig}. Therefore we demonstrate that every cluster, which is found in LF_{NES}, is at least a subcluster of a cluster in LF_{orig}. Furthermore every cluster found in LF_{NES} matches lines of LF_{orig}.

To improve the log line content generation the relations between consecutive log lines in a specific time interval can be further investigated. Then for example also variable parts such as IP addresses can be meaningful inserted.

Moreover the section shows that using the right support threshold value N allows us to generate NES data of high quality. The quality of the LF_{NES} is underpinned by calculating the *MCR*, the *NoO* and the *DRCF*. The high *MCR* and the low *NoO* as well as the results we obtained running the clustering algorithm again on LF_{NES} (cf. Table 11) indicate that the generated log line content fit with the log line content occurring in LF_{orig}. While calculating the *DRFC* demonstrates that the relative frequency of clusters in LF_{NES} converges towards the relative cluster frequencies in LF_{orig} with increasing number of generated log lines, Fig. 8 exemplary visualizes that also the transition matrices of LF_{orig} and LF_{NES} match each other.

4. An illustrative application

In this section we show an example of application in which we use LF_{NES} for testing and evaluating the anomaly detection system (ADS) AECID (Automated Event Correlation for Incident Detection) [14,15], which exploits log files for detecting anomalies in computer networks.

4.1. Functionality of AECID

In contrast to many other rule-based ADSs [16], which are based on blacklist approaches, AECID is a self-learning ADS which implements a white-list approach. This means that the algorithm learns normal system behavior and can afterwards recognize anomalous behavior. AECID is independent from knowledge about the semantics and the syntax of log lines. While processing log data AECID builds a *system model M*, comprising the following main building blocks [17,15]:

- *Search-Patterns*(P): Patterns are random substrings of the processed log lines which categorize the information stored in a log line.
- *Event Classes* (C): Event classes classify log lines by using the known patterns *P. Note*: One log line can be classified by more than one class.
- *Hypothesis* (H): Hypothesis describe possible implications of log lines based on the event classes classifying them.
- *Rules* (R): A rule is a hypotheses which has been proven as stable. This means the hypothesis has held in a significant time of evaluations.

The system model M (cf. Eq. (31)) is therefore defined by the set of known patterns $\mathbb{P}$, the set of known event classes $\mathbb{C}$, the set of known hypothesis $\mathbb{H}$ and the set of known rules $\mathbb{R}$:

$$M = (\mathbb{P}, \mathbb{C}, \mathbb{H}, \mathbb{R}) \tag{31}$$

The rules are used for detecting anomalies in the log data. Therefore one rule consists of a conditional event, which is specified by the class C_{cond}, an implied event, which is specified by the class C_{impl} and a time-window t_w, which either can be positive or negative. A rule evaluates to anomalous if C_{cond} occurs in the log file and the implication $C_{cond} \rightarrow C_{impl}$ does not hold in t_w.

In the following section we adapt LF_{orig} and LF_{NES} with the configuration U4C2 (cf. Table 4) for evaluating if the log data generated with our approach is suitable for testing and evaluating AECID.

4.2. Is NES data suitable to evaluate AECID?

In the following section we verify that LF_{NES} generated with our approach is suitable to test and evaluate AECID. AECID can run on LF_{NES} since it is independent from the syntax and the semantics of its input log data. Moreover we intend to show that LF_{NES} can be used to evaluate and test AECID in a specific user environment, which is characterized by LF_{orig}. We first ran AECID on LF_{orig} with the configuration U4C2 (cf. Table 4) and then on LF_{NES} we generated based on LF_{orig} with the support threshold value N as given in Table 10 ($N = 12801$). Since AECID depends on the log file length, both log files consist of 1.600.217 log lines (cf. Table 4).

To evaluate if LF_{NES} is suitable for testing AECID under conditions of a real network environment characterized by a real log data probe, we use for AECID the basic configuration given in [15].

To decide if LF_{NES} is suitable to test and evaluate AECID, we focus on two statistics relevant for assessing AECID's performance:

(i) Average Line Coverage *ALC*.
(ii) False Positive Rate *FPR*.

The *ALC* is calculated as shown in Eq. (32); it is the ratio between the Average Number of Enforced Patterns *ANEP* in

Table 12
Results for the *ALC* and the *FPR*, when running AECID with the basic configuration on LF_{orig} and LF_{NES} based on the configuration U4C2.

-	*ALC* LF_{orig}	*ALC* LF_{NES}	*FPR* LF_{orig}	*FPR* LF_{NES}	$\lvert\Delta ALC\rvert$	$\lvert\Delta FPR\rvert$
Mean	17.5831	18.5233	0.0526	0.0546	0.9402	0.0020
Median	17.7767	18.3686	0.0403	0.0407	0.5919	0.0004
Minimum	15.1145	16.7241	0.0025	0.0088	1.6096	0.0063
Maximum	19.9802	21.6642	0.1329	0.1922	1.6840	0.0593

Table 13
Steps of the roll-out process of an IDS [18,19].

I. SELECTION	**EVALUATION**	Type (e.g., HIDS, NIDS), performance, capabilities (logging, detection, prevention), technical support, scalability
II. DEPLOYMENT		Architecture design (e.g., location), staged deployment, component tests, personnel training, configuration, components security
III. OPERATION		Maintenance, update, tuning, alert handling, alert response, alter configuration, periodical verification and optimization

the event classes $\mathbb{C}$[2] and the percentage of enforced patterns ϕ_e[3] in every class $C \in \mathbb{C}$:

$$ALC = \frac{ANEP}{\phi_e}. \tag{32}$$

The *ALC* specifies how many patterns $P \in \mathbb{P}$ match those log lines on average that triggered the creation of a new class. Therefore it is an indicator for the number of patterns covering every log line on average. Moreover it provides more knowledge about the set of patterns $\mathbb{P}$ and the set of classes $\mathbb{C}$ than the total number of generated patterns and classes.

The *FPR* is usually calculated as shown in Eq. (33). The *FPR* is the ratio between the number of anomalous rule evaluations if no anomaly occurred, i.e. false positives *FP*, and all rule evaluations. The number of rule evaluations is the sum of the *FP* and the true negatives *TN*, i.e. all not anomalous rule evaluations if no anomaly occurred:

$$FPR = \frac{FP}{FP + TN}. \tag{33}$$

Since we consider both LF_{orig} and LF_{NES} as anomaly free, the *FPR* is simply the ratio between all anomalous rule evaluations and all rule evaluations. Therefore it can be called anomalous evaluation rate.

Table 12 shows the results of the analysis of the *ALC* and the *FPR*, when running AECID with the basic configuration on LF_{orig} and LF_{NES}. Since AECID uses pseudo random numbers for picking patterns and generating classes and hypotheses, we executed it 100 times with the same configuration and then calculated the mean, the median, the minimum and the maximum of the results.

First we focus on the *ALC*. The *ALC* for LF_{orig} is on average 17.5 patterns and for LF_{NES} it is around 18.5 patterns. The median of both files is even closer than the mean. Both the minimum and the maximum *ALC* in LF_{NES} are slightly larger than the *ALC* values obtained with LF_{orig}. In both cases the range between the minimum and the maximum value of the *ALC* is around 4.9. On average the difference between the *ALC* of LF_{orig} and the *ALC* of LF_{NES} is less than 1 pattern. The table also shows that according to the *ALC* the algorithm performs slightly better with LF_{NES}. This can be explained by the fact that LF_{NES} is based on more deterministic conditions.

Since the *FPR* is a ratio it is given in percent. The average *FPR* obtained with LF_{NES} is just 0.2% higher than the *FPR* obtained with LF_{orig}. The gap between the median values is only 0.04%. The minimum and the maximum value of both files show that the range of the *FPR* is quite large. The results show that AECID is not deterministic, because of the influence of the pseudo random numbers, which are used to control the generation patterns, classes, hypotheses and rules. But since AECID implements a self-learning approach and the tested log files only map 10 h in real time this dependency would be lowered by training the algorithm with longer log files.

According to the *ALC* and the *FPR* values AECID obtains very similar results for both LF_{orig} and LF_{NES}. This proves that it is possible to effectively test and evaluate AECID's performance in a network environment with the characteristics of LF_{orig}, by using NES data generated with our approach.

4.3. Experiences with AECID

After verifying that our generated NES data is suitable for testing and evaluating AECID, it can also be used for identifying the optimal configuration of AECID for a specific network environment. Therefore the seed value for generating pseudo random numbers in AECID should be fixed to provide comparable results. Then the input parameters of AECID can be changed and applied in various combinations. The *FPR* and the *ALC* then can be exploited to decide, which is the optimal configuration. Since the considered log data should be anomaly free (however otherwise the *FPR* of every generated rule has to be compared) the *FPR* should be low and the *ALC* high.

Also attacks for testing the attack detection capability of AECID can be simulated with our approach. For example to simulate an attacker that tries to access the data base without being detected, the logging function of the data

[2] Every event class *C* enforces a number of patterns *P*, which have to occur in a log line classified by *C* (cf. [15]).

[3] ϕ_e is one of the AECID's input parameters specifying, which percentage of patterns matching to the log line processed during the generation of the class *C* has to be enforced in the class *C* (cf. [15]).

base server would be disabled. Therefore the generation of log lines related to the data base server can be suppressed in a specified time interval. To perform more complex attacks also the transition matrix can be modified for a specified time interval.

5. Roll-out of an IDS

In this section we define the roll-out of an intrusion detection system (IDS) within a medium or large-scale enterprise IT environment in a step-by-step set-up process to show how much effort is required to achieve this. The two standards [18] published by the International Organization for Standardization (ISO) and [19] published by the National Institute of Standards and Technology (NIST) serve as starting point. Other reports discussing roll-out of IDSs following similar approaches are [20–22]. Referring to this procedure we point out which steps can be simplified and optimized with our approach.

The roll-out of an IDS can be structured as shown in Table 13. According to the standards [18,19] the three main parts of the set-up process are selection, deployment and operation. Furthermore evaluation is part of all these three steps. Table 13 also summarizes which criteria are considered when and which actions are performed.

In the roll-out process the **EVALUATION** of IDSs is the biggest challenge. Reasons for this on the one hand are that no standardized methodologies for testing IDSs exist and on the other hand there are no standard test environments for IDSs available [19]. Hence organizations depend on vendor brochures, white-papers and product demonstrations, which are usually not objective and therefore insufficient and also on third-party reviews of individual products and comparisons of multiple products. Since every network infrastructure is different also tests in lab environments are insufficient to rate the performance of IDSs in specific environments. This circumstances force organizations to perform evaluations on their running productive systems, which might expose the infrastructure to dangerous or unstable situations [18,19]. Therefore, as shown in Section 3 our novel approach allows to generate high quality NES data, which enables detailed simulation of an organization's network infrastructure which then can be exploited for extensive evaluations.

5.1. Selection

Since there exist various IDSs, the first step of enrolling a product is **SELECTION**. Therefore on the one hand the criteria summarized in Table 13 and on the other hand the system environment, IDS security policies and financial costs build a basis for selecting the optimal IDS candidates. Decisions based on system environment, IDS security policies and financial costs as well as on the type of the required IDS and the provided technical support have to be made by ICT security experts. The other criteria account for with evaluation methods as mentioned in the beginning of this section. In this context our approach allows us to evaluate the performance of IDSs as partly shown in Section 4. In opposite to testing an IDS directly in an organization's network infrastructure our approach also allows us to evaluate the scalability of a product, for example by rescaling the time differences between consecutive log lines, which simulates a larger volume of network traffic. The scalability of a product is important, because otherwise in case an organization's network infrastructure grows a new IDS solutions has to be selected. Our approach can be also applied for testing capabilities of a product. For example to evaluate the detection capability of an IDS the system it monitors has to be attacked. In Section 4.3 we discussed how cyber attacks can be simulated with our approach, which is important since it is not advisable to verify the effectiveness of an IDS in a network environment by self-attacking it.

5.2. Deployment

After selecting an IDS the **DEPLOYMENT** process starts. First an architecture of the IDS implementation has to be designed, which includes specifying the locations of sensors and also interactions with other system components are taken into account. Further both standards [18,19] recommend a staged deployment, i.e. deploying an IDS first only for a small part of a network and then expanding it incremental, which makes it easier for the staff to acquire insights into new products. Furthermore component tests simplify the evaluation of new products and lower the risk of problems during the deployment phase. Our approach also allows performing off-line component tests with highly realistic NES data. Also the personnel has to be trained to get familiar with new IDS solutions. NES data makes it possible to accomplish this training within a sandbox environment and outside a running productive system. A major point of deployment is identifying the optimal configuration of the deployed IDS, i.e. the configuration that addresses the highest risks of the organization. The configuration should also be part of the selection phase, since it strongly influences the performance and effectiveness of an IDS. Since the configuration heavily depends on the network infrastructure – every network is different – it cannot be evaluated in a laboratory environment or based on published tests and comparisons of vendors. Here our generated NES data offers the advantage that on the same highly realistic data set several configurations can be easily and fast tested and compared, which enhances and accelerates the configuration process tremendously – partly shown in Section 4.

5.3. Operation

After the deployment the **OPERATION** phase follows. In this context operation among other things covers maintenance, updating, alert handling and alert response, and also tuning the IDS as well as altering its configuration. Therefore periodic verification and evaluation has to be performed to continuously optimize the IDS. Updates, for example, can implicate altering the configuration. Therefore it is possible to test the updated software first with generated NES data, which allows us to adopt the configuration immediately. Since also the monitored infrastructure usually underlies frequent changes, it is possible

to periodically generate new NES data files with our approach easily and fast to continuously verify and evaluate the performance and effectiveness of the deployed IDS. If required the IDS and configuration can be tuned and optimized accordingly. Also threats are evolving over time and therefore our approach allows by simulating new threats in NES data files off-line testing of the detection capabilities of the applied IDS.

This demonstrates that our approach can be applied to simplify and optimize the roll-out of an IDS; hence it can be exploited especially for evaluation – which more or less is part of the whole process – configuration and optimization.

6. Background and related work

Data logging has a widespread application area, also beyond the enterprise Information and Communication Technology (ICT) sector, e.g. embedded systems for runtime verification. For example clinical information log files have been used for automated identification of patient specific clinician information needs [23]. Also in space engineering, log data is for example used to perform runtime verification [24].

In the ICT sector log files and log data analysis are used for various purposes. For example log data can be used for error analysis [25]. In this context also the analysis of empirical and statistical properties of failure and error logs is important [26]. Furthermore log data is investigated for digital forensics [27]. The field of digital forensics is widespread. In this area also research for modern technologies, including cloud computing [28,29] and file sharing [30] is done. Moreover log data is exploited by frameworks for ensuring accountability [31]. Database logs for example can be used to back up and restore database content in case of a system crash or a destruction caused by an unauthorized access violation [32,33]. On the other hand Web log data forms the basis of mining the interests of users of a Website [34]. In this context for example frequent pattern mining in log files is performed [35]. Also security- and computer network analysis tools such as anomaly detection [36] and intrusion detection systems [16] exploit log files to identify anomalous behavior. Additionally log files are a major part of system monitoring [37], which is also important for modern technologies, e.g. mobile devices [38]. Since cars and other vehicles get smarter and smarter also monitoring systems are invented to track a drivers actions [39].

Especially testing and evaluating tools for security- and computer network analysis on running productive systems raises two major disadvantages: on the one hand, during this period the productive system cannot be secured properly and on the other hand private informations are exposed, which ends up in violations of privacy. Furthermore, a running productive system must be attacked for evaluating if a security solution can detect anomalous behavior at all. Due to this fact this cannot be done reasonably. On the other hand, tests under conditions similar to those in a laboratory environment are usually not realistic enough, because of the missing complexity produced by the noisy network base load. Therefore, highly realistic test environments for sandbox-benchmarking are required, which allow us to evaluate security mechanisms outside of running productive systems. Some examples for such sandboxes are:

- *Virtual Security Testbed* (*ViSe*) [40]: ViSe is a virtual environment, which allows going back to a former snapshot, if a system got infected by malware or was put out of action.
- *Lincoln Adaptable Real-Time Information Assurance Testbed* (*LARIAT*) [41]: LARIAT was the first attempt, to invent a comprehensive test environment for IDSs.
- *Lincoln Laboratory Simulator* (*LLSIM*) [42]: LLSIM is a completely virtual further development of LARIAT, implemented in Java [43]. LLSIM offers a customizable test environment, in which hundreds of components of standard hardware can be simulated.
- *Testbed for Evaluating Intrusion Detection Systems* (*TIDeS*) [44]: TIDeS is a test environment, which tries to quantify the evaluation process, to choose a suitable IDS for a specific network environment.
- *Cyber Defense Technology Experimental Testbed* (*DETER*) [45]: Among cyber-security scientists DETER is one of the leading test environments. It was invented within a collaboration of the National Science Foundation, the US Department of Homeland Security, UC Berkeley and McAffee Research.

The presented test environments all follow network centralized approaches, but a lot of security- and computer network analysis tools operate on a higher log-level-layer [46,47]. Therefore, functionalities are needed, which also simulate human users' behavior. The approach presented in [13] for example simulates a simple network, where virtual users perform specified actions on a Web platform and log files from monitoring a Web server, a database and a firewall are produced.

Simulating complex networks usually consumes a lot of physical resources and requires a high financial budget. Furthermore it is exceptionally hard to model SCADA systems (due to strict temporal constraints) or the hardware of mainframe networks. Therefore cost-efficient and easy to use approaches for simulating ICT networks are required.

Our proposed approach uses a log line clustering algorithm to provide knowledge about the properties of the network which should be modeled. Traditional clustering methods [48,49] work well for data with numerical attributes in low-dimensional data spaces, with dimension k below 10. But (i) the attributes of a log line are of categorical nature and (ii) the data space in which a log line is represented, can be high-dimensional, since the number of words a log line consists of is not limited. Problem (ii) occurs, because traditional clustering methods do not detect clusters, which exist in subspaces of the original data space [50]. Nowadays various algorithms for clustering high-dimensional data with categorical attributes exist. Examples are CLIQUE [50], MAFIA [51], PROCLUS [52] and CACTUS [53], which all try to avoid the above-mentioned problems, which can occur using traditional clustering methods. But, most of the existing high-dimensional clustering algorithms for data with categorical attributes are not applicable for clustering log file data. The main reason is that most of these algorithms,

including those mentioned above, do not take into account the basic characteristics of log files. Therefore we decided to apply the clustering algorithm first published in [1], which was especially invented for clustering log lines.

For generating synthetic log data based on the results of log line clustering we use results of the Markov chain theory [8,54]. Methods for simulating Markov chains can be found in [11,55]. Markov chain models have been used in various application areas to make forecasts. In the meteorology they have been used for example to predict wind speeds [56] and rain days [57,58]. Also supermarket inventory systems have been simulated applying a Markov chain model [59]. In the genetic research DNA sequences have been analyzed using the Markov theory [60,61]. In software engineering Markov chains are used for statistical testing of software [62]. These are just a few examples, where Markov chains have been applied to model sequences of events. Therefore it is reasonable to consider Markov chains for generating sequences of log lines.

7. Conclusion and future work

In this paper we presented a novel approach for generating network event sequence (NES) data for sandbox-benchmarking of security- and network analysis tools. The approach takes as an input a small set of log data obtained from a real network environment. On the input data first a log line clustering algorithm is applied and then a Markov chain simulation is performed. While log line clustering enables classification of log lines and generating log line description, Markov chain simulation guarantees realistic log line sequences. This model allows the generation of log files of any size and configurable complexity based on the properties of a specific network environment. To verify the effectiveness and high quality of the so generated log data we first performed a qualitative evaluation. Therefore we introduced novel metrics such as the mean coverage rate *MCR* and the difference of the relative cluster frequencies *DRCF*. To prove the similarity between the original and the generated log file we executed the clustering algorithm on the generated log data and compared the clusters obtained with the original and those obtained with the generated log data.

Next to this qualitative evaluation, we also presented quantitative evaluation results. Therefore we first applied an anomaly detection system called AECID on the NES data as an illustrative example of an application for the generated NES data. We proved that the NES data is feasible for testing, evaluating and discovering suitable configurations of AECID. Therefore we executed AECID with LF_{orig} and LF_{NES} and compared the average line coverage *ALC* and the false positive rate *FPR*. Moreover we discussed the concept, how the optimal configuration of AECID can be evaluated and how attacks can be simulated using our approach for generating NES data. Additionally we defined a step-by-step enrollment process for IDSs in real environments and in accordance to major standards from ISO and the NIST and argued, which parts can be simplified and optimized applying the proposed approach.

The proposed model is composed by several modules, which can be flexibly improved or replaced. For example SLCT, the log line clustering algorithm we used can be exchanged with other clustering methods. Furthermore the Markov chain simulation can be tuned. Different transition matrices for different time periods could be utilized since the network behavior changes over the day (nobody is working during the night and also update and backup processes are mostly done during the night). Moreover the creation of a log line can depend not only on one preceding log line, but also on a set of preceding log lines.

Our work has important implications on future methods to design, evaluate and run anomaly detection systems, but also to test network performance tools, and other tools that process log data. On the one hand testing in a running productive system is complex, errorprone and risky. Therefore evaluation in an active ICT network creates negative impact on the system stability. This leads to the approach implementing sanboxes to perform off-line tests. Hence the question how to get realistic stimuli has to be answered. The proposed model therefore demonstrates a novel approach for generating NES data. On the other hand testing in a running productive system raises privacy issues, since user names, passwords, IP addresses, etc. get exposed. Thus log files first have to be sensitized. Since the proposed model only needs a small piece of log data as input, this can be done manually. Afterwards based on the input a sensitized NES data file of any size can be generated, which then can be used for off-line tests under highly realistic conditions.

Acknowledgments

This work was partly funded by the European Union FP7 Project ECOSSIAN (607577) and carried out in course of a master thesis at the Vienna University of Technology.

References

[1] R. Vaarandi, A data clustering algorithm for mining patterns from event logs, in: 3rd IEEE Workshop on IP Operations Management, 2003. (IPOM 2003), 2003, pp. 119–126. http://dx.doi.org/10.1109/IPOM.2003.1251233.
[2] J. Stearley, Towards informatic analysis of syslogs, in: 2004 IEEE International Conference on Cluster Computing, 2004, pp. 309–318. http://dx.doi.org/10.1109/CLUSTR.2004.1392628.
[3] R. Vaarandi, SLCT Version 0.05, ⟨http://ristov.users.sourceforge.net/slct/⟩, 2007.
[4] R. Vaarandi, Mining event logs with slct and loghound, in: Network Operations and Management Symposium, 2008. NOMS 2008. IEEE, Salvador, Bahia, Brazil, 2008, pp. 1071–1074. http://dx.doi.org/10.1109/NOMS.2008.4575281.
[5] R. Vaarandi, K. Podiņš, Network ids alert classification with frequent itemset mining and data clustering, in: 2010 International Conference on Network and Service Management (CNSM), IEEE, Niagara Falls, Canada, 2010, pp. 451–456.
[6] R. Balakrishnan, K. Ranganathan, A Textbook of Graph Theory, Universitext (Berlin. Print), Springer, New York, 2012.
[7] B. Everitt, S. Landau, M. Leese, D. Stahl, Cluster Analysis, Wiley Series in Probability and Statistics, Wiley, Chichester, UK, 2011.
[8] J.R. Norris, Markov Chains, no. 2, Cambridge University Press, New York, USA, 1998.
[9] C.M. Grinstead, J.L. Snell, Introduction to Probability, American Mathematical Society, Swarthmore, USA, 1997.
[10] A. Klenke, Wahrscheinlichkeitstheorie, vol. 1, Springer, Berlin, Germany, 2006.
[11] O. Häggström, Finite Markov Chains and Algorithmic Applications, vol. 52, Cambridge University Press, Cambridge, UK, 2002.

[12] N. Kusolitsch, Maß-und Wahrscheinlichkeitstheorie: Eine Einführung, Springer-Verlag, Vienna, Austria, 2011.
[13] F. Skopik, G. Settanni, R. Fiedler, I. Friedberg, Semi-synthetic data set generation for security software evaluation, in: 2014 Twelfth Annual International Conference on Privacy, Security and Trust (PST), IEEE, Toronto, Canada, 2014, pp. 156–163.
[14] I. Friedberg, F. Skopik, R. Fiedler, Cyber Situational Awareness Through Network Anomaly Detection: State of the Art and New Approaches, 2015.
[15] I. Friedberg, F. Skopik, G. Settanni, R. Fiedler, Combating advanced persistent threats: from network event correlation to incident detection, Comput. Secur. 48 (2015) 35–57.
[16] S. Axelsson, Intrusion Detection Systems: A Survey and Taxonomy, Technical Report, Technical report Chalmers University of Technology, Goteborg, Sweden, 2000.
[17] F. Skopik, I. Friedberg, R. Fiedler, Dealing with advanced persistent threats in smart grid ict networks, in: Innovative Smart Grid Technologies Conference (ISGT), 2014 IEEE PES, IEEE, Washington, USA, 2014, pp. 1–5.
[18] I.-T.R. ISO, Iso/iec 27039, Information Technology—Security Techniques—Selection, Deployment and Operations of Intrusion Detection Systems.
[19] K. Scarfone, P. Mell, Guide to intrusion detection and prevention systems (idps)(draft), NIST Special Publ. 800 (2012) 94.
[20] J. Snyder, Guide to Network Intrusion Prevention Systems, PCWorld, ⟨http://www.pcworld.com/article/144634/guide_network_intrusion_prevention_systems.html⟩, 2008.
[21] P. Innella, O. McMillan, D. Trout, Managing Intrusion Detection Systems in Large Organizations, Part One, International Series in Operations Research & Management Science, Symantec, ⟨http://www.symantec.com/connect/articles/managing-intrusion-detection-systems-large-organizations-part-one⟩, 2010.
[22] E. Yakabovicz, Intrusion Detection System Deployment Recommendations, TechTarget, ⟨http://searchfinancialsecurity.techtarget.com/tip/Intrusion-detection-system-deployment-recommendations⟩, 2008.
[23] E.S. Chen, J.J. Cimino, Automated discovery of patient-specific clinician information needs using clinical information system log files, in: AMIA annual symposium proceedings, vol. 2003, American Medical Informatics Association, Washington, USA, 2003, p. 145.
[24] H. Barringer, A. Groce, K. Havelund, M. Smith, Formal analysis of log files, J. Aerosp. Comput. Inf. Commun. 7 (11) (2010) 365–390.
[25] T.-T.Y. Lin, D.P. Siewiorek, Error log analysis: statistical modeling and heuristic trend analysis, IEEE Trans. Reliab. 39 (4) (1990) 419–432.
[26] R.K. Sahoo, M.S. Squillante, A. Sivasubramaniam, Y. Zhang, Failure data analysis of a large-scale heterogeneous server environment, in: 2004 International Conference on Dependable Systems and Networks, IEEE, Florence, Italy, 2004, pp. 772–781.
[27] S. Raghavan, Digital forensic research: current state of the art, CSI Trans. ICT 1 (1) (2013) 91–114.
[28] T. Sang, A log based approach to make digital forensics easier on cloud computing, in: 2013 Third International Conference on Intelligent System Design and Engineering Applications (ISDEA), IEEE, Hong Kong, 2013, pp. 91–94.
[29] K. Ruan, J. Carthy, T. Kechadi, I. Baggili, Cloud forensics definitions and critical criteria for cloud forensic capability: an overview of survey results, Digit. Invest. 10 (1) (2013) 34–43.
[30] C. Quinn, M. Scanlon, J. Farina, M.-T. Kechadi, Forensic analysis and remote evidence recovery from syncthing: An open source decentralised file synchronisation utility, Digital Forensics and Cyber Crime, Springer, Seoul, South Korea, 2015, 85–99.
[31] S. Sundareswaran, A.C. Squicciarini, D. Lin, Ensuring distributed accountability for data sharing in the cloud, IEEE Trans. Depend. Secure Comput. 9 (4) (2012) 556–568.
[32] R. Fang, H.-I. Hsiao, B. He, C. Mohan, Y. Wang, High performance database logging using storage class memory, in: 2011 IEEE 27th International Conference on Data Engineering (ICDE), IEEE, Hannover, Germany, 2011, pp. 1221–1231.
[33] P. Frühwirt, P. Kieseberg, S. Schrittwieser, M. Huber, E. Weippl, Innodb database forensics: reconstructing data manipulation queries from redo logs, in: 2012 Seventh International Conference on Availability, Reliability and Security (ARES), IEEE, Prague, Chzech Republic, 2012, pp. 625–633.
[34] T. Murata, K. Saito, Extracting users' interests from web log data, in: Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence, IEEE Computer Society, Hong Kong, 2006, pp. 343–346.
[35] R. Iváncsy, I. Vajk, Frequent pattern mining in web log data, Acta Polytech. Hung. 3 (1) (2006) 77–90.
[36] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: a survey, ACM Comput. Surv. (CSUR) 41 (3) (2009) 15.
[37] S.E. Hansen, E.T. Atkins, Automated system monitoring and notification with swatch, in: LISA, vol. 93, 1993, pp. 145–152.
[38] A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, A. Markopoulou, Antmonitor: a system for monitoring from mobile devices, in: Proceedings of the 2015 ACM SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data, ACM, London, UK, 2015, pp. 15–20.
[39] R.S. Ling, R.A. Hutchinson, W.J. Steigerwald III, W.A. Say, P.L. O'malley, D.A. Shrallow, W.C. Everett, R.J. McMillan, Vehicle Monitoring System, US Patent 8,140,358, March 20, 2012.
[40] M. Richmond, Vise: A Virtual Security Testbed, University of California, Santa Barbara, Technical Report.
[41] L.M. Rossey, R.K. Cunningham, D.J. Fried, J.C. Rabek, R.P. Lippmann, J. W. Haines, M. Zissman, et al., Lariat: Lincoln adaptable real-time information assurance testbed, in: Aerospace Conference Proceedings, 2002, vol. 6, IEEE, Big Sky, Montana, USA, 2002, pp. 6–2671.
[42] J.W. Haines, S. Goulet, R.S. Durst, T.G. Champion, et al., Llsim: Network simulation for correlation and response testing, in: Information Assurance Workshop, 2003, IEEE Systems, Man and Cybernetics Society, IEEE, New York, USA, 2003, pp. 243–250.
[43] Oracle Corporation, Java Development Kit Version 8 Update 45, ⟨http://www.java.com⟩, 2015.
[44] G. Singaraju, L. Teo, Y. Zheng, A testbed for quantitative assessment of intrusion detection systems using fuzzy logic, in: Information Assurance Workshop, 2004. Proceedings. Second IEEE International, IEEE, Charlotte, North Carolina, USA, 2004, pp. 79–93.
[45] T. Benzel, The science of cyber security experimentation: the deter project, in: Proceedings of the 27th Annual Computer Security Applications Conference, ACM, Orlando, FL, USA, 2011, pp. 137–148.
[46] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, M. Rajarajan, A survey of intrusion detection techniques in cloud, J. Netw. Comput. Appl. 36 (1) (2013) 42–57.
[47] R.P. Menon, Log analysis based intrusion prediction system, in: Emerging ICT for Bridging the Future-Proceedings of the 49th Annual Convention of the Computer Society of India (CSI), vol. 1, Springer, Hyderabad, India, 2015, pp. 409–416.
[48] P. Berkhin, Grouping multidimensional data, A Survey of Clustering Data Mining Techniques, Springer, Berlin, Germany, 2006, 25–71.
[49] R. Xu, D. Wunsch, et al., Survey of clustering algorithms, IEEE Trans. Neural Netw. 16 (3) (2005) 645–678.
[50] R. Agrawal, J. Gehrke, D. Gunopulos, P. Raghavan, Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications, vol. 27, ACM, New York, USA, 1998.
[51] S. Goil, H. Nagesh, A. Choudhary, Mafia: Efficient and scalable subspace clustering for very large data sets, in: Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1999, pp. 443–452.
[52] C.C. Aggarwal, J.L. Wolf, P.S. Yu, C. Procopiuc, J.S. Park, Fast algorithms for projected clustering, in: ACM SIGMoD Record, vol. 28, ACM, Philadelphia, USA, 1999, pp. 61–72.
[53] V. Ganti, J. Gehrke, R. Ramakrishnan, Cactusclustering categorical data using summaries, in: Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, San Diego, USA, 1999, pp. 73–83.
[54] R. Serfozo, Basics of Applied Stochastic Processes, Springer Science & Business Media, Berlin, Germany, 2009.
[55] W. Ching, M. Ng, Markov Chains: Models, Algorithms and Applications, International Series in Operations Research & Management Science, Springer, New York, USA, 2006.
[56] S.P. Kani, M. Ardehali, Very short-term wind speed prediction: a new artificial neural network-Markov chain model, Energy Convers. Manag. 52 (1) (2011) 738–745.
[57] C. Haan, D.M. Allen, J. Street, A Markov chain model of daily rainfall, Water Resourc. Res. 12 (3) (1976) 443–449.
[58] R. Singh, C. Patel, M. Yadav, P. Singh, K. Singh, et al., Weekly rainfall analysis and markov chain model probability of dry and wet weeks at Varanasi in Uttar Pradesh, Environ. Ecol. 32 (3) (2014) 885–890.
[59] C. Poudyal, D.N. Khanal, S. By, R. Stockbridge, A Discrete Time Markov Chain Model in Supermarkets for a Periodic Inventory System with One Way Substitution.
[60] H. Almagor, A markov analysis of dna sequences, J. Theoret. Biol. 104 (4) (1983) 633–645.
[61] A. Hobolth, A markov chain monte carlo expectation maximization algorithm for statistical analysis of dna sequence evolution with neighbor-dependent substitution rates, J. Comput. Graph. Stat.
[62] J. Whittaker, M.G. Thomason, et al., A markov chain model for statistical software testing, IEEE Trans. Softw. Eng. 20 (10) (1994) 812–824.